CODE GENERATION ALGORITHMS FOR DIGITAL SIGNAL PROCESSORS

Guido Costa Souza de Araújo

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING

June, 1997

Abstract

The dramatic reduction in the cost of electronic devices combined with large improvements in design productivity due to the use of automatic tools are gradually opening up the possibility for high-performance, very low cost computation. This is reflected in the increasing demand for high-performance portable devices which have low power consumption and cost. The best way to design such systems is to use dedicated architectures. *Digital Signal Processors* (DSPs) are specialized architectures which can provide these features for applications which require intensive numeric computation, like those running in telecommunication systems. This thesis deals with code generation for DSPs. It addresses two main problems in this area. The first is the generation of code for program basic blocks. The second is the problem of code generation for address computation. This work makes two contributions for the basic block code generation problem. First, it proposes a model for the processor architecture which captures the impact of the datapath design in the code generation algorithm. Based on this model, it proves the existence of an optimal $O(n)$ algorithm for expression tree code generation for a class of DSP architectures. Experimental results are used to confirm the correctness of the algorithm. The contribution of this thesis for the address generation problem is an algorithm for the allocation of array references to address registers, which maximizes the use of the registers and reduces the cost of addressing operations in the program. Experiments are performed which show that the proposed optimization can considerably improve the final addressing code, when compared with code from the best optimizing compiler available for the target DSP.

Acknowledgments

I thank my wife Silvana, and my daughter Ana for their support, and most of all, for their understanding during the long periods of absence in which algorithm analysis had higher priority than a walk in the park.

I am indebted to my advisor, Sharad Malik, for providing me with the opportunity to work under his supervision, and for his involvement, enthusiasm and valuable advice throughout the course of my research.

I would like to acknowledge the comments and suggestions that I received from the readers of this manuscript, Sharad Malik, Andrew Appel, Margaret Martonosi, and Srinivas Devadas, which helped improve its clarity and contents.

A word of thanks also goes to the staff of the Electrical Engineering Department for the support I have received during my stay at Princeton.

I thank all my friends who created, in the midst of the turmoil of their own deadlines, opportunities to share friendship and kindness. They made life at Princeton a remarkable and pleasant experience.

I dedicate this thesis to my wife, daughter and parents for their love and support all these years. *Obrigado pessoal...*

This research was supported by the National Council for Technological and Scientific Development of Brazil (CNPq), Grant 204033/87-0 and by the Institute of Computing, UNICAMP, Brazil.

GUIDO ARAÚJO

Contents

Chapter 1

Introduction

1.1 Shifting the Computation Focus: Text, Speech and Image

The communication between human individuals is based on three main forms of expression: text, speech and image. They are representations of our thoughts which are used to communicate an idea, express our feelings (e.g. art), and describe the world surrounding us.

Information carried by text is bounded by the words, the style and the ability of the writer to express his (her) ideas. Speech opens up the possibility for vocal intonation and symbolic expressions, bringing to the listener a more detailed picture of the speaker's thoughts and feelings.

Text and speech enable abstract and discursive manipulation. Both are based on the usage of signs (letters and phonemes) which work as atomic components in building more complex forms of expression, like statements and phrases. An *image* does not allow the same possibilities though. Although an image can sometimes carry much more information than equivalent text and speech, it does not hold the semantic structure required to describe the interaction of the objects it represents. For example, an image showing people talking, watching TV, and reading books will

hardly be enough to describe the arguments discussed on these paragraphs. On the other hand, many pages of text or discourse would be required, just to describe the scene the reader can observe through his (her) window right now.

Text, speech and image are complementary expressions of our thoughts and ideas. They play different roles in different aspects of human interaction. Individually they are not enough to replicate our complete sensorial experience though. Otherwise, books, radio and movies would not coexist today as they do. Therefore, it is natural to believe that at some point of the human development the technologies associated to these forms of communication would seek to merge in a way which resembles our own experience.

The technologies for text, speech and image have been developed separately in time. Gutenberg's press (1450), Bell's telephone (1876), Marconi's radio (1909), and Baird's television (1926) are parts of the history. These technologies are continuously evolving, but they have always used very distinct means of communicating the information, or in other words very distinct *media*. Never before in human history, has merging them into a single framework seemed as possible as it is today. Because of that the best word to describe this convergence is probably *multimedia*. Its long term effects on our lives are as difficult to anticipate, as were the consequences of the press, the telephone, and the TV for our ancestors.

Two important breakthroughs have enabled the techniques required for the design of today's multimedia systems. The first is the availability of inexpensive and fast computers which allow a uniform and reliable representation of distinct media. Computers are designed with digital circuits, which use transistors as their building blocks. History teaches us that every 50 to 70 years revolutionary new ideas mature to the point where they can be deeply incorporated into people's life. The invention of the transistor 50 years ago (Shockley, Bardeen and Brattain, 1947) marks the beginning of such an era, and its effects have already had an important influence on how

society works. These devices are getting faster and cheaper at an astonishing rate, permitting the design of inexpensive and easy-to-use computer systems. The second enabling technology is the advancement in telecommunications, which, although in a slower pace, has produced a wave of new innovations (e.g. fiber-optics). These two technologies form the driving force behind the revolution called *multimedia*.

One of the big consequences of this revolution is the change in the way computers are designed and used. The computation focus has moved from centralized (e.g. mainframes) to distributed (e.g. networks), and from general-purpose[1] to specialized and user oriented (e.g. laptop computer). A couple of decades ago, computing was associated with numeric processing of scientific applications. With the introduction of the personal computer, symbolic and textual manipulation became the focus of the efforts in computer development. The emergence of multimedia has added speech and video to the scene, increasing the diversity of the computing landscape. Nowadays, text, speech, image and numeric data live together. The difference in the amount of computation required by each of these data types is promoting a redistribution of the computation power in a computer system. This has had a major impact in the way these systems are designed. The effort put in designing a computer today is gradually moving towards architectural styles that can improve, not only the performance of numeric and symbolic applications, but also the performance of applications which manipulate speech and images. This can be observed in the increasing usage of dedicated audio and video processors, and in the introduction of specialized functionality into architectures which were not capable of efficiently handling that before. Intel Multimedia Extensions (MMX) [1], Sun Visual Instruction Set (VIS) [2] and MIPS Digital Media eXtension (MDMX) [3] are examples of efforts on this direction. In

[1]The term *general-purpose computer* here will be associated to computer systems which were not designed with a specific application in mind. One example is a *desktop computer*.

order to clarify the role played by these new data types in modern computer systems[2], it is important to understand the type of applications which use them, and its performance requirements.

1.2 New Objects, Increasing Requirements

There are two basic problems associated with the processing of speech and image. The first is the problem of manipulating the information. Typical examples of that are: speech recognition, video browsing, virtual reality and graphics design. The second problem has to do with its transmission. It is important that the data sent at one end of the link is received at the other end, without much degradation and within an adequate time frame.

The computational power required by these problems are tremendous, and some of them can be barely tackled with the technology available today. The main reason behind that is the stringent time constraints associated with speech and image. Unlike text and speech, image information has to be processed within specific time limits determined by our senses. The processing of information within the time limits imposed by its environment is usually called *real-time* computing. Consider for example the speech coding algorithm used in a cellular telephone. This algorithm transforms the analog signal created by the voice into its digital encoded form, which is much more immune to noise and amenable to processing [4]. Coding algorithms are divided into two routines, one to *encode* the analog signal in the transmitter, and another to *decode* the digital signal in the receiver. It is reasonable to expect that the time required for the decoding algorithm to decode an incoming word (i.e. latency) is small enough, such that the listener is not capable of noticing the delay. On the other hand, the algorithm execution must also be fast enough, such that it finishes

[2]The term computer system here does not mean exclusively a *desktop computer*, but any system designed with a processor.

processing the first word before the next one arrives (i.e. it has to allow for high *throughput*).

The throughput required by a cellular telephone is a consequence of the amount of data needed to encode the voice, and of the time allowed for the device to process it. The ratio between these two quantities is known as *bandwidth*. Consider for example the case of audio. The human ear can recognize sounds with frequencies up to 20 KHz (for speech this is only 4 KHz). Nyquist's theorem states that the sampling rate of a signal needs to be at least twice its highest frequency [4]. If each sample is encoded using a *byte*, then the required bandwidth for audio[3] is **40 Kbytes/sec** (8 Kbytes/sec for speech). Bandwidths even higher than that are necessary for images. The human vision needs at least 30 *frames per second* for the recognition of a smooth video image. A typical image has 352 x 240 *pixels*. If a pixel is encoded using 12 bits, then a single video frame will need 127 Kbytes. Since at least 30 frames have to be displayed per second, then the resulting bandwidth is **3810 Kbytes/sec**. For a comparison on the magnitude of these numbers, consider the case of transmitting a single page of text. The page containing these words has roughly 1000 letters which can be easily encoded into the same number of bytes. Assuming even that this page has to be read in one second, the required bandwidth is only **1 Kbyte/sec**. The numbers above clearly reflect the relative amount of information available in text, speech and image as discussed before. Moreover, they give a good indication of the relative computational effort needed for the real-time processing of each one of these data types.

The high bandwidths demanded by audio and image processing can only be managed by means of compression techniques. The idea is to reduce the amount of data needed to encode the information. Decreasing the size of the data to be processed

[3]High-quality stereo sound requires higher bandwidths. For example, CD stereo sound has a bandwidth of 175 Kbytes/sec.

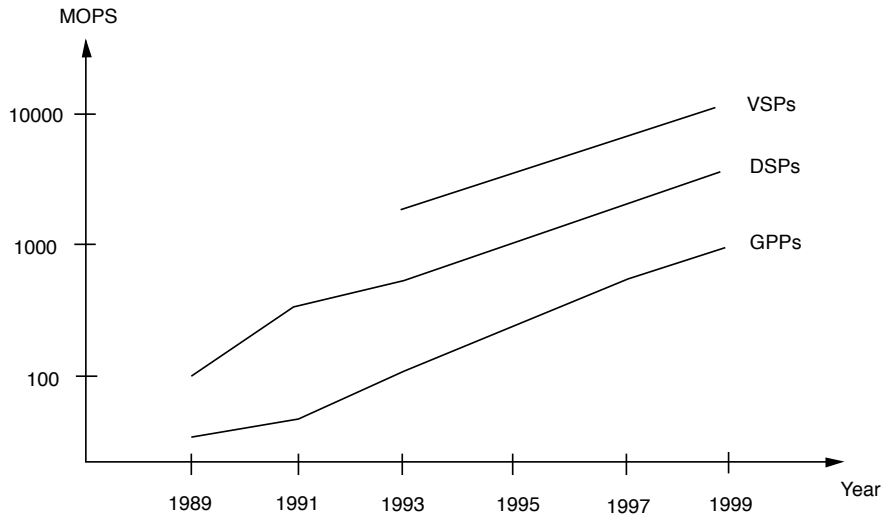


Figure 1.1: Trends in the performance of programmable processors (Bhaskaran V. and Konstatinides K., *Image and Video Compression Standards*, Kluwer Academic Publishers, Chapter 8, pp. 210)

.

brings two benefits. First, more time is available to finish processing the information (e.g. to filter the noise). Second, reduced system resources are used to store the data (e.g. smaller memory size). Many compression standards have been proposed to address that. They are, for example: ADPCM [5] for audio, JPEG [6] for still images, and MPEG for video [7]. These standards are very representative of typical multimedia applications, and can give a clear idea of the computational power required by each of these data types.

For the purpose of comparison, consider only the encoding part of the standards. JPEG needs close to 23 MOPS[4] [8] to compress a still image, while MPEG requires around 5000 MOPS [8] to encode a video stream. The graph in Figure 1.1 shows the trends in the performance of different architectures that can be used for these tasks. In the graph, GPP stands for *General Purpose Processor*. This category encompasses the modern microprocessors used for general-purpose applications, like those running

[4]MOPS stands for *Million Operations per Second*

on our desktop computers. Dedicated microprocessors, known as *Application Specific Processors* (ASPs), have largely been used to provide computational power when GPPs are not up to the task. Unlike their general-purpose cousins, ASPs are designed with a particular set of applications (or *application domain*) in mind. Two classes of ASPs have been used in the audio and image domains: Digital Signal Processors (DSPs) and Video Signal Processors (VSPs). The reasons for specialized processors can now be understood. Observe that although GPPs are capable to perform JPEG encoding (352 x 240 pixels image) in less than a second, they cannot provide the computation power required by MPEG encoding, which can only be achieved by a VSP. It is certainly true that, in the future, GPPs will be able to handle MPEG, but when this occurs the demand for more advanced applications will again require more powerful resources, that only domain specific architectures can provide. Architectural specialization has been used for other reasons besides providing better performance. Understanding these reasons brings to light another important change in the way computers are used today.

1.3 Specialization and Programmability

The dramatic reduction in the cost of silicon area, and the big improvements in productivity due to automatic design tools, are gradually opening up the possibility for high-performance, very low-cost computation. The increasing demand for portable and cheap computing devices in consumer electronics is a reflection of this change. The design of such devices has to satisfy many other constraints besides high performance. The two major ones are *low cost* and *low power consumption*. In order to meet these constraints, architecture designers have to make better usage of the silicon. The final goal is to increase the computation performed per unit of silicon area ($MIPS/mm^2$), while reducing the amount of power required to run the application

($watts/MIPS$). This is best achieved by architectural specialization.

Consider, for example, the case of an architecture which contains a functional unit capable of counting, in a single machine cycle, the number of active bits in a register. The best algorithm to count bits in a GPP architecture would use a table to map the contents of the register to the number of active bits it stores. This approach would use too much memory space though. Since the goal is to minimize cost consider an implementation which does not use so many memory resources. An algorithm based on shift and mask operations can count the number of bits in four steps: (1) shift the register containing the data; (2) mask out all but the least significant bit of the register; (3) add the resulting bit to the accumulator register; (4) if the register is not zero, jump to (1). Assuming that each step takes one machine cycle, this algorithm would take at least four cycles in a GPP. Thus, the performance improvement using the specialized architecture is four-fold, assuming that both architectures have similar machine cycles. If the application makes intensive usage of such algorithm, this result not only in a considerable speedup, but also in a smaller design, and in less power consumption. The GPP architectural solution would use an entire ALU and register file to complete the task while an specialized architecture could do it with a smaller combinational unit and fewer registers. Also every algorithm step in a GPP implies in fetching and decoding instructions from memory what considerably adds up to the final power consumption of the application.

Adding specialized features to a design improves its performance, at the expense of increasing design complexity, and hence of making programming difficult. The trade-off between adding features for improved performance and its impact in programmability was studied before in the case of RISC[5] architectures [9]. In the design of a RISC processor the goal is to improve the performance of the *common case* instructions. The *common case* in a general-purpose (i.e. *common*) application leads

[5] *Reduced Instruction Set Computer.*

very naturally to a regular architectural style. This is not the same for ASP architectures, where the dedicated architecture reflects an application domain in which many specialized operations take place. Dedicated functionality is also a trend which has been observed in other areas of computer architecture, as in *custom computation* [10, 11, 12, 13]. In any case, the rule of improving the *common case* performance can also be applied, but unlike for RISC processors, it results in a very specialized and irregular architecture.

The main impact of architectural specialization shows up when code needs to be produced by the compiler, a task called *code generation*. Unfortunately, the ability of compilers to capture the existence of specialized functionality is very restricted. There are two major reasons for this. First, identifying special functional units requires bringing much architectural information into the compiler. This is not desirable given that it does not preserve *retargetability*. Retargetability is the ability of the compiler to be easily modified such that it can be used for another architecture. Retargetability is a extremely desirable feature in the ASP world, where the design of new processors for new applications is very common. The best approach to address this is probably the use of *domain compilers*, i.e. compilers which are easily retargetable within a specific application domain (e.g. DSP applications). The second problem caused by specialization is the lack of compiling algorithms for these architectures, and the computational hardness of the known ones. More research needs to be done in order to improve the understanding of the problems in this area.

The conclusion which can be drawn from the above discussion is that applications which demand very high computation performance at low cost, usually require specialized architectures. This in turn frequently impacts programmability. The trade-off between performance and programmability spans a range of solutions. For example, Figure 1.2 shows a pictorial graph where many architectural options are considered in the implementation of a particular audio application. They range from

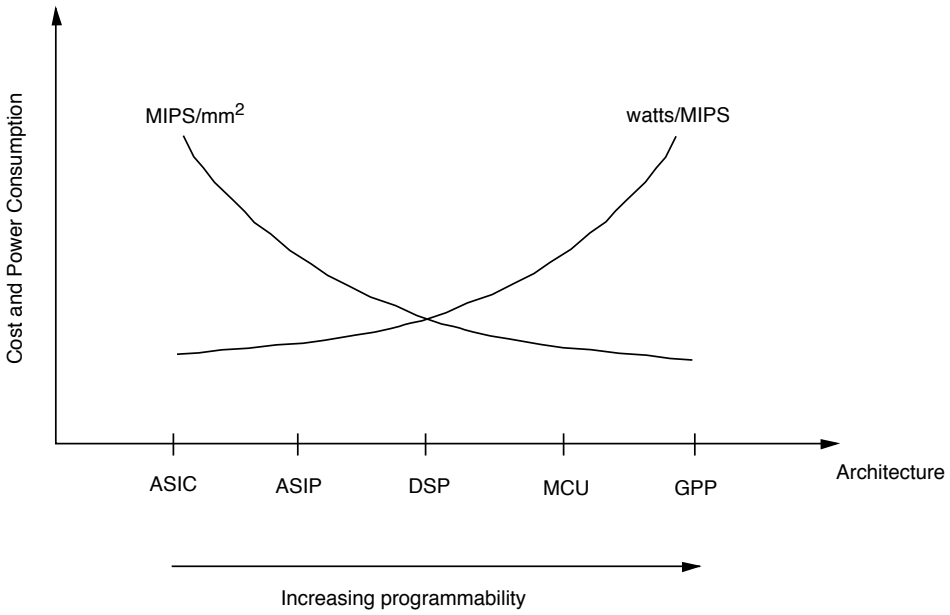


Figure 1.2: Implementation of a DSP application using different architectural styles.

a complete hardware solution using an *Application Specific Integrated Circuit* (ASIC) to a GPP. An *Application Specific Instruction Set Processor* (ASIP) is an ASP architecture where the hardware and instruction set are designed together to implement a very specific algorithm. *Microcontroller Units* (MCUs) are smaller versions of GPPs, which are used in many industrial applications. The architectures are ordered based on increasing programmability. The graph shows two curves. The first ($MIPS/mm^2$) is determined dividing the effective computation performed by the application by the area of the processor, while the second ($watts/MIPS$) represents the average power used to perform that computation[6]. The curves reflect the effective computation per unit of cost (area and power consumption) that can be achieved by each architecture. From Figure 1.2 one can observe that, for an audio application, DSPs are the best solution since they counterbalance low cost/power and programmability. DSPs are specialized microprocessors designed to speed up applications that require extensive

[6]Assume here that the instructions in each architecture perform approximately the same amount of effective computation.

real-time numerical computation, like audio applications. This thesis considers the application domain of DSPs.

1.4 The DSP Application Domain

DSPs have been used in a number of areas beyond standard signal processing. They can be found in telecommunications (e.g. speech encoding), computer graphics (e.g. 3D rotation), instrumentation (e.g. seismic processing), automotive control (e.g. engine control), consumer electronics (e.g. cellular telephone), and signal processing (e.g. signal filtering).

The growth of the DSP market has demonstrated that these devices are starting to play a very important role in the electronic industry of today (circa 1996). The global DSP market is expected to grow 40% per year [14], while the personal computer market is starting to level off at 14% [15]. These numbers reveal the importance DSPs will have on the design of the future electronic systems. The emergence of DSPs is primarily driven by the explosion in the telecommunication and multimedia markets, and by the increasing participation of these technologies in all areas of human endeavor.

Given the diversity of the applications in the DSP domain, a complete characterization of the *typical DSP application* is a hard task to perform. Fortunately, these applications share a number of well defined characteristics. First, the input data, i.e. the subject of the computation, is usually a sequence of samples from some sort of digitized signal, originating from microphones, sensors, antennas, video cameras, etc. Second, applications require extensive numeric computation, which must be performed under very stringent time constraints. This happens because they usually run on real-time systems such as cellular telephones, radar, sonar, graphic cards and others, which demand high throughput and very low latency. Another important

characteristic of these applications has to do with the type of systems they are used in. It is increasingly common to find DSP applications in consumer electronic devices, for which low cost and power consumption are important requirements.

A typical DSP application, like those found in many real-time systems, uses a simple repetitive algorithm to process its input. The application typically reads a subset of the input sequence, storing it into a memory array. It then performs many arithmetic operations on the array elements using program loops to compute the required operation. Finally, it starts another computation cycle by reading the next subset of the input sequence. Many different arithmetic operations are used in the design of the algorithm which performs the desired computation on the input sequence. From all these operations, there is one that plays an important role because it is present in the majority of the DSP programs: the *inner product*. The *inner product* of two arrays A and B, $i = 1, \ldots N$, is the quantity $< A, B > = \sum_i^N A_i.B_i$.

DSPs are largely used in systems where GPPs are not capable of meeting domain specific constraints. In the case of portable devices, for example, the power consumption and cost may make the use of GPPs prohibitive. The same is true when high-performance arithmetic processing is required to implement dedicated functionality, at low cost, as in the case for many consumer electronic products. The cost difference between a DSP and a GPP can be very large. For example, a Motorola DSP-56004 processor costs \$16.80 [16], whereas the cost of the cheapest Intel Pentium processor is \$184 [16] (circa 1995). The small cost achieved by a DSP is predominantly due to its small silicon area and from the fact that die costs are proportional to the third (or higher) power of the die area [9]. The die size of a typical DSP ranges from $25mm^2$ to $60mm^2$ [17]. In contrast, the die size of a GPP ranges from $160mm^2$, for an Intel Pentium, to $250mm^2$, for a Sun SuperSparc [18].

DSP architectures make efficient usage of the silicon area by only providing the functionality required in their application domain. As the name implies, GPPs are

targeted to a broad class of problems, and therefore must provide the functionality required by a large number of different applications. This results in large designs which use much silicon area.

Although the first DSPs attained surprisingly good performance [19], the difference in performance between modern DSPs and GPPs has considerably shrunk since then. Actually the performance of high-end GPPs running DSP applications can sometimes match the one achieved by modern DSPs (circa 1996). For example, the execution of a 1024 points FFT[7] (*Fast Fourier Transform*) in a DEC 21064 processor takes $T = 480\mu s$ [20]. The same application in a ADSP-21060 DSP takes $T = 460\mu s$ [21] to execute. The reduced performance gap has become the major argument for those who advocate that DSPs are fated to disappear [20]. Current initiatives to incorporate DSPs functionality into GPP architectures seem to endorse that. An example is the Intel *Native Signal Processing* (NSP) initiative [1]. The major idea behind NSP is to bring into GPP designs typical DSP functional units, like multiply and accumulate (MAC), thus allowing DSP applications to run efficiently, altogether with standard general purpose applications. The alternative to this approach is to use DSPs as co-processors coupled to a host GPP. It is certainly true that NSP will ease the design and integration of DSP applications into standard *desktop computers* and *workstations*. Some questions remain open though. It is still not clear if such systems, with their intrinsic time-sharing nature, will be able to meet the real-time requirements of typical DSP applications. As was mentioned previously, the listener of a network conversation will not tolerate a noticeable delay in the incoming voice, caused by an *interrupt* during the execution of the speech decoding routine. Signal processing issues related to general-purpose computing are an important research problem, but unfortunately it is far from the scope of this thesis.

[7] FFT is a typical DSP algorithm used to convert signals from time to frequency domains.

Processor	Area (mm^2)	Relative Area	Power (Watts)	Relative Power
DEC-21064	234	2.9	30	7.5
ADSP-21060	60	0.7	3.6	0.9
x486DX	81	1	4	1

Table 1.1: Absolute and relative silicon area (A) and consumed power (P) when comparing the DEC-21064 and the ADSP-21060 processors using the x486DX as a reference.

The majority of DSPs are not used in standard desktop computers, but on systems which require the combination of: *low cost*, reduced *power consumption* and *high performance*. This is the subject domain of this thesis. In this domain, the comparison between DSPs and GPPs reveals a very different perspective. Assume that both processors can execute the application program within the required latency and throughput. The figure of merit, which will be used for the comparison, can be defined as $F = A^3 \times P$, where A is the silicon area and P its power consumption relative to a reference machine. It reflects the fact that the smaller F, the smaller the values of A and P, and the more suitable the processor will be to run that application. Area is used here as a measure of the device cost in order to factor out market fluctuations. F is proportional to the third power of A because this reflects the actual impact of the die size in the final chip cost [9]. Intel 486DX processor was used as a reference machine for the purpose of comparison. The absolute and relative values of these quantities for each processor are represented in Table 1.1 [21, 22].

Applying the numbers in Table 1.1 into F results in $F = 183$ for the DEC-21064 and $F = 0.3$ for the ADSP-21060, when both run the 1024 point FFT algorithm. It is clear that the ADSP-21060 is a better processor choice for a system designed under these cost considerations. Metric F is not a definite metric of the type of applications studied here, but it provides a satisfactory first approximation.

1.5 Compiling for DSPs

Computer programs are usually written in a high-level language (e.g. C), which is converted by a compiler into some sort of *Intermediate Representation* (IR) data structures. The code generation algorithms inside a compiler convert the IR data structures into a sequence of machine instructions from the target processor. A very large set of optimization techniques [23] are commonly used in this process to minimize the execution time and the amount of memory used by the final executable program (i.e. *object code*). Compiling techniques is a very active area of research and development, and a considerable amount of interesting work has already been done here. Unfortunately, not much effort has been directed towards the understanding of the code generation problem for ASPs, and DSPs in particular, even considering that ASPs encompasses the majority of processors in use today[8]. The main reason for that is probably the small size of the programs running on these processors, which permits assembly programming. Unfortunately, assembly programming is a cumbersome and time consuming task, which is not enough for the design of the large DSP applications of today. The combination of that with the increasing time-to-market pressures has raised the demand for the use of high-level languages for these systems as well. A recent survey by a large DSP *design-house* [25] has ranked the compiler the most important tool missing from the designer's arsenal.

The focus of this thesis is on the understanding of the problems and on proposing solutions for code generation for a class of DSPs. The work here is part of a larger project, known as *The SPAM Project* [26].

The goal of the SPAM project is the design of a retargetable compiler for DSP processors. Due to the low cost and low power constraints discussed before, the code

[8]The share of GPPs produced annually is only 6%, while ASPs represent 94% of the total number of processors produced every year (1994) [24].

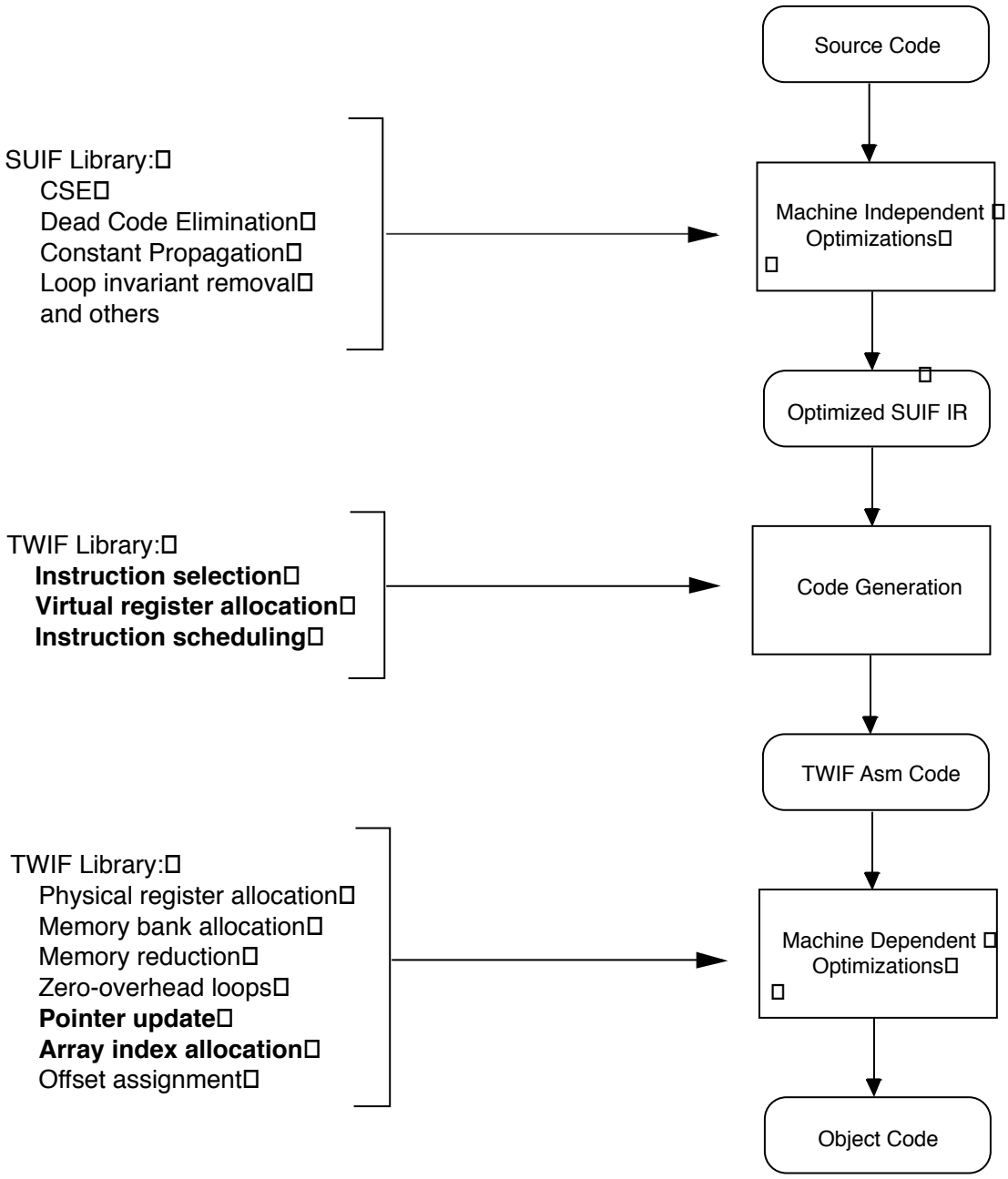


Figure 1.3: The SPAM compiler flow; the research problems addressed by this thesis are highlighted.

quality demanded by DSP applications is very high. Thus, the emphasis of the research is on the design of optimizations techniques which can help meet this quality. Due to the above constraints, DSP architectures have limited *on-chip* program memory. In this case, reducing code size becomes an important objective function in the design of such optimizations, given that a single extra bit can result in the code not fitting on the chip.

The SPAM compiler execution flow is represented in Figure 1.3. The front-end is based on SUIF[9] [27], a compiler framework which is used to convert the program into an IR, and to perform basic machine independent optimizations [23]. Due to their specialized features, code optimizations for DSPs, and ASPs in general, are very machine specific. Hence, the SPAM compiler performs code generation, including *instruction selection*, *virtual register allocation* and *instruction scheduling* early in the compilation flow, before any machine dependent optimization is performed. After code has been scheduled, machine dependent optimizations take place. This includes the final allocation of physical registers to the virtual registers used by the final code. Many optimizations have been designed for the SPAM project [28, 29, 30, 31, 32], which deal with various idiosyncrasies present in these processors. All optimizations developed in the SPAM project are part of a library called TWIF. The TWIF library contains pre-designed optimization modules which are needed for the implementation of a new DSP compiler.

This thesis addresses some of the problems in compiling for DSPs. It proposes solutions for basic block code generation for a class of DSP architectures, and it investigates optimizations for address computation on these architectures. The results of this research were implemented into the TWIF modules highlighted in Figure 1.3.

[9]Stanford University Interchange Format.

1.6 Organization of this Thesis

In conclusion, DSP applications require extensive arithmetic computation capability combined with small cost and low power, and this can only be achieved by means of specialized architectural features. The immediate consequence of that is an irregular architecture, which usually makes compiling a very difficult task.

The architectural style found in DSPs is described in detail in Chapter 2. The important features which are common to all DSPs are analyzed there, with emphasis on the impact they have in the code generation task. That chapter also proposes an architectural model based on the topology of the processor datapath. The idea is to isolate the datapath features which are important to the task of code generation. The proposed architectural model is then used in Chapter 3 to investigate the problem of basic block code generation for DSPs. The main contributions of that chapter are two: an optimal $O(n)$ code generation algorithm for a class of DSP processors, when the basic block is an expression tree, and a heuristic otherwise. To the best of our knowledge, this was the first time an algorithm for this problem was proved optimal for such architectures. The experimental results show that improved code can be produced when this approach is used. Another important problem in compiling for DSPs is the generation of address code. Chapter 4 investigates this problem. It proposes a general approach for the allocation of address registers to pointer variables and array references in a program. The idea is to maximize the usage of the address registers available in DSPs, and to minimize the generation of addressing code. The experimental results show that improved code can be obtained when comparing with the code generated by the best optimizing compiler for the target processor. Finally, Chapter 5 lists the main contributions of this thesis and proposes new avenues on how to extend the work did here, and Appendix A describes the interface of the TWIF modules which implement the proposed algorithms.

Chapter 2

Architectural Model

2.1 DSP Architectural Features

Because of the hard performance constraints found in the DSP application domain, the final goal of a DSP design should be to enable the architecture to perform an arithmetic operation every machine cycle. In order to achieve that, the processor architecture is carefully designed such that the instruction cycle time equals the cycle time of the hardware, for the majority of the instructions. Ideally this implies that the machine should be able to perform four operations each machine cycle: (1) *fetch* an instruction; (2) compute the address of the operands of the instruction; (3) *load* the instruction's operands; and (4) perform the actual computation. Obviously this can only be achieved by means of *Instruction Level Parallelism* in combination with specialized hardware units and memory organization.

DSP architectures can be classified according to the type of data they process as *fixed-point* and *floating-point* DSPs. In applications running on a fixed-point DSPs, programmers are responsible for scaling the result of the integer operations. This is automatically done in floating-point DSPs. Floating-point units are extremely costly in terms of silicon area and clock cycles. Usually, for the same technology, a fixed-point DSP consumes much less power and is $\approx 50\%$ faster [19] than the equivalent

floating-point implementation. Therefore even though it relieves the programmer from the burden of scaling, floating-point DSPs are not the best choice when low power and high performance are simultaneous system requirements. For this reason, the majority of the systems based on DSPs use fixed-point DSPs. This class of architectures will be the focus of this thesis. Hence, the term *DSP* will refer from now on to *fixed-point DSP*.

This chapter describes the features which make a typical DSP architecture. The goal here is not to describe implementation details of each feature, but to analyze its impact on the code generation task. The interested reader should refer to Lee's survey papers [19, 33] for an extensive comparison between a number of DSP architectures. An updated version of this work can also be found in [34]. Annual reports on all commercial DSPs, including benchmark analysis of typical applications[1], are also available (e.g. [35]).

2.1.1 Memory System

The demand for high performance of DSP applications typically requires one instruction to be executed for each machine cycle. In order to achieve that, *memory space* is divided into *program memory* and *data memory*, which can be accessed simultaneously using separate buses. This allows for an instruction to be fetched while the previous instruction is executed. This architecture style is known as *Harvard architecture*.

DSPs have *on-chip* data memory, based on fast static RAMs, and *on-chip* non-volatile program ROM. Because of its high-performance requirement these memories are designed to have *single cycle access times*. Unlike general-purpose architectures, DSPs are not designed with cache or virtual memory systems, since data and program streams usually fit into the available *on-chip* memories. Because *on-chip* memories are

[1]This is the Berkeley Design Technology (BDT) benchmark, which is based on DSP program kernels.

fast and cache misses are not an issue, some DSPs are based on *memory-register* architectures (e.g. TMS320C1x/C2x/C5x/C54x). In this architecture, binary instructions take one operand from memory, and the other from some register R in the datapath, storing the result into A (Figure 2.1(a)). In memory-register DSP architectures, the memory system is formed by a single memory bank, and since instructions directly read one operand from memory, the instruction latency is sometimes large.

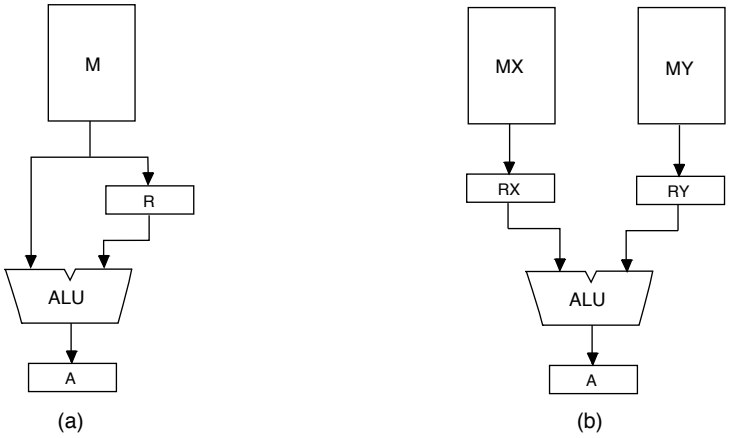


Figure 2.1: (a) Memory-register architecture using single bank; (b) Register-register architecture using dual banks.

With the increase in the size of the *on-chip* memory, latency has become considerably high, what can substantially degrade the clock cycle of the processor. In order to avoid that, modern DSP designs use *register-register* (or *load-store*) architectures, where operands are first loaded into registers before the computation takes place, or memory-register architectures with paged memories (see Section 2.1.3 for details). Due to the desirable single cycle requirement, all operands of an instruction in a register-register DSP are fetched from data memory simultaneously. In order to achieve this required bandwidth, the data memory space is divided into multiple banks (typically two), as in the case of the Motorola 56000 [36] and of the ADSP 2100 [37]. Variables in different memory banks are accessed in parallel, using dedicated addressing units, for improved performance (see Chapter 3 for details). In

a typical case, operands of the next instruction are fetched in parallel during the execution of the current instruction. Since the majority of the instructions in any application require two operands, it is easy to understand why dual memory banks is the most common form in this type of design. This organization (Figure 2.1(b)), combined with *register-register* instructions can be found in many DSP architectures [36, 37]. They work as follows. During the execution of an instruction, the ALU computes the required operation using the current contents of registers RX and RY, while the new contents of these registers are read from memory banks MX and MY. The result of the computation is stored into register A.

2.1.2 Datapath Architecture

The goal in the design of a DSP datapath is to implement those functional units which can speed up costly operations that frequently occur in the processor's application domain. The datapath typically has a very irregular topology, targeted to maximize the usage of functional units and registers and at the same time minimize the total silicon area and power consumed by the processor.

Functional Units

DSPs usually have a set of very specialized functional units. The reason, as mentioned before, is that DSP applications frequently make intensive usage of special operations, like the inner product computation of two arrays ($< A, B > = \sum_i A_i B_i$). The inner product can be efficiently performed by a unit which computes the product $a_i.b_i$ while accumulating the previous product $a_{i-1}.b_{i-1}$ into a register (known as *accumulator*). For this reason this unit is called the *Multiply ACcumulate* (MAC) Unit. Almost all DSP architectures have MAC units. MAC units can be implemented in a single separate module [37], or by a combination of the processor's Multiplier and ALU

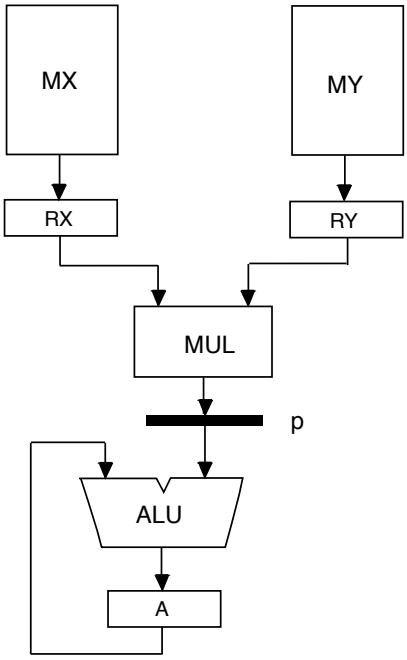


Figure 2.2: Pipelined Multiply and Accumulate unit (MAC).

[38, 36]. The functional unit of Figure 2.2 is a typical example of a pipelined MAC unit. There, the multiplication of the operands in RX and RY is stored into register p, while the previous contents of p is accumulated into a. The pipeline stage register (e.g. p register in Figure 2.2) can be exposed to the ISA, as in the case of the TMS320C25 [38] or not, as in the ADSP 2100 [37]. Observe that the MAC unit permits the result of the multiply and accumulate operations to occur with a throughput of one machine cycle, while in a non-superscalar GPP this would take 2 cycles, corresponding to two separate instructions.

The majority of DSP architectures also have *shifter* units, which are used to perform numerical scaling, bit extraction, extended precision arithmetic and overflow prevention. Shifters are usually located between the output of a register or a data bus, and a functional unit (e.g. SHIFTER_2 at output of register p in Figure 2.3). They are used by instructions to select a shifting operation for a particular operand before the

main operation is performed. Consider, for example, the case when the programmer wants to round the contents of a register before the operation using it is performed. *Scaling* is a very common operation in fixed-point DSPs, when the exponents of the numbers are adjusted before and after an operation. Some DSPs have specialized shifting operations and instructions (e.g. NORM in the NEC μPD77016) which can speed up the scaling task.

Datapath Interconnection

Due to design requirements, DSP designers frequently constrain the interconnectivity between registers and functional units. There are two main reasons for that. First, the desired functionality usually requires a particular datapath topology. Second, broad interconnectivity translates into datapath buses and/or muxes, which results in increased cost and instruction performance degradation. The datapath of the TMS320C25 processor is a typical example of the restricted interconnectivity present in DSPs. Observe in Figure 2.3, that register p is not allowed to load the contents of t. This would require using one more multiplexer at the input of p which adds to the final cost of the device. In this case the designers could not identify any use for this connection, since t can be loaded into p by multiplying its contents by one, an instruction which takes a single machine cycle.

Register Set

The majority of DSPs have a small number of specialized register files (1 - 4) containing few (1 - 8) registers each. The specialized nature of the register files is again a consequence of the high performance and low cost requirements of these devices. Due to register specialization, some DSP architectures (e.g. TMS320C25) have instructions which require operands to be located in specific register files. Many of these instructions store the result of the computation into yet another register file, different

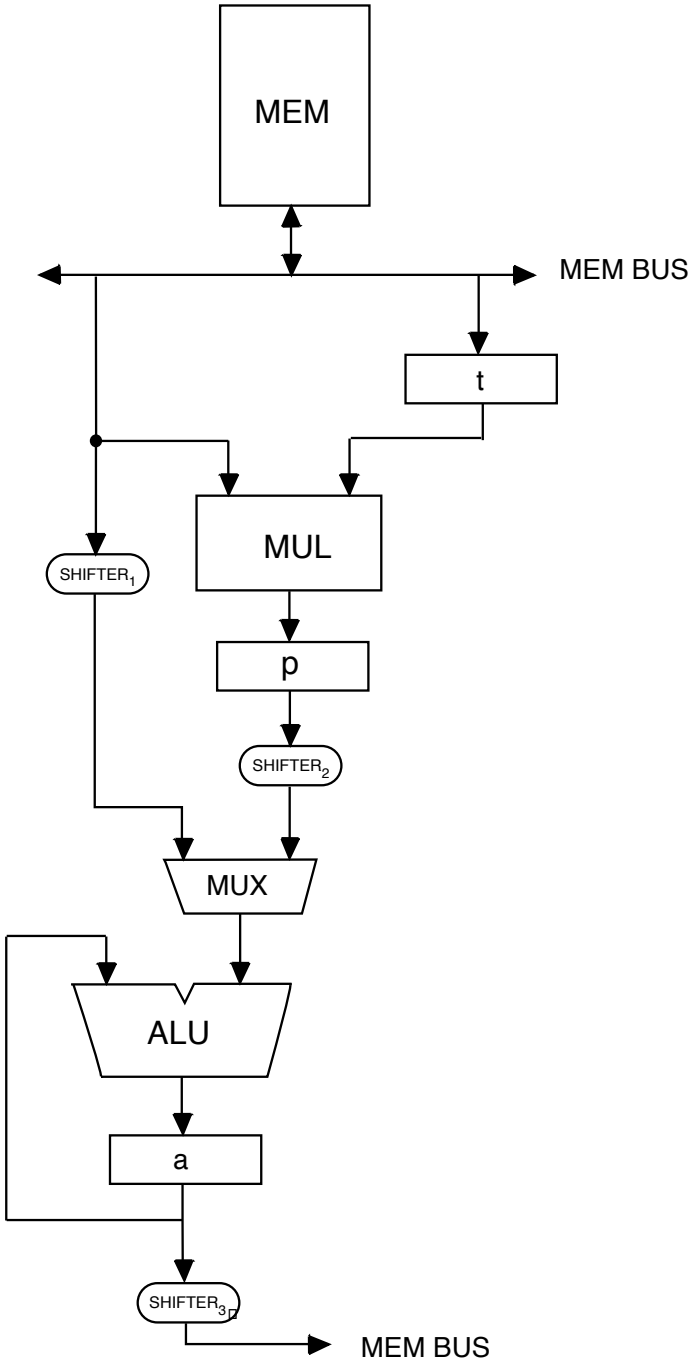


Figure 2.3: The TMS320C25 datapath architecture.

from those required for its operands. Typically, register files used by operands and result are different. Because of that, many DSPs are known as *heterogeneous register architectures*. For example, in the architecture of Figure 2.3, one can think of registers a, p, t as three register files of size one each. Each of these (size-one) register files has a particular functionality. Consider, for example, register p. The result of a multiplication instruction can only be stored into register p and in no other register. Also the two operands of the multiplication must be located in memory and in register t.

Unlike DSP architectures, instructions in GPPs usually do not restrict the registers they use, provided they come from the same register file (hence operand registers are typically homogeneous). This considerably simplifies the code generation problem, since it decouples the tasks of instruction selection from register allocation. Due to this property GPPs are known as *homogeneous register architectures*. Some DSPs are homogeneous register architectures. Consider, for example, the NEC μPD77016 [39]. This DSP has a single register file and instructions which can simultaneously read two operands and store the result of the computation into the register file. The main disadvantage of this design is certainly the need for a *multi-ported* register file[2], which requires a larger silicon area and consumes more power. Floating-point DSPs are also a case where homogeneous register architectures are used (e.g. TMS320C3x/4x), but in this case, as mentioned before, cost and low power consumption are not as stringent requirements as for fixed-point devices.

Pipeline

DSP architectures are pipelined (typically 3 - 4 stages). During the *Instruction Fetch* stage (*IF*), an instruction word is read from the program memory. In the *Instruction Decode* stage (*ID*) the instruction is decoded and its operands are read from the data

[2]A multi-ported register file is a register file which allows more than one *load/store* operation to occur at the same time.

memory and/or registers. Finally, during the *Execute* stage (*EX*) the instruction computation is performed. Pipelining efficiently speeds up the throughput of a DSP, without almost no degradation in latency. A pipeline *hazard* is a situation where an instruction in the pipeline cannot complete its task due to resource conflicts (*structural hazard*), lack of operand availability (*data hazard*) or because the fetched instruction belongs to the control path which was not taken by the program (*control hazard*) [9].

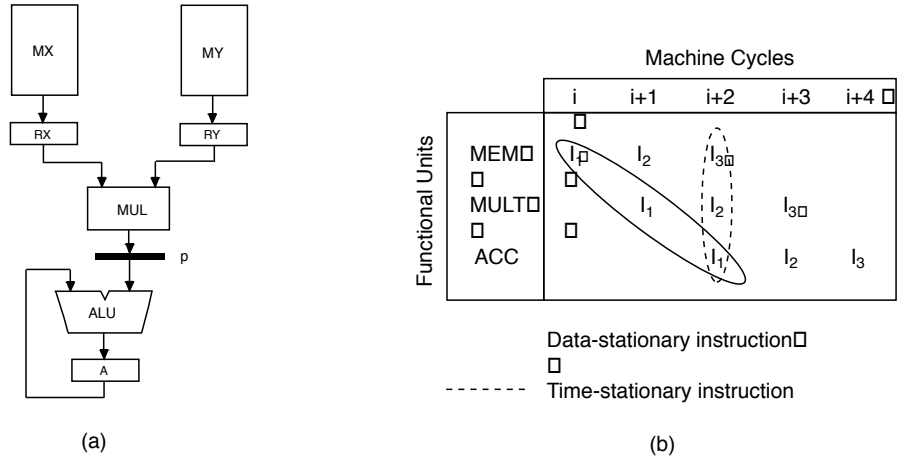


Figure 2.4: (a) Pipelined Multiply and Accumulate (MAC) unit; (b) MAC reservation table using different code styles.

There are three fundamental techniques to deal with pipeline hazards in programmable processors: *interlocking*, *time-stationary* coding and *data-stationary* coding [33]. The idea behind interlock is that the programmer should not be bothered with the internal timing of the pipeline or parallelism of the architecture. The programming model for an architecture using interlock assumes that each instruction completes before the next instruction begins. If hazards occur when the processor tries to meet this requirement, the pipeline *stalls*. In order to eliminate the hazard, the processor may have to discharge some instructions which have already entered the pipeline. This increases the number of cycles per instruction that is executed. Texas Instruments DSPs use interlocking techniques [38, 40, 41].

Two coding techniques are usually employed when the pipeline is exposed to the programmer: time-stationary and data-stationary. Consider for example the pipeline section of a MAC unit in Figure 2.4(a). The registers in the figure show a three stage *data-pipeline*: *load* operands, multiply (*mul*) and accumulate (*acc*). Figure 2.4(b) shows the reservation table associated with three *mac* instructions (I_1 – I_3) progressing through the pipeline. The rows in the table represent the pipeline stages and the columns the progression of time in machine cycles. For the case of time-stationary coding, the instruction describes all the operations which are to be executed in the pipeline stages in a *single machine cycle*. The programming style is therefore *time-oriented*, or in other words, the programmer has complete control over the operation of the pipeline in each machine cycle. Time-stationary encoding has two major advantages. First, the timing of a program is clear. When interlock is used it is difficult to determine how many cycles an instruction will take to execute. Second, handling interrupts becomes more efficient. Since the programmer has complete control of the pipeline there is no need to flush this prior to an interrupt. Time-stationary encoding is largely used in many DSP architectures [36, 37]. In data-stationary encoding the instruction specifies the operations to be executed on a *particular data item*. This makes coding more natural to the programmer, but it has the drawback that the result of an instruction may not be immediately available to subsequent instructions [33]. Data-stationary encoding has been used in the AT&T DSP32/32C processors.

Branches

One major difficulty in using pipelined architectures is the overhead they can introduce when a branch instruction is executed [9]. This is due to the fact that the condition associated to the branch is computed, in the best case, at the end of the

ID stage. At that point, the following instruction (i.e. the instruction at $PC^3 + 1$) has already been fetched. If the branch condition is *false* then the fetched instruction (which is in the *IF* stage of the pipeline), has to be discharged, implying moving it through the pipeline without performing any useful computation [9]. If an interlock is used to hide the pipeline from the programmer [38, 40], then the conditional branch instruction will have an associated overhead. Branch instructions in DSPs use *control code bits*, which are automatically set by previous ALU operations. This permits the branch condition to be detected early in the pipeline, an approach which combined with short pipelines leads to small interlock penalties (typically 1 cycle) caused by branch mispredictions.

Subroutine Call and Interrupts

The call mechanism available in DSPs is very similar to those found in a GPP. In general, the *PC* is pushed into a *hardware stack*, and restored back when the return instruction from the subroutine is executed.

Interrupts are usually handled similar to subroutine calls. The *PC* is also pushed onto the stack, and the programmer is responsible for the context switch [38, 36]. Some devices [40, 37] have a very low overhead context switching mechanism for interrupts, in which a fresh set of registers (*shadow registers*) is provided for the interrupt routine.

Zero-Overhead Loops

Loop statements are important program structures in the DSP application domain. These algorithms are largely based on the processing of long sequences of array elements which are usually digitized forms of speech and video streams.

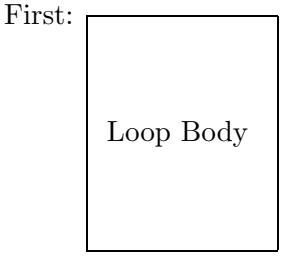
[3] *Program Counter*

In order to minimize the cost associated with the iteration over the array elements, DSP architectures have a *zero-overhead loop* (ZOL) unit. This is a hardware module (Figure 2.5) which contains: (1) a register (COUNT) where the loop induction variable is stored; (2) a register (BEGIN) which stores the address of the first instruction in the loop; and (3) a register (END) which stores the address of the last instruction in the loop (*Last*). In this setup, the ISA also provides a REPEAT instruction (Figure 2.5(b)) which takes as operand *Last* and the number of times the loop body is to be executed (n).

```
ZOL()
begin
   BEGIN ← First;
   END ← Last;
   COUNT ← n;
   if (PC == Last and
         COUNT < N) then
      PC ← First;
   else
      PC ← PC + 1;
   endif;
end;
```

(a)

REPEAT *Last*, n



(b)

Figure 2.5: (a) Zero Overhead Loop algorithm; (b) REPEAT loop construct.

After REPEAT is decoded, *Last* is loaded into END, the address of the first instruction of the loop (i.e. $First$), is loaded into BEGIN, and n stored into COUNT. At each iteration the ZOL unit compares the content of the PC with the address in END. A match occurs when the last instruction in the loop body is identified. The content of COUNT is then decremented. If this result in null, PC is loaded with the next instruction following the loop body (i.e. $Last + 1$), otherwise the content of BEGIN is transferred into PC and the loop starts a new iteration.

The majority of DSP architectures have ZOL capabilities [38, 39, 36, 37]. Some

DSP architectures provide ZOL units which can operate on large loop bodies (e.g. TMS320C50/C540), while in others they only work with single instruction loop bodies (e.g. TMS320C25). In this case the REPEAT instruction has a single operand, i.e. the number of times the following instruction is to be repeated.

Residual Control

DSP architectures usually have arithmetic instructions which can be *residually controlled* [38]. These are instructions for which the execution behavior depends on specific bit-values stored in a *control register*, which were previously set by another instruction. A typical example of that are arithmetic instructions for which the final result will depend on the value of the *sign-extension* bit set by an *operation mode instruction* (e.g. the SSMX sign-extend instruction in the TMS320C25). Depending on this bit being active, the final result of the current instruction may be sign-extended or not. Another application of residual control are *conditionally executed* instructions. As before the execution of the instruction will depend on the value of a control register bit (e.g. ADSP 2100).

2.1.3 ISA

The design specialization found in DSP architectures reflects into the ISA of the processor. The ISA of a typical DSP has very specialized instructions which take operands and store the resulting computation into well defined registers. As a result, limited freedom is available for the code-generation algorithms. The requirements for high-performance also constrain the use of dedicated addressing modes and instruction formats. Instructions are usually encoded such that almost all instructions are fetched in a single machine cycle.

Instruction Encoding

As mentioned before, an important requirement in the design of a DSP is the single-cycle instruction execution time. Under this constraint, it is extremely desirable to minimize the number of bits used to encode a DSP instruction, such as to fit the entire instruction into a single memory word. The goal is to avoid using more than one *read* operation to fetch a complete instruction from the program memory. In order to deal with increasingly growing address spaces used by the programs[4] some DSP architectures use *pages* (e.g. TMS320C1x/C2x/C5x) and PC-relative addressing. In a *paged memory* system (Figure 2.6) the programmer controls the current page, by means of a *Data-Page Pointer* (DP). In this case the instruction operand specifies the *offset* within the current page. The final address of the data is computed in hardware by appending the *offset* bits to the DP bits.

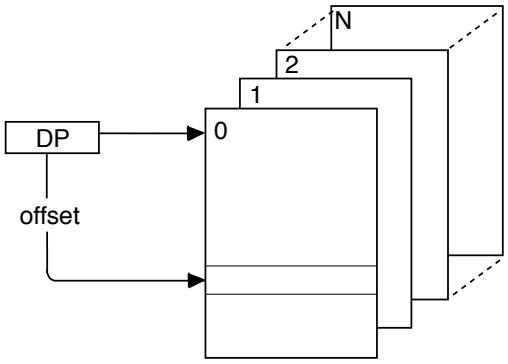


Figure 2.6: Address computation using paged memory.

Addressing Mode

In typical applications, DSPs are used to process sequential streams of data. Since performance is a major requirement, specialized functional units known as Address

[4]The address space for general purpose applications grows ≈ 1.6/yr[9]. It is reasonable to believe that, due to their reduced size, DSP applications will also expand in a similar, but smaller, rate.

Generation Units (AGUs) are employed to provide fast address computation in parallel to the datapath computation.

Due to the encoding constraints mentioned before, *absolute addressing* mode does not play an important role in the design of DSP instructions. Actually, it is very common to find great emphasis on the use of *indirect addressing* instructions in the description of DSP ISAs. In indirect addressing instructions, the register specified in the instruction does not contain the required data, but its address. The instruction operand is obtained by *indirectly* accessing that register. There are two main reasons why indirect addressing is favored in DSP architectures. First, as discussed before, it reduces the number of bits used to encode the instruction. Second, indirect addressing gives the programmer the ability to compute the address of the data used in the next instruction in parallel to the execution of the current instruction, if the architecture permits so. This feature and its impact on code generation, will be studied in great detail in Chapter 4.

The AGU of a standard DSP enables the existence of three basic instruction addressing modes: *linear*, *modular* and *reverse arithmetic addressing*. In the *linear addressing* mode, the address of the data is computed by adding (subtracting) an offset to (from) an address register AR. Since sequential access is very common in the DSP domain, all DSPs have linear addressing modes based on unitary auto-increment (decrement). Others permit offsets larger than one, like the Motorola 56000 and the ADSP 2100. In *modular addressing* the final address is determined by applying function $mod(x) = x$ - $mod\ N$ on the result of $offset \pm AR$. This is very useful when the addressed data structure is a circular buffer[5]. *Reverse arithmetic addressing* is an specialized form of addressing in which the carry bits of the arithmetic operation (e.g. offset increment/decrement) are propagated in the reverse order (i.e. from the most significant to the lowest significant bit). This addressing mode is used to determine

[5]Circular buffers are very common in the DSP application domain.

the address of the data resulting from the *Fast Fourier Transform* (FFT) computation of a signal . Exploiting bit-reverse addressing is a very hard problem in compiling for DSPs, which has not been solved yet.

The AGU is usually formed by a set of register files and a simple ALU capable of performing basic arithmetic operations. Three register files are present in a typical AGU: the Address Register File (ARF), the Offset Register File (ORF), and the Modulo Register File (MRF). Let AR_i, OR_i and MR_i be registers in ARF, ORF and MRF respectively. In order to allow for a simple design, these registers usually have a one-to-one correspondence with each other. For example AR_2 can only be used with OR_2 or MR_2. The function of register AR_i is to store the final result of the address computation and to directly address the data in memory. Register OR_i holds the offset used to increment (decrement) AR_i and register MR_i stores the size of the circular buffer during modular addressing.

Figure 2.7 shows the possible addressing modes available in a typical DSP, and an example of an instruction using the particular mode to load data into the accumulator A. When indirect access using unitary auto-increment (decrement) is the selected mode (Figure 2.7(a)), the data is first fetched from the memory position pointed by AR_1, after which the contents of AR_1 are incremented (decremented). If the data to be accessed next is located in a memory position far from the one currently pointed by AR_1, indirect addressing using offset registers should be employed (Figure 2.7(b)). In this case AR_1 is used to get the current data, after which the contents of register OR_1 are added (subtracted) to (from) AR_1. Finally, if modular addressing is the chosen mode (Figure 2.7(c)), the result of operation $AR_1 \pm OR_1$ is compared with the contents of MR_1. If $AR_1 \pm OR_1$ is smaller than MR_1, then this quantity is stored into AR_1, otherwise AR_1 receives the difference $AR_1 \pm OR_1 - MR_1$, and the circular buffer wraps around. Observe that **all** addressing modes above are *post-modified* modes. In other words, the present contents of AR_1 are used to fetch the

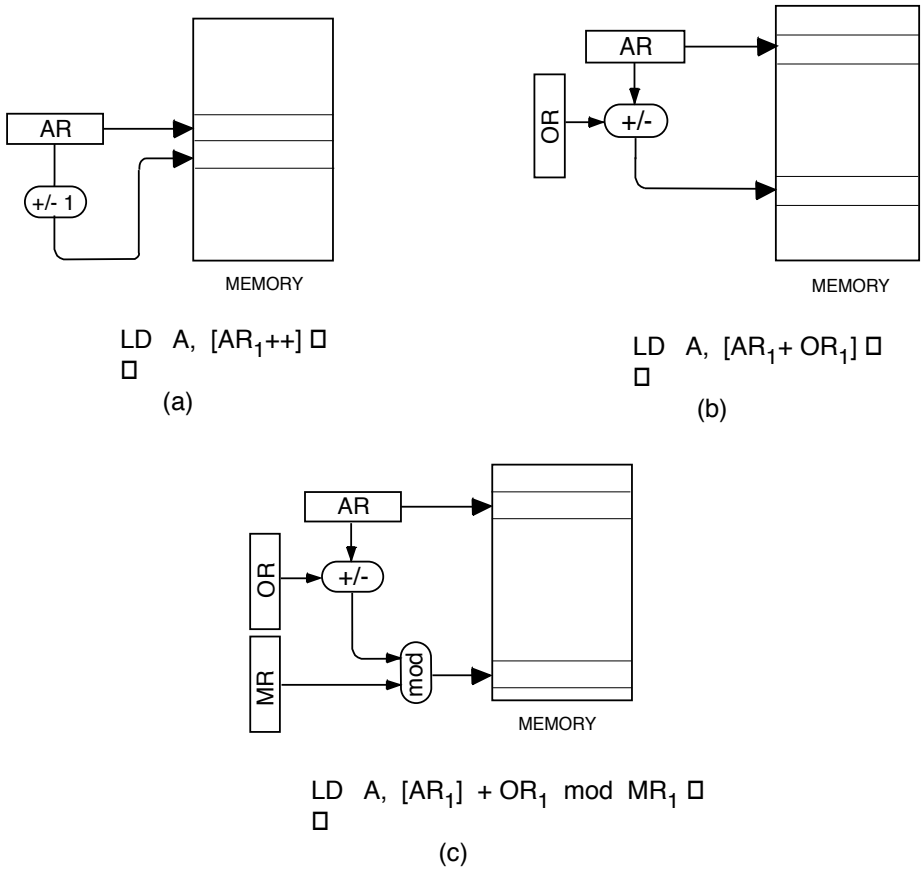


Figure 2.7: (a) Auto-increment (decrement) addressing mode; (b) Linear addressing mode with offset; (b) Modular addressing mode.

operand required by the current instruction, and only after this happens is the new address stored into AR_1. In this case the compiler should use the current instruction to compute the address of the operand required by the one following it. Consider now the situation where the above addressing modes are not post-modified, i.e. the address of the current operand is computed during the current instruction. In this case, the current instruction cannot finish until the operand address is computed, adding one extra cycle to its execution time. This is the case of the *offset indirect* mode in the Motorola 56000 processor, which takes 2 cycles instead of one.

The basic idea behind the design of DSP addressing modes is to guarantee that the

addresses of the operands of the next instruction are computed before this instruction is fetched.

2.2 The RTG Model for ISA

A key aspect of DSP architectures is how instructions in the processor ISA make use of the datapath structures. A careful analysis of the various types of DSP datapaths reveals a large variation of design styles. In this section a structural model for the ISA is proposed, which seeks to address this variety. The *Register Transfer Graph* (RTG) model exposes the storage locations found in the datapath of the processor, and how instructions in the ISA use the *transfer paths* defined between these locations to perform the required computation. Based on this model a classification of the DSP architectures is proposed (Section 2.3). Later this classification is used to derive a number of interesting insights regarding code generation for *heterogeneous register* architectures.

Register Set and Classes

The *register set* of a processor is the set formed by all storage locations (i.e. *registers*) in the processor datapath, which are available to the programmer through the ISA. In order to identify the functionality associated with each register, the concept of *register classes* is required. One can think of a register class as a set of registers which are homogeneous (in the way defined in Section 2.1.2), and have the same storage functionality. Consider, for example, the TMS320C25 architecture of Figure 2.3. The register set of that processor contains only three registers a, p and t. One can think of these registers as being divided into three *register classes* (or register files), one associated with each register. Each class has a distinct function in the datapath. Observe, for example, that register p forms a unique class since this is the only register

which can store the result of the multiplication. Similarly register t is another register class because this is the only register which can store an operand for the multiplier. The other operand always comes from the memory. *Memory* is assumed to be an infinite storage resource which, because of its uniqueness, will not be considered as a register class.

The concept of register classes can also be applied to homogeneous register GPP architectures. Consider, for example, a typical RISC processor with 32 integer registers (R_0 – R_{31}) and 32 floating-point registers (FP_0 – FP_{31}). From the perspective of instructions in the ISA all integer registers, but register *zero* (R_0), are indistinguishable from each other. In other words, all integer registers are created the same and do not play a specific role in any instruction. A similar statement can also be made of the floating-point registers. As a consequence, a typical RISC architecture has three register classes: $\{R_1, \ldots, R_{31}\}$, $\{R_0\}$ and $\{FP_0, \ldots, FP_{31}\}$.

The Instruction Set Architecture

The RTG model requires a description of the target processor ISA. This can be done through a table which captures the operations performed by an instruction, and the storage locations it uses. Table 2.1 shows a partial specification of the instructions in the ISA of the TMS320C25 processor. As mentioned before, the TMS320C25 has three register classes $\{a\}$, $\{p\}$, $\{t\}$ with one register per class. The *opcode* of the instructions are showed in the second column of Table 2.1. The third column lists the name of the register classes which are the operands used by the instructions. Assume here that source and destination registers have direct paths to their functional units, i.e. they do not need to go through other registers while the instruction is executed. This is universally true for all DSP architectures due to obvious efficiency reasons. The fourth column lists the register class used to store the result of the instruction. The fifth column shows the number of clock cycles required by each instruction.

Instr.	Opcode	Sources	Dest.	Cycles	Three-address
1	add m	a,m	a	1	a $\leftarrow$ a + [m]
2	apac	a,p	a	1	a $\leftarrow$ a + p
3	spac	a,p	a	1	a $\leftarrow$ a - p
4	mpy m	t,m	p	1	p $\leftarrow$ [m] $*$ t
5	mpyk k	t,k	p	1	p $\leftarrow$ t $*$ k
6	lack k	k	a	1	a $\leftarrow$ k
7	pac	p	a	1	a $\leftarrow$ p
8	sacl m	a	m	1	a $\leftarrow$ [m]
9	lac m	m	a	1	[m] $\leftarrow$ a
10	lt m	m	t	1	t $\leftarrow$ [m]
11	lact m	m,t	a	1	a $\leftarrow$ [m] $<<$ t
12	spl m	p	m	1	[m] $\leftarrow$ p

Table 2.1: Partial description of the the TMS320C25 processor.

Notice that all of them execute in a single cycle, what is typical of DSP architectures. Finally, the sixth column describes the operation performed by the instruction, using a *three-address* notation. Three-address is a standard representation of processor instructions, where the destination of the instruction, its two operands (hence the name three-address) and the operation it performs are represented. In this notation any reference in square brackets is associated with a memory position.

Notice from the ISA of Table 2.1 that instructions implicitly define the register classes they use. For example, instruction *apac* can only take its operands from registers a and p, and always computes the result back into a. Observe also the presence of instructions, like *lac m* (load register a from memory position m), and *pac* (move register p into register a) whose only purpose is to transfer data through the datapath.

The ISA description of Table 2.1 represents only the instructions which are *atomic*, i.e. which cannot be further subdivided into its parallel components. An instruction is said to be a *complex instruction* if it can be divided into its *atomic* components.

The reasoning behind that comes from the fact that the code generation approach adopted in this thesis assumes that the *Instruction Level Parallelism* (ILP) available through complex instructions is explored only after sequential code generation has been performed. In this case local [42] and global compaction techniques [43] can be used after sequential code has been produced. Consider, for example, the TMS320C25 complex instruction *lta m* (which being complex is not present in Table 2.1). This instruction loads register t and accumulates the previous contents of register p into the accumulator a. These two operations can be performed by compacting atomic instructions *lt* and *apac* from Table 2.1. The code generation approach used in this thesis (Chapter 3) assumes that, once sequential code generation has been performed, compaction techniques can identify and generate such complex instructions. This is true for the majority of the complex instructions in DSP processors.

The Register Transfer Graph

Given a program basic block the task of code generation is to transform the IR associated with the basic block into a sequence of instructions from the target processor. The next step in modeling a DSP architecture is to provide an abstraction which permits algorithms to extract from the ISA those features which are important for the code generation task. The graph model in this section, called RTG, is such an abstraction. It will be used throughout this thesis in the analysis and design of code generation algorithms.

Definition 1 *The RTG is a directed labeled multigraph where each node represents a register class. Each edge in the RTG from node r_i to node r_j is labeled with those instructions in the ISA that take operands from location r_i and store the result into location r_j.*

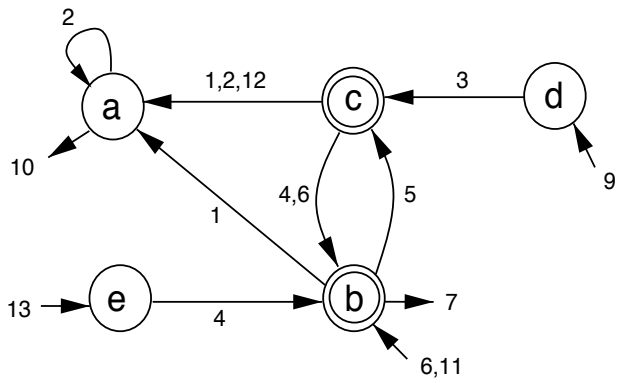


Figure 2.8: Example of an architecture RTG

The nodes in the RTG represent two types of storage: *single-register* and *multiple-register*. Multiple-register nodes are associated with a register class that can store multiple operands. A single-register node (or simply register) is a register class of unitary capacity. Multiple-register nodes are distinguished from single-register nodes by means of a double circle. For example, the RTG of Figure 2.8 has three single-register nodes (a, d and e), and two multiple-register nodes (b and c). The RTG is a labeled graph where each edge has labels corresponding to the instructions that require that particular transfer operation. In the RTG of Figure 2.8, for example, binary instructions 1 and 2 take one operand from c and store the result into a. Hence, the edge from c to a will have at least two labels: 1 and 2. The other operand of 1 comes from b, while the second operand of 2 comes from node a. Therefore, edges (b, a) and (a, a) will be annotated respectively with instructions 1 and 2. The presence of a self-loop (edge (a, a)) on node a indicates that for instruction 2 register a behaves as an accumulator.

Memory is assumed to be an infinitely large storage resource, and is *not* represented in the RTG for simplicity. When a transfer path exist between an RTG node and the memory, an arrowhead is used to indicate that, with the direction of the edge showing the direction of the transfer operation. An incoming (outgoing) arrow

is associated with a *load* (*store*) operation. In Figure 2.8 node b can load (store) data from (to) memory, using instruction 11 (7). Nodes d and e can only perform load operations, by means of instructions 9 and 13, while register a can only store data into memory using instruction 10. Observe that node c is an internal datapath node which has no access to memory. Arrows from memory can also be used to describe memory-register instructions. For example, instruction 6 is present in edge (c, b) and in the incoming arrow to b. Therefore 6 is a binary instruction which takes one operand from c and the other from memory.

The number of operands of the instructions in the processor ISA can be identified by matching the labels from the incoming edges of a particular RTG node. For example, consider the labels of all incoming edges into node b. Since only incoming edges (c, b) and (e, b) are annotated with 4, it is clear that 4 is a binary instruction which takes operands from register c and e and stores the result into b. Some instructions, like 3, are unary and will show up in only one edge, i.e. (d, c).

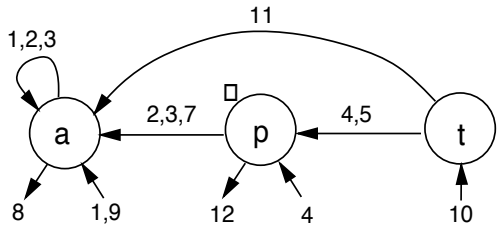


Figure 2.9: The RTG of the TMS320C25 processor.

Example 1 Consider, for example, the partial ISA description in Table 2.1. The RTG of Figure 2.9 was formed from that description. The instruction numbers were used to label each edge of the graph. Remember that Table 2.1 is a partial description of the TMS320C25, and therefore not all instructions of the target processor are represented there. Nevertheless the instructions in Table 2.1 were selected such that the RTG of Figure 2.9 is complete. The actual architecture of the TMS320C25

processor is shown in Figure 2.3. Notice that only registers a and p are used to store the result of any instruction operation. Register a is an accumulator (observe the self-loop in the RTG) which is used to store the result of the ALU operations, while p stores the result of the multiplication. Instructions 9 and 10 (8 and 12) are *load* (*store*) instructions from (into) a and p. Observe that instructions 1 and 4 are memory-register, and 2 and 3 are register-register (accumulator based) instructions.

2.3 RTG Based Datapath Classification

The goal of this section is to identify basic datapath computation structures which are present across many different DSP designs. The emphasis is not on classifying different types of functional units or register sets found in these processors, but on identifying how these elements interact, and how the compiler can make efficient usage of this interaction in order to generate high-quality code.

The classification is based on the RTG model described in the previous section, with the arrows for memory access being simplified, such that the incoming and outgoing arrows of a node are merged to form double arrowheads. The classification is directed towards the analysis of sequential code generation. In this sense it is not complete, given that it does not capture many ILP features available in all existing DSP processors. Assume that ILP is handled after sequential code generation, by means of compaction techniques. The classification goal is to reveal datapath properties which directly impact the code generation task. It seeks to identify basic datapath structures which are available in all commercial DSP processors. With this as the basis, datapaths are divided in two broad categories, according to the usage they make of memory operands. They are: *memory-register* and *register-register* datapaths.

2.3.1 Memory-Register Datapaths

This was the datapath used in the first computers. It is formed by a single register (a) which can load (store) data from (into) memory. The binary instructions in the ISA take two operands. One operand is stored in register a and the other operand in memory. This architecture style is very adequate for computations in which the next operation immediately uses the result of the previous one, like in the summation (or accumulation) of a sequence of numbers or partial results. This operation is as common in DSP applications, as it is in many numerical programs like those which ran on the first computers[6]. Due to this reason register a is known as *accumulator*. Observe that if the result of any operation is always stored in the accumulator, binary instructions (e.g. Figure 2.10(c)), need only to specify the address of one operand, since the other operand is implicit in the accumulator. This datapath is shown in Figure 2.10(a), and its corresponding RTG in Figure 2.10(b).

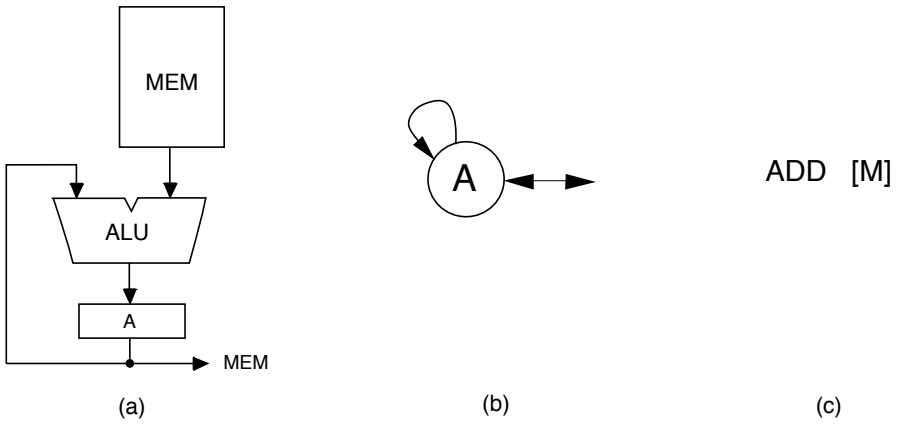


Figure 2.10: (a) Memory-register architecture; (b) Corresponding RTG; (c) Typical instruction.

The fact that one operand is stored in memory is a major drawback in the performance of this datapath. Here, the latency of any instruction is the sum of the

[6]The first computers were primarily used to compute the trajectory of projectiles and other numerical military applications.

memory access time plus the latency of the functional unit. This has a significant impact in the final cycle time of the processor, given that memory latency has become increasingly large. The impact is reduced because memory in DSPs are static on-chip RAMs, which have faster access times when compared with their dynamic cousins. The main advantage of such a datapath is that it can potentially reduce the number of load/store operations required to perform a particular computation. Unfortunately this advantage is not enough to compensate for the large and increasing gap between memory and register access times. Code generation for expression trees on this architecture is trivial, since at each operation one can first emit code for the operand located in memory [44].

Memory-register DSP architectures

Despite of its drawbacks, a large number of DSP processors use this datapath design [38, 40, 45]. They use *paged-memory* techniques (see Section 2.1.1) to diminish the size of the memory block accessed each time, thus minimizing the memory access time. Paged-memory systems have the disadvantage that they require the programmer to manage the *Data-Page* (DP) pointer. This is not a major problem though, given that *data-flow* analysis techniques can be used to minimize the cost associated with this task [29].

An example of such a DSP is the TMS320C25 shown in Figure 2.3. This is a memory-register architecture because a number of instructions (e.g. 1 and 4 in Table 2.1) have one operand in memory. Observe that its datapath can implement a *mac* instruction: an addition occurs in the ALU at the same time that a multiplication is performed in the MUL. As it was mentioned before, this is an important operation for DSP applications. The RTG associated with the TMS320C25 datapath was described before in Figure 2.9.

2.3.2 Register-Register Datapaths

This datapath design, shown in Figure 2.11(a), is the response of designers to the increasing gap between memory and processor latencies (also known as *Memory Wall* [46]). The reasoning behind this is as follows. Based on the fact that memory access is very expensive, in cycle terms, it is better to increase the number of (fast) registers and to design instructions which only use registers as operands. The drawback in this approach is that it requires compilers to be able to keep the required operands in registers, thus avoiding expensive memory accesses.

Instructions from architectures containing such datapaths have to specify two registers as operands and one register as the destination of the operation Figure 2.11(c). This datapath style has been largely used in the design of modern processors. Since modern compilers can efficiently keep operands in registers using a number of register allocation techniques [47, 48], it became the de facto standard in the design of today's RISC processors [9].

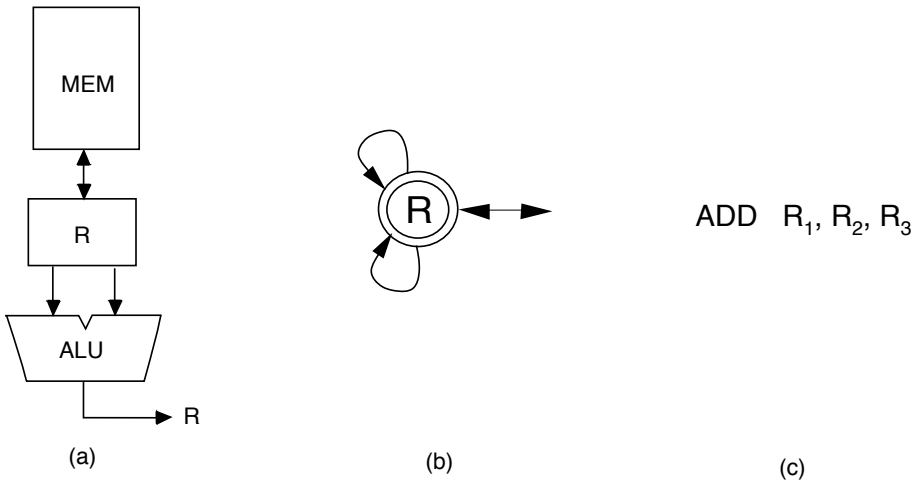


Figure 2.11: (a) Register-register architecture; (b) Corresponding RTG representation; (c) Typical instruction.

Register-register DSP architectures

The high performance requirements of DSP applications lead designers to make extensive use of parallelism in the design of register-register DSP datapaths. A very common technique, employed in a number of processors [36, 37, 39], is to fetch the operands of the next instruction during the execution of the current one. The idea here is to hide the latency of the memory access by transferring the data from/to memory during the time when an instruction performs a computation. The immediate consequence of that is an increase in the memory bandwidth requirement. In order to meet this requirement DSPs use memory banks, which can be accessed in parallel (see Section 2.1.1 for details). Although multi-ported memories[7] are another way to achieve the required bandwidth, they are usually not considered a design option because of their large silicon area constraints. When memory banks are used, a dual-bank organization is preferred, given that the majority of the instructions in a DSP application are binary instructions. Memory banks have a disadvantage though. They require the compiler to be able to perform efficient bank allocation of program variables. This originates from the need to minimize the cost of moving variables to another bank, resulting from datapath constraints. Consider, for example, the dual-bank architecture from Figure 2.12. It contains memory banks MX and MY, and registers RX and RY, which load/store data from these banks. Due to datapath constraints parallel cross-load instructions (e.g. $RX \leftarrow [MY]$ and $RY \leftarrow [MX]$) are not permitted during the same machine cycle. Actually this is the case of the Motorola 56000 processor. Therefore, the compiler has to minimize conflicts resulting from variables being required in specific memory banks, such as those those which occur due to non-commutative instructions.

Because this datapath style allows for two simultaneous load operations to occur

[7] A multi-ported memory is a memory which allows more than one *load/store* operation to occur at the same time.

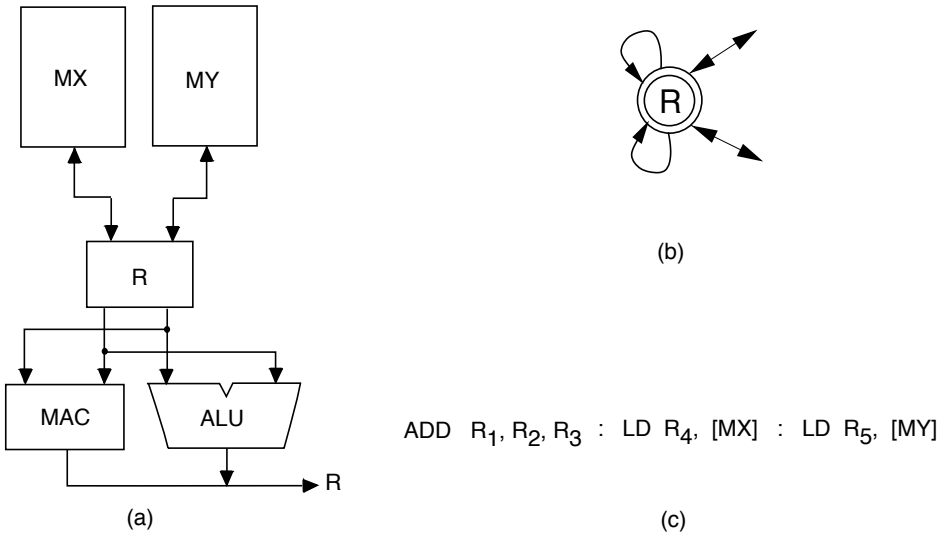


Figure 2.12: (a) Dual-Load Execute architecture with muti-ported register file; (b) The corresponding RTG; (c) Typical binary instruction.

during the execution of the instruction, it is referred to as a on *Dual-Load-Execute* (DLE) architecture. A number of modern DSP designs are based on DLE architectures. The different variations on this theme are listed in the following sections.

Dual-Load-Execute I

In this case the datapath has one multi-ported register file (R) and a memory system based on two memory banks, MX and MY (Figure 2.12(a)). This combination allows for five accesses of registers in R to occur simultaneously. A typical binary instruction executed in this architecture is shown in Figure 2.12(c). The instruction reads two registers R_1 and R_2, and stores the resulting computation into R_3. Observe that at the same time, registers R_4 and R_5 load the operands for the next instruction from banks MX and MY. This architecture has a dedicated unit (MAC) which allows a `mac` instruction to be executed in a single machine cycle. The same unit is also used to perform multiplication.

Although DLE-I architectures considerably improve the performance, in terms of

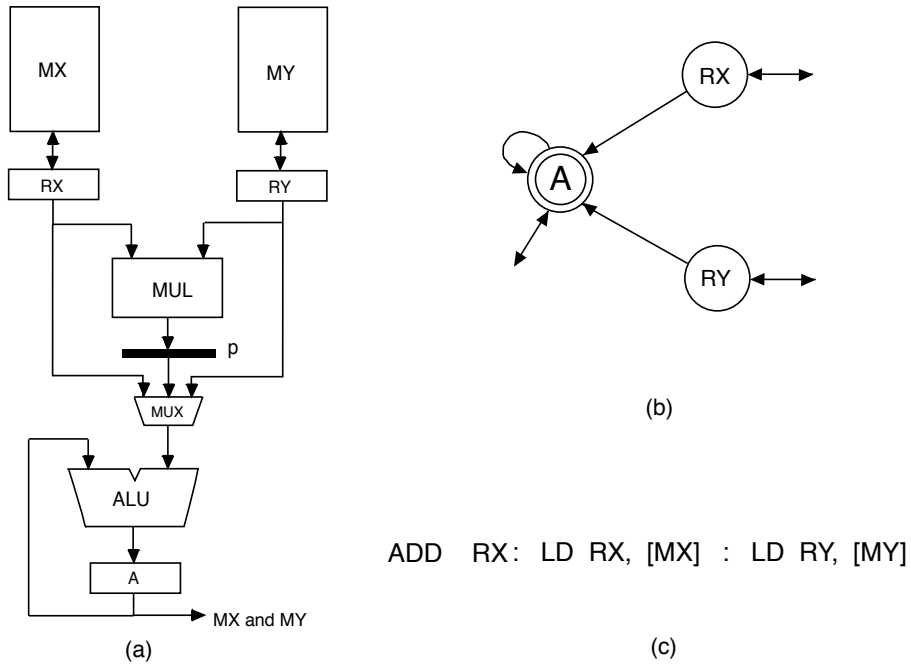


Figure 2.13: (a) Dual-Load-Execute architecture with separate *load* registers; (b) The corresponding RTG; (c) Typical instruction.

the number of cycles, they have a major drawback. Multi-ported register files are very expensive in terms of silicon area and design effort, both of which impact the final cost of the device. On the other hand, this datapath enables easy code generation, which can become a big advantage if the size of devices continues to decrease at the current rate. The NEC [39] processor uses a DLE-I datapath architectural style.

Dual-Load-Execute II

The DLE-II architecture (Figure 2.13) is another variation of the previous design. It seeks to eliminate the costly multi-ported register file, while maintaining the Dual-Load-Execute property. In this case two registers (RX and RY) are used to perform the dual-load operation. The result of any instruction is always stored in the accumulator register file A. A typical instruction in this architecture (Figure 2.13(c)) performs its operation using a register from A and the current contents of registers

RX and/or RY. Meanwhile registers RX and RY can be loaded with the operands for the next instruction. Since registers RX and RY are separated from A, no multi-port register file is required, which results in a significant reduction of the total silicon area used by the processor.

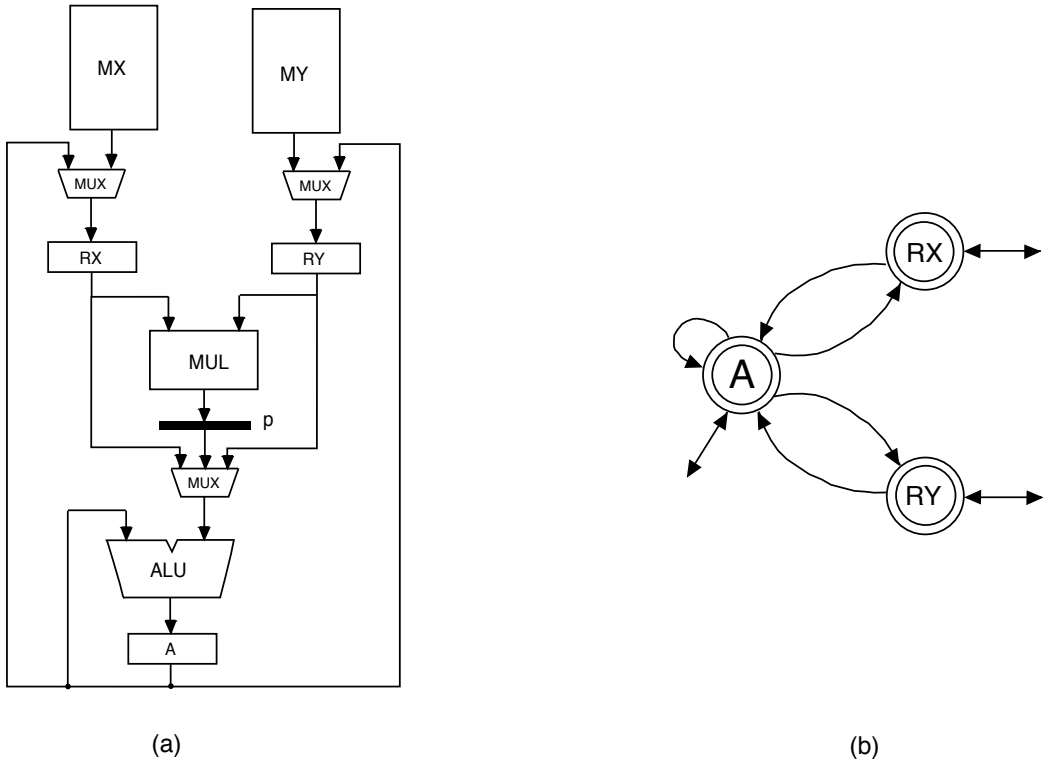


Figure 2.14: (a) Motorola 56000 DSP architecture; (b) Corresponding RTG.

Another version of this design can be obtained by making RX and RY to be register files instead of single registers. This is the approach used in the design of the ADSP-2100 DSP [37] and the Motorola 56000 [36]. Observe that the architecture of Figure 2.13 has a major drawback. This can be observed when the result of an ALU operation needs to be used as the operand of a multiplication. Consider for example the expression $a * (c - d)$. Assume a is loaded into RX, and $c - d$ is stored into accumulator A_0 from file A. Since there is no direct path from register file A to RY, then the result in A_0 will take 2 cycles to reach register RY, where it is needed for the

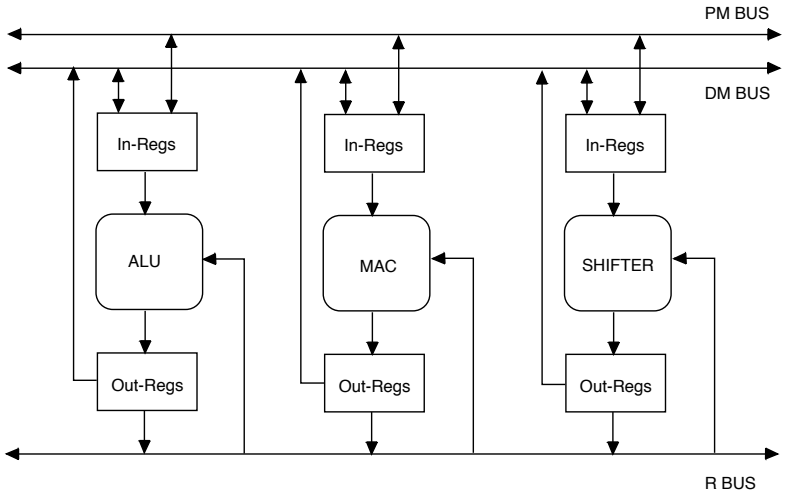


Figure 2.15: ADSP 2100 architecture.

multiplication with RX. One cycle is required to take A_0 into memory and another one to bring it into RY. If a transfer path existed from A to RY then this overhead could be reduced to a single cycle. This is the approach used by the Motorola 56000 DSP [36] as shown in Figure 2.14.

Some DSPs have very complicated datapath topologies, with many functional units. Nevertheless, they frequently use the structure in Figure 2.14 as a building block in the design of these units. The ADSP 2100 processor is a typical example of such processor. It has three functional units: MAC, ALU and SHIFTER, where MAC and ALU are based on the style of Figure 2.14. Its architecture is shown in Figure 2.15. For the sake of simplicity not all registers and their interconnections are represented. Registers can be divided into *input registers* and *output registers*. *Input registers* play a similar role as registers RX and RY in Figure 2.14, while *output registers* can be used as accumulators A. The input registers read data from the *Data-Memory Bus* (DM Bus) and from the *Program-Memory Bus* (PM Bus). The output registers use the *Result Bus* (R Bus) to transfer operands to other functional units.

2.3.3 RTG Based Architectural Classification

The above described datapath classification covers an extensive number of processor architectures. These architectures can also be classified in a different way, based on the structure of the RTG associated to the architecture datapath. An architecture is said to be *acyclic* if its RTG has no cycles, and *cyclic* otherwise. Similarly, a processor has a cyclic (acyclic) ISA if its RTG has (no) cycles. An *RTG cycle* is defined to contain *at least* two distinct RTG nodes. Therefore a self-loop, like the one found in node a of Figure 2.9, is not considered a RTG cycle.

The largest share of the DSP market today (circa 1996) is dominated by an acyclic architecture (e.g. TMS320C25 and TMS320C50). Other acyclic DSP architectures can also be found in the Fujitsu FDSP-4 and in the NEC uPD77016 processors. Memory-register architectures, like the one in Figure 2.10, are typically acyclic architectures. The same can be said of register-register architectures (e.g. RISC), as the one from Figure 2.11. For reasons related to cost and increasing memory latency, the majority of the new DSP architectures being designed today, such as the Motorola 56000 and the ADSP 2100, are cyclic architectures.

The importance of this classification will become clear in the next chapter. Chapter 3 demonstrates that acyclic ISAs always have optimal linear time code generation algorithms for expression trees. The ideas in Chapter 3 can also be extended, although in a limited form, to cyclic ISAs. In general, the goal of that chapter is to give new insights into the relation between code generation and datapath topology, clarifying this relation and showing how to use it to design new code generation algorithms.

Chapter 3

Code Generation for Acyclic Instruction Set Architectures

3.1 Introduction

Chapter 2 studied the main architectural features of DSPs and proposed an architectural model (RTG), which can capture the information available in the ISA of these processors. In that chapter, the RTG properties of some known DSPs were studied, and processors were classified according to the existence of cycles in the RTG. The reason for this classification and its role in code generation will be clarified in this chapter.

As mentioned before, an architecture has a *cyclic* ISA if its RTG has cycles, and an *acyclic* ISA otherwise. While several DSP architectures being designed today have cyclic ISAs, acyclic ISAs are still present in a large fraction of the DSPs used today. The problem of basic block code generation, for programs executing on these architectures, will be addressed in this chapter. Moreover, the algorithms developed here can also be applied, in a restricted way, to architectures having cyclic ISAs, as will be shown later. Also, the insights gained by modeling code generation using the RTG opens up new avenues for the design of algorithms for these and other

application specific architectures.

3.2 Overview

The problem of code generation for basic blocks can be divided into three tasks: *instruction selection*, *register allocation* and *scheduling*. Instruction selection is the task of selecting instructions to match the operations described in the *Intermediate Representation* (IR) of the basic block. Register allocation has to do with identifying the registers which will be used by the selected instructions. Scheduling is the task of ordering the execution of the instructions.

The order in which these tasks are performed is important (i.e. referred to as *phase ordering*) and different orders usually result in code with different quality. Moreover, depending on the architecture at hand, some tasks have to be performed together. For example, in General-Purpose Processors (GPPs) instruction selection can be performed before combined register allocation and scheduling (e.g as in the Sethi-Ullman algorithm), while for some DSPs instruction selection and register allocation must be performed together.

Code generation is, in general, a hard problem. Fortunately, there is a large body of work done in code generation for GPPs. Instruction selection for basic block expressions subsumes Directed Acyclic Graph (DAG) covering, which is an NP-complete problem [49]. Sethi *et al.* showed that the problem of optimal code generation for DAGs is NP-complete even for a single register machine [50, 51]. It remains NP-complete for expressions in which no shared term is a subexpression of any other shared term [52]. An efficient solution for a restricted class of DAGs has been proposed in [53]. Code generation for expression trees has a number of $O(n)$ algorithms, where n is the number of nodes in the tree. These algorithms offer solutions for the cases of stack machines [54], register machines [44, 55] and machines with specialized

instructions [56, 57]. They form the basis for code generation for single issue, in order execution, general-purpose architectures. Recent studies in code generation for RISC architectures, which consider the problems of register allocation and scheduling, can be found in [58] and [59].

The problem of generating code for DSPs and embedded processors has not received much attention though. This was probably due to the small size of the programs running on these architectures, which enabled assembly programming. With the increasing complexity of embedded systems, programming such systems without the support of high-level languages has become impractical. Many of the problems associated with code generation for DSPs were first brought to light by Lee in [19, 33], a comprehensive analysis of the architecture features of these processors. The problem has its roots in the 70's, when the basic principles for code generation for accumulator-based, stack-based, memory-register, and register-register architectures were studied [51]. Coffman and Sethi in their comprehensive paper [60] addressed some of the basic issues related to that. More recently, a number of researchers have tackled some of the basic problems in compiling for these architectures. Marwedel *et al.* [61] proposed a tree-based mapping technique for compiling algorithms into microcode architectures. Paulin *et al.* [62] uses a tree-based approach for algorithm matching and instruction selection, where registers are organized in classes and register allocation is based on a left-first algorithm. Datapath routing techniques have also been proposed [63] to perform efficient register allocation. Wess [64] proposed a combined approach for register allocation and instruction selection using the concept of *trellis diagrams* [65]. An overview of the current research work on code generation for DSP processors, and embedded processors in general, can be found in [66].

As discussed in Chapter 2, the TI TMS320C25 Digital Signal Processor [67] will be considered the target architecture for this chapter. The reason is that this processor is part of the TI TMS320 family of DSPs, which makes a large fraction of all

commercial DSP processors in use today. The TMS320 family is composed of fixed-point (TMS320C1x/C2x/C5x/C54x) and floating-point processors (TMS320C3x/4x). As discussed in Chapter 2, the TMS320C25 is an acyclic ISA containing specialized memory-register and register-register instructions. It has three separate register-files (a, p and t) containing a single register each. The TMS320C25 has the largest share among all fixed-point DSP processors in use today (circa 1997).

The code generation approach used in this thesis divides the problem into two phases. First, sequential code generation is performed, followed by local code compaction. DSP architectures usually have *Instruction Level Parallelism* (ILP) features which have to be explored in order to achieve high quality code. The work developed here addresses only the problem of sequential code generation. The approach used for the problem is based on two tasks. First, (Section 3.3) instruction selection and register allocation are performed together, followed by scheduling (Section 3.4).

3.3 Instruction Selection and Register Allocation

In homogeneous register architectures the selection of an instruction has no connection whatsoever with the types of registers that the instruction uses. Selecting instructions for heterogeneous register architectures, like DSPs, usually requires allocating a register from specific register-files as operands for particular instructions.

As a consequence, the IR patterns associated with instructions in this kind of architecture must carry information regarding the type of register the instruction uses. The strong binding between instruction selection and register allocation indicates that these tasks must be performed together [64].

Consider, for example, the IR patterns in Figure 3.1 corresponding to a subset of the instructions in the TMS320C25 ISA, as described in Table 2.1. In Figure 3.1 each instruction is associated with a tree-pattern whose nodes are composed of operations

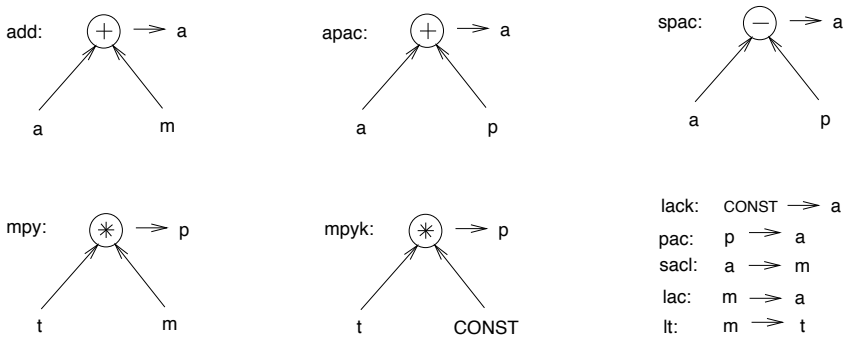


Figure 3.1: IR patterns for the TMS320C25 processor.

Instruction	Cost	Operands	Destination	Linearized Notation
add m	1	a,m	a	a ← PLUS(a,m)
apac	1	a,p	a	a ← PLUS(a,p)
spac	1	a,p	a	a ← MINUS(a,p)
mpy m	1	t,m	p	p ← MUL(t,m)
mpyk c	1	t,c	p	p ← MUL(t,CONST)
lack c	1	c	a	a ← CONST
pac	1	p	a	a ← p
sacl m	1	a	m	m ← a
lac m	1	m	a	a ← m
lt m	1	m	t	t ← m

Table 3.1: Partial ISA of the TMS320C25 processor.

(PLUS,MINUS,MUL), registers (a,p,t), constants (CONST) and memory references (m).

These tree-patterns, can also be described in a prefixed linearized functional notation, similar to three-address form, as shown in Table 3.1. Notice that each instruction implicitly defines the register class it uses for operands and results. For example, the instruction *apac* can only take its operands from registers a and p, and always computes the result back into a. Observe also the presence of operations which transfer data through the datapath like *lac* m (load register a from memory position m), and *pac* (move register p into register a). The associated cost in this case is only the

cost of moving the data from the source register into the destination register. Since registers in DSP architectures are a scarce resource, the final code quality is very sensitive to the cost of routing data through the datapath, as observed in [63].

3.3.1 Problem Definition

Optimal instruction selection combined with register allocation is the problem of determining the best cover of an expression tree such that the cost of each pattern match depends not only on the number of cycles of the associated instruction, but also on the number of cycles required to move its operands from the location they currently are to the location where the instruction requires them to be.

3.3.2 Problem Solution

A solution for this problem is to use a variation of the Aho-Johnson algorithm [55] such that each node keeps not only all possible costs for matches at that node, but also all possible costs resulting from matching the node and moving the result from where it is originally computed into any other reachable location in the datapath [65].

Tree-grammar parsers have been used as a way to implement code-generators [68, 69, 70]. They combine dynamic programming and efficient tree-pattern matching algorithms [71] for optimal instruction selection. Twig [68] and IBURG [69] are versatile code-generator generators. In IBURG, pattern matching is based on an open-coding style in which the patterns are identified directly via `IF` and `SWITCH` statements. OLIVE [70] is a code-generator generator aimed to add a powerful specification language, such as the one found in Twig, on the top of IBURG.

OLIVE was used to implement the combined instruction selection and register allocation algorithm described here. It takes as input a set of grammar rules where tree-patterns are described in a prefixed linearized form, as the one shown in Table 3.1.

The IR patterns from Table 3.1 are converted into the partial OLIVE description of Figure 3.2. Notice that non-terminals are represented by lower case letters and terminals by capital letters.

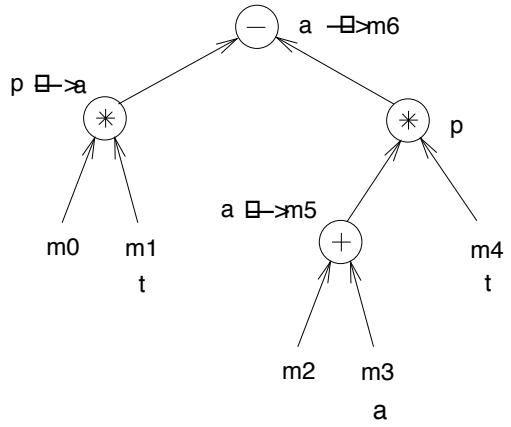
```
a : PLUS(a,m)    {} = {};     (1) add  m
a : PLUS(a,p)    {} = {};     (2) apac
a : MINUS(a,p)   {} = {};     (3) spac
p : MUL(m,t)     {} = {};     (4) mpy  m
p : MUL(t,CONST) {} = {};     (5) mpyk k
a : CONST        {} = {};     (6) lack k
a : p            {} = {};     (7) pac
m : a            {} = {};     (8) sacl m
a : m            {} = {};     (9) lac  m
t : m            {} = {};    (10) lt   m
```

Figure 3.2: Partial OLIVE specification for the TMS320C25 processor (instruction numbers and names on right are not part of the specification).

Rules 1 to 3 and 4 to 5 correspond to instructions that take two operands and store the final result in a particular register (a and p respectively). Rule 6 describes an immediate load into register a. Rules 7 to 10 are associated with data transfer instructions and bring the cost of moving data through the datapath into the total cost of a match. For the sake of simplicity, Figure 3.2 does not list all patterns corresponding to each commutative operation. For example, instruction *add m* can be specified in two different ways: PLUS(a,m) and PLUS(m,a). Nevertheless, consider for the remainder of this chapter that all commutative forms of any operation pattern are available whenever required.

If instruction scheduling does not add any resource conflicts, then the algorithm above results in optimal code. This follows from the fact that this algorithm is a variation of the provable optimal Aho-Johnson dynamic programming algorithm [55].

Example 2 Consider, for example, the expression tree of Figure 3.3. Once optimal



(a)

lt m1	t ← [m1]
mpy m0	p ← t * [m0]
pac	a ← p
sacl m7	[m7] ← a
lac m3	a ← [m3]
add m2	a ← a + [m2]
sacl m5	[m5] ← a
lt m4	t ← [m4]
mpy m5	p ← t * [m5]
lac m7	a ← [m7]
spac	a ← a - p
sacl m6	[m6] ← a

(b)

lt m4	t ← [m4]
lac m3	a ← [m3]
add m2	a ← a + [m2]
sacl m5	[m5] ← a
mpy m5	p ← t * [m5]
lt m1	t ← [m1]
pac	a ← p
sacl m7	[m7] ← a
mpy m0	p ← t * [m0]
pac	a ← p
lt m7	t← [m7]
mpyk 1	p ← t * 1
spac	a ← a - p
sacl m6	[m6] ← a

(c)

lac m3	a ← [m3]
add m2	a ← a + [m2]
sacl m5	[m5] ← a
lt m1	t ← [m1]
mpy m0	p ← t * [m0]
pac	a ← p
lt m4	t ← [m4]
mpy m5	p ← t * [m5]
spac	a ← a - p
sacl m6	[m6] ← a

(d)

Figure 3.3: (a) Matched IR tree for the TMS320C25; (b) SNF Left-first schedule; (c) SNF Right-first schedule; (d) Optimal schedule.

instruction selection is performed, register classes are allocated to operations. Observe that the accumulator a matches the PLUS operation and afterwards the result is moved into memory position m_5. This is not considered a memory spill since it reflects an architecture feature of the processor and not a resource conflict. The heterogeneous register characteristic of DSPs can be observed in Figure 3.3 by the need to move data from p to a after the MUL operation on the left subtree is executed.

3.4 Scheduling

Optimal instruction selection and register allocation for an expression tree is not enough to produce optimal code. For optimal code the instructions must be scheduled in such a way that no memory spills are introduced. Notice that memory positions allocated in the previous phase are not considered spills, since they result from the topology of the datapath which enforces the existence of memory transfer operations. In other words, they result from the optimal selection of memory-register instructions in the given ISA, and not from the presence of resource conflicts.

Aho-Johnson [55] showed that, by using dynamic programming, optimal code can be generated in linear time for a wide class of architectures. The schedule they propose is based on their *Strong Normal Form Theorem*. This theorem guarantees that any optimal code schedule for an expression tree, for a given architectural model, can always be transformed into *Strong Normal Form* (SNF). A code sequence is in SNF if it is formed by a set of code sub-sequences separated by memory storages, where each code sub-sequence is determined by a *Strongly Contiguous* (SC) schedule. A code sequence is a SC schedule if it is formed as follows: at every selected match m, with child subtrees T_1 and T_2, contiguously schedule the instructions corresponding to subtree T_1, followed by the instructions corresponding to T_2, and finally the instruction corresponding to pattern m.

3.4.1 Problem Definition

SC schedules have been successfully used in the design of algorithms for homogeneous register set architectures (e.g. [44]). Unfortunately, SC schedules are not an efficient way to schedule instructions for heterogeneous register set architectures. They produce code sequences whose quality is extremely dependent on the order the subtrees are evaluated. Consider for example the IR tree of Figure 3.3(a). The expression tree was optimally matched using the approach proposed in Section 3.3 and the target ISA. It takes variables at memory positions m_0 to m_4 and stores the resulting computation into one variable at memory position m_6, using m_5 as a temporary storage.

The code sequences generated for three different schedules and its corresponding three-address representation are showed in Figure 3.3(b-d). Memory position m_7 was used whenever a spill location was required by the scheduler. For the code of Figure 3.3(b) the left subtree of each node was scheduled first, followed by its right subtree and then the instruction corresponding to the node operation. The opposite approach was used to obtain the code of Figure 3.3(c). Neither the SC schedules in Figure 3.3(b) and (c), nor any SC schedule will ever produce optimal code. This is obtained using a non-SC schedule that first schedules the addition $m_2 + m_3$ and then the rest of the tree, as in Figure 3.3(d). Notice that this schedule is indeed an SNF schedule, since first the subtree corresponding to $m_2 + m_3$ is contiguously scheduled followed by a storage operation into memory position m_5 and by another code sequence resulting from a SC schedule of the rest of the tree.

From Figure 3.3 one can verify how the appropriate SNF schedule minimizes spilling. For example, if the tree of Figure 3.3(a) is scheduled using left-first, the result of operation $m_0 \times m_1$ is first stored in p and then moved into a. Just after that, register a has to be used to route the result of $m_2 + m_3$ into memory position m_5. But a still contains a live result (the result of $m_0 \times m_1$).

In this case, the code-generator has to emit code to save the value of a into

memory and recover it later. This would not be required if the scheduler had first stored $m_2 + m_3$ into m_5, before loading a with the result of $m_0 \times m_1$.

Problems like the one illustrated above are very common in DSP architectures. The obvious question it raises is if there exists a guaranteed SNF schedule such that no spilling is required. Next section proves that this schedule does exist, under certain conditions that depend exclusively on the ISA of the target processor. Before moving to that, the problem needs to be defined formally: Given an optimally covered expression tree for an heterogeneous register architecture, determine an instruction schedule that does not introduce any spill code.

3.4.2 Problem Solution

This section is divided into two parts. First, it states and proves a sufficient condition that a heterogeneous register architecture has to satisfy in order to enable spill free schedules. Second, it provides, for these architectures, a linear time algorithm for optimal code scheduling of expression trees, and proves its optimality.

Allocation Deadlock

Let T be an expression tree with unary and binary operations. Let $L : T \rightarrow R \cup M$ be a function which maps nodes in T to the set $R \cup M$, where $R = \{r_i, 1 \leq i \leq N\}$ is a set of N registers, and M the set of memory locations. Let u be the root of an expression tree, with v_1 and v_2 children of u. Consider that after allocation is performed registers $L(v_1) = r_1$, $L(v_2) = r_2$ are assigned to v_1 and v_2 respectively. Let T_1 and T_2 be the subtrees rooted at v_1 and v_2, as in Figure 3.4. From now on the terms expression tree and allocated expression tree will be used interchangeably, with the context distinguishing if the tree is allocated or not.

Definition 2 *An expression tree contains an allocation deadlock if the following conditions are true: (a) $L(v_1) \notin M$, $L(v_2) \notin M$; (b) $L(v_1) \neq L(v_2)$ and (c) there exist nodes w_1 and w_2, $w_1 \in T_1$ and $w_2 \in T_2$ such that $L(w_1) = L(v_2)$ and $L(w_2) = L(v_1)$.*

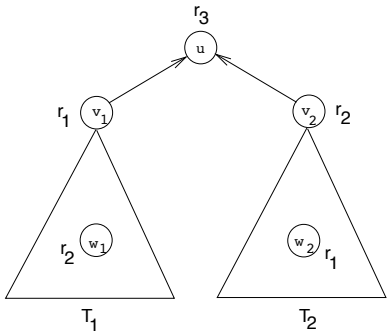


Figure 3.4: Allocation deadlock in an expression tree.

The above definition is illustrated in Figure 3.4. This is the situation where two sibling subtrees T_1 and T_2 contain each at least one node allocated to the same register as the register assigned to the root of the other sibling tree. Using this definition it is possible to propose the following result.

Lemma 1 *Let T be an expression tree. If T does not have a spill free schedule, then it contains at least one subtree which has an allocation deadlock.*

Proof. Assume that all nodes u in T are such that T_u is free of allocation deadlocks and that no valid schedule exists for T. According to Definition 2 T_u does not have an allocation deadlock when:

(a) $L(v_1) = M$ ($L(v_2) = M$). In this case, a SNF schedule exists if subtree T_1 (T_2) is scheduled first followed by subtree T_2 (T_1).

(b) $L(v_1) = L(v_2)$. This case cannot happen since no non-unary operator of an expression tree takes its two operands simultaneously from the same location.

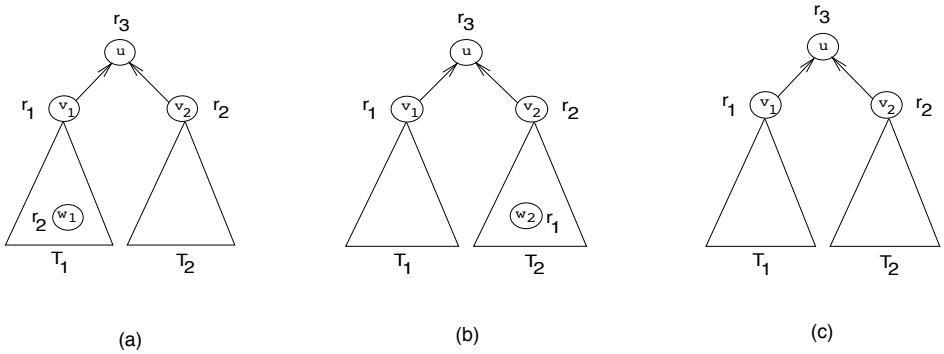


Figure 3.5: Trees without allocation deadlock.

(c) $L(v_1) \neq L(v_2)$, $L(w_1) = L(v_2)$, but no node w_2 exist for which $L(w_2) = L(v_1)$. In this case, it is possible to schedule T_1 first, followed by T_2 and the instruction corresponding to node u. This is a valid schedule because just after the schedule of T_1 is finished, only register r_1 is live, and therefore, since no register r_1 exists in T_2 no resource conflict will occur when this subtree is scheduled (Figure 3.5(a)).

(d) $L(v_1) \neq L(v_2)$, $L(w_2) = L(v_1)$, but no node w_1 exist for which $L(w_1) = L(v_2)$. This is symmetric to the previous case. Schedule T_2 first followed by T_1 and the instruction corresponding to u (Figure 3.5(b)).

(e) $L(v_1) \neq L(v_2)$, but no nodes w_1 and w_2 exist. This case is trivial, and any *SC* schedule results in a spill free schedule (Figure 3.5(c)).

Since the above conditions can be applied to any node u, T will have a valid schedule that is free of memory spilling code. This contradicts the initial assumption.

Lemma 2 *Let T be an expression tree. If T has no subtree containing an allocation deadlock, then it must have a spill free schedule.*

Proof. Directly from the theorem above.

The goal now is to determine those architectures for which the subtrees of any expression tree have no allocation deadlock. This is the problem studied next.

RTG Properties

In Chapter 2 it was shown that an ISA is acyclic if its architecture RTG has no cycles. The immediate consequence of that is the fact that any cyclic path in the architecture datapath has to go through memory. This is an important property of an acyclic ISA.

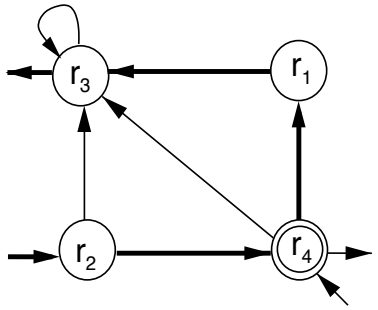


Figure 3.6: (a) The RTG section corresponding to registers r_1 and r_2; (b) A cycle in the datapath from register r_1 to r_2 has to pass through memory.

Property 1 Let r_1 be a register in the RTG of an acyclic ISA. Any datapath cycle containing this register, which is not a self loop, will traverse memory.

In particular, any cycle in the RTG which passes through a pair of registers r_1 and r_2 must at some point traverse memory. This allows another RTG property to be stated.

Property 2 Let r_1 and r_2 be registers in the RTG of an acyclic ISA. For each datapath cycle containing these registers, the path from r_1 to r_2, or the path from r_2 to r_1 will go through memory.

Example 3 Consider for example the RTG of Figure 3.6 and the register nodes r_1 and r_2. Many cyclic paths contain r_1 and r_2. Consider for example the cycle through nodes $\{r_2, r_4, r_1, r_3\}$, as shown by the dark edges of Figure 3.6. Observe that in order to return back to node r_2 from node r_3 the cycle has to use the outgoing edge out of r_3, and the incoming edge into r_2, or in other words, it must traverse memory.

The fact that no path from node r_3 to r_2 exists in the RTG is a property of the datapath, resulting from a decision of the architecture designer based on his (her) knowledge of the application domain. Later in this section this issue will be discussed in detail. Based on Property 2 an important result can now be proved.

The RTG Theorem

Lemma 3 *Let T be an expression tree generated from an acyclic ISA processor. T contains a set of subtrees which are free of allocation deadlock.*

Proof. The above statement can be proved using induction on the height of the tree.

Basis.

Let T' be the highest subtree containing no allocation deadlock, which can be generated in the target architecture. Let h' be the height of T'. It is obvious that all other trees, with height smaller than h', will have no allocation deadlock, since they can only be generated by removing nodes from T', and removing nodes does not create allocation deadlocks. Therefore, all trees for which the height $h \leq h'$ can be decomposed into subtrees that are free of allocation deadlock. In particular, T' itself is the highest of these subtrees.

Induction.

Let T be an expression tree of height h, with children v_1 and v_2. Let T_1 (T_2) be a subtree of T rooted at u_1 (u_2), and $h_1 \geq h'$ ($h_2 \geq h'$) its corresponding height. Assume, for the sake of clarity, that T_1 (T_2) is the left (right) subtree of T, as shown in Figure 3.7. Let $L(u) = r_3$, $L(v_1) = r_1$ and $L(v_2) = r_2$, be the registers assigned to these nodes after register allocation has been performed. Now assume that the statement is true for subtrees T_1 and T_2, but not for tree T. In other words, consider that T cannot be decomposed into subtrees which are free of allocation deadlock.

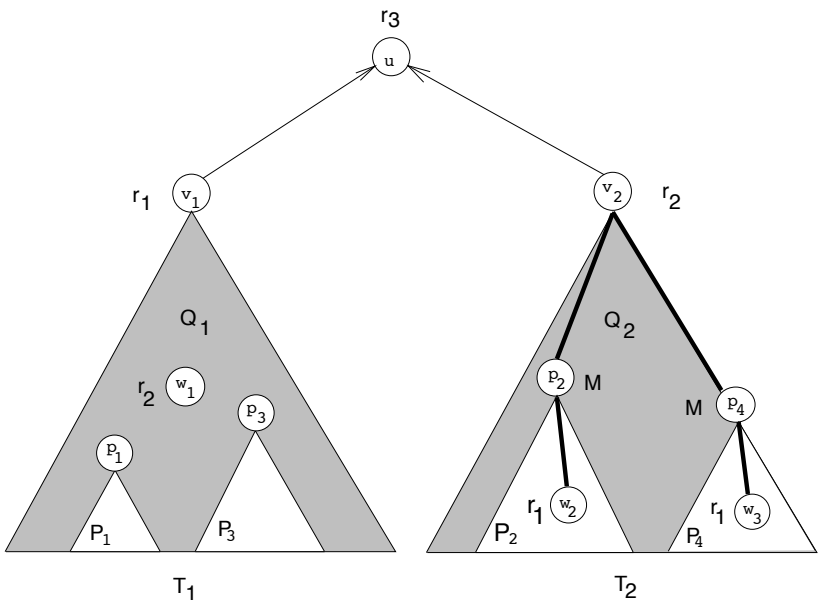


Figure 3.7: The RTG Theorem.

Given that the statement is true for T_1 and T_2, the only possible reason for T not to comply is if the allocation deadlock was caused by registers r_1 and r_2 from nodes u_1 and u_2. In this case, there exists at least a node w_1 (w_2) in subtree T_1 (T_2) which was allocated to register r_2 (r_1). In order to illustrate the generality of the argument, assume also the existence in Figure 3.7 of one more node (w_3) in T_2 for which $L(w_3) = r_1$. From Property 2 it was shown that, in an acyclic ISA, the path from r_1 to r_2 or the path from r_2 to r_1 has to go through memory. Assume, without loss of generality, that the subject architecture is represented by the RTG of Figure 3.6. In this case, any path from r_1 to r_2 will traverse memory. For the case of Figure 3.7, these paths are shown as dark lines going through nodes $\{w_2,\ p_2,\ v_2\}$ and $\{w_3,\ p_4,\ v_2\}$. Therefore, memory must have been allocated to nodes p_2 and p_4 in T_2 during register allocation. The same might have happened as well during the allocation of nodes p_1 and p_3 in subtree T_1. Observe that subtrees which have their root allocated to memory can be removed from the original tree and scheduled later, since a live

value in memory will not conflict with any other storage resource in the architecture. The conclusion which can be drawn here is that by removing subtrees P_2 and P_4 from T_2 the remaining tree T (formed by $T_1 \cup Q_2 \cup \{u\}$) will have no allocation deadlock (Lemma 1). Furthermore, if subtrees P_1, P_2, P_3 and P_4, which have roots allocated to memory (M), are removed from T, the remaining subtree (i.e. $Q_1 \cup Q_2 \cup \{u\}$) will be free of allocation deadlock as well. By hypothesis, T_1 and T_2 can be decomposed into subtrees which are free of allocation deadlock, and thus can P_1, P_2, P_3 and P_4. Therefore, T can be recursively decomposed into a set of subtrees which are free of allocation deadlock.

Theorem 1 [RTG Theorem] *If the ISA of processor P is acyclic, then for any expression tree generated from P there exists a schedule which is free of memory spills.*

Proof. This follows directly from two previous results: (a) the fact that any expression tree generated from an acyclic ISA can be decomposed into subtrees which are free of allocation deadlock (Lemma 3); and (b) the existence of a spill free schedule for any expression tree not containing an allocation deadlock (Lemma 2). For example, in the tree of Figure 3.7 the schedule can be determined by first scheduling the subtrees P_1, P_2, P_3 and P_4, followed by scheduling tree $Q_1 \cup Q_2 \cup \{u\}$.

3.4.3 Optimal Scheduling Algorithm

This section discusses a two pass algorithm for optimal code generation of expression trees for an acyclic ISA. The algorithm, listed in Figure 3.8, follows from the proof of Theorem 1. It takes an optimally allocated expression tree and produces spill free code. Its complexity is linear in the number of vertices n of the subject tree.

In its first pass, represented by procedure *GetUsage* (Figure 3.8) the algorithm does a traversal of the tree. At each node u two sets are computed: $u.memset$ and

```
GetUsage (u)
begin
    u.memset = φ;
    u.regset = φ;
    foreach v ∈ u.children
        GetUsage(v);
        if v.match is memory
            u.memset = u.memset ⋃ {v};
        else
            u.memset = u.memset ⋃ v.memset;
            u.regset = u.regset ⋃ v.regset ⋃ {v.match};
        endif;
    endfor;
end
```

```
        OptSchedule (u)
        begin
            foreach p ∈ u.memset
                OptSchedule (p);
            FreeSchedule (u);
        end


        FreeSchedule (u)
        begin
            if is_memory (u.match)
                return;
            if ¬ is_leaf (u)
                v1 = unique (u.children);
                FreeSchedule(v1);
                foreach w ∈ u.children – {v1}
                    FreeSchedule (w);
            endif;
            emit (u);
        end
```

Figure 3.8: First pass: GetUsage; Second pass: OptSchedule and FreeSchedule.

$u.regset$. Set $u.memset$ contains pointers to those nodes inside T_1 and T_2 which were allocated to M and that have no other node allocated to M on its path to u. One example of that node is p_1 in Figure 3.7. Set $u.regset$ keeps the names of the registers which were allocated to nodes inside subtrees Q_1 and Q_2. After *GetUsage* is finished it leaves the sets $u.memset$ and $u.regset$ inside each node u in the subject tree. This phase can be implemented efficiently if the computation of *GetUsage* is included in the task of selecting the best instruction match during the execution of the dynamic programming algorithm used to implement instruction selection and register allocation. In the second pass, *OptSchedule* executes a series of two tasks. First, for all $p \in u.memset$, *OptSchedule* recursively schedules all subtrees rooted at p. These subtrees are P_k, for $k = 1, 2, \ldots$ and correspond to the blank areas inside T_1 and T_2 in Figure 3.7. Once this is finished, different memory positions are live but not conflicting, and subtrees T_1 and T_2 are reduced to Q_1 and Q_2 respectively. Second, procedure *FreeSchedule* (the implementation of the proof of Lemma 2) traverses the subject tree. At each node it uses function *unique* to determine the child of u, say v_1, whose $v_1.match$ is a register that is not present in the $regset$ of the other child of u. The existence of v_1 is guaranteed by Theorem 1. Once v_1 is determined, the algorithm schedules its subtree Q_1. After that, register $v_1.match$ and a set of memory positions are the only live locations. Therefore, as Q_2 does not contain any node allocated to $v_1.match$, conflicts will not occur if Q_2 is scheduled after that.

Theorem 2 *Algorithm OptSchedule is optimal and is $O(n)$, where n is the number of nodes in the subject tree T.*

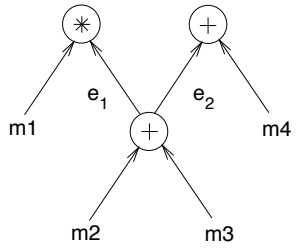
Proof. The first part is trivial, since *OptSchedule* implements the proof of Theorem 1. Also from Theorem 1 the algorithm divides T into a set of disjunct subtrees $(P_1 \ldots P_4,\ Q_1 \cup Q_2 \cup \{u\})$ and recursively schedules each of them. Therefore every node in T is visited only once. Hence, the algorithm is $O(n)$.

3.5 Heuristic for DAGs

Instruction selection for an expression DAG requires DAG covering which is known to be NP-complete [49]. In practical solutions to this problem heuristics have been proposed, which divide the DAG into its component trees by selecting an appropriate set of trees. However, this dismantling of the DAG into component trees is not unique and there are several ways in which this can be done. Traditionally, the heuristic employed in the case of homogeneous register architectures is to disconnect multiple fanout nodes of the DAG [23].

Dividing a DAG into its component trees requires disconnecting (or *breaking*) edges in the DAG. For the code generation task, breaking a DAG edge between nodes u and v implies the allocation of temporary storage to save the result of operation u while this is not consumed by operation v. This storage location is traditionally the memory but it can, in general, be any location in the datapath.

The key idea proposed here is a heuristic which uses architectural information from the RTG in the selection of component trees of a DAG, so as to minimize the number of memory spills in the resulting code. Consider for example the DAG of Figure 3.9. Notice that two different approaches can be used to decompose this DAG into its component trees, depending on which edge (e_1 or e_2) is selected to be broken. From now on, a broken edge will be represented by line segments orthogonal to the subject edge. As one can see in Figure 3.9(b) one extra instruction is generated when the dismantling heuristic is based on breaking edge e_2 instead of e_1. Incidentally, the code in Figure 3.9(a) is also the best sequential code one can generate from the subject DAG. Observe from the architectural description in Table 3.1, that the multiplication operation requests its operands from memory (m) and t, and that the result of the addition operation always produces its result in the accumulator a. Notice also from the TMS320C25 RTG of Figure 2.9 that to bring any data from a to register t one




```
lac m2      a ← [m2]
add m3      a ← a + [m3]
sacl m5     [m5] ← a
lt m1       t ← [m1]
mpy m5      p ← t * [m5]
add m4      a ← a + [m4]
```

(a)

```
lac m2      a ← [m2]
add m3      a ← a + [m3]
sacl m5     [m5] ← a
lac m5      a ← [m5]
add m4      a ← a + [m4]
lt m1       t ← [m1]
mpy m5      p ← t * [m5]
```

(b)

Figure 3.9: (a) Breaking edge e_1; (b) Breaking edge e_2.

has to go through memory.

From Figure 3.9 one can see that the result of the addition operation $m_2 + m_3$ has to be stored into a and must be moved to m or t in order to be used as an operand of the multiplication operation. But to move data from a to t one has to go through memory (m). Suppose the memory position selected to store this temporary result is m_5. Hence, by breaking DAG edge e_1 one is just assigning in advance a memory node which will appear on that edge, during the instruction selection phase of code generation. Notice that the existence of a register-transfer path which always goes through memory whenever data is moved from a to t, is a property of the target datapath. Similarly, the register-transfer path from a to p must also pass through memory.

Notice also that when edge e_2 is broken, pattern PLUS(a,m) (instruction *add* m_4) cannot be used to match the addition of m_4 with the result of $m_2 + m_3$ in the accumulator a. In this case, instruction *lac* m_5 in Figure 3.9(b) has to be issued in order to bring the data from m_5 back to the accumulator, adding a new instruction to the final code.

3.5.1 Problem Solution

The heuristic proposed to address the problem just described is divided into four phases. In the first phase partial register allocation is done for those datapath operations which can be clearly allocated before any code generation task is performed in the DAG. During the second phase architectural information is employed to identify special edges in the DAG which can be broken without introducing any loss of optimality for the subsequent tree mapping stages. In the third phase edges are marked and disconnected from the DAG. Finally, component trees are scheduled and optimal code generated for each component tree.

Partial Register Allocation

A general property of heterogeneous register architectures is that the results of specific operations are always stored in well defined datapath register classes (or *locations*). This does not imply total register allocation because data has to be routed through the datapath to locations required by other instructions. Take for example operations *add* and *mul* in the target processor. Notice that they implicitly define the primary storage resources that are used for the operation result. In this case, no register allocation task is required to determine that registers a and p are respectively used to store the immediate result of operations *add* and *mul* (observe Table 3.1). Thus, partial allocation can be performed well in advance, even before the task of breaking

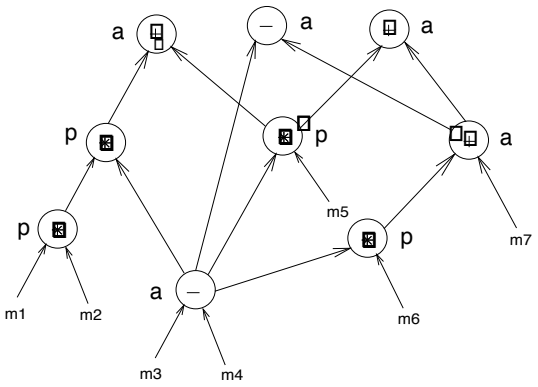


Figure 3.10: Expression DAG after partial register allocation phase.

the edges of the expression DAG takes place.

Example 4 Consider for example the expression DAG D of Figure 3.10. The labels on the side of its nodes show the register classes used to store the result of each operation after partial register allocation is performed.

Natural Edges

It was shown before, in Figure 3.9, that some edges have specific properties originating from the target architecture, which allow them to be disconnected from the DAG without compromising optimality. These edges, termed *natural edges*, are defined as follows.

Definition 3 [**Natural Edges**] *If the instruction matching edge* (u, v) *always produces a sequence of data transfer operations in the datapath which pass through memory, edge* (u, v) *is referred to as a natural edge.*

Now, given an expression DAG D (e.g. Figure 3.10), and an acyclic ISA, it can be shown that a number of edges in D are natural edges. In order to do that, some properties of DAG edges have to be proved.

Let r_1 and r_2 be a pair of registers in the datapath of an architecture which satisfies the RTG criterion. Also let $L : D \rightarrow R \cup M$, be a function which maps nodes in D into the set of datapath locations $R \cup M$, where R is the set of registers in the datapath and M is the set of memory positions.

Figure 3.11: (a) (u, v) is natural; (b) (u, v) is natural if r_i has no self-loop in RTG.

Lemma 4 *Let r_1 and r_2 be registers of an acyclic ISA, such that there exists a memory node on any RTG path from r_1 to r_2. Therefore any edge (u, v) in D for which $L(u) = r_1$ and $L(v) = r_2$ is a natural edge.*

Proof. Observe in Figure 3.6 that a path from registers r_1 to r_2 will be traversed whenever instruction selection is performed on edge (u, v). Thus, a memory operation will always be selected during instruction selection on (u, v) and therefore, (u, v) is a natural edge (Figure 3.11(a)).

Lemma 5 *Edges (u, v) for which $L(u) = L(v) = r_i$, $i = 1, 2$ are natural edges only if no self-loop exists on register node r_i in the RTG representation of the target architecture (Figure 3.11(b)).*

Proof. Property 2 states that any RTG cycle of an acyclic ISA, which is not a self-loop, will always traverse memory. Thus, if register r_i has no self-loop in the RTG, then any loop starting at r_i will go through memory. Therefore, memory *load/store* operations will be selected whenever instruction selection is performed on edge (u, v). Hence, (u, v) is a natural edge.

Notice that the task of breaking natural edges does not introduce any new operations into the DAG because, as the name implies, during the instruction selection phase a memory operation is naturally selected due to constraints in the architecture datapath topology. As a result, no potential optimality is lost by breaking natural edges.

Example 5 Consider each one of the lemmas above and the RTG for the TMS320C25 processor in Figure 2.9.

(1) Lemma 4 shows that when $r_1 = a$ and $r_2 = p$ every edge (u, v) such that $L(u) = a$ and $L(v) = p$ is a natural edge.

(2) Now Consider Lemma 5. First take the situation where $r_i = p$. From the RTG of Figure 2.9 observe that register p has no self-loop. Since the RTG is acyclic then any DAG edge (u, v) such that $L(u) = L(v) = p$ is a natural edge. Now consider the case where $r_i = a$. Register a in Figure 2.9 contains a self-loop and thus, nothing can be said regarding these edges.

Pseudo-Natural Edges

The following two lemmas show how DAG edges can sometimes interact such that a set of edges must result in storage in memory. These edges are called *pseudo-natural* edges.

Definition 4 [Pseudo-Natural Group] *Let u and w be operands of v as in Figure 3.12. A pseudo-natural group is the set $S = \{(u, v), (w, v)\}$, where at least one of the edges in S is a natural edge. The elements of S are called pseudo-natural edges.*

A *pseudo-natural* edge is an edge which when disconnected from the DAG has a non-zero probability of impacting optimality. Breaking a pseudo-natural edge does

not guarantee optimality since one cannot determine which edge in S is natural. However, there exist a non-zero probability that this will be the case.

Figure 3.12: (a) One of the edges is always natural; (b) Edge (w, v) is a natural edge.

Lemma 6 *Consider operation v and its operand nodes u and w in Figure 3.12(a). If partial register allocation of these operations is such that $L(u) = L(v) = L(w) = r_i$, $i = 1, \ldots |R|$, then set $\{(u, v), (w, v)\}$ is a pseudo-natural group.*

Proof. Notice that no binary operation v can take both its operands simultaneously from the same register. We have to consider here two situations:

(a) If node r_i has a self-loop in the architecture RTG, one of the edges, e.g. (u, v) could be matched by an instruction which takes one operand from r_i. On the other hand, when this same instruction matches the other edge, i.e. (w, v), it will make use of a register which is contained in an RTG loop (not a self-loop) that goes from r_i back to r_i. Similarly as in Lemma 5, matching (w, v) will introduce a sequence of transfer operations which necessarily goes through a memory node in the RTG, making (w, v) a natural edge.

(b) If no self-loop node r_i exists in the architecture RTG, then both edges are natural edges according to Lemma 5.

Lemma 7 *Consider operation v and its operand nodes u and w of Figure 3.12(b). Let the partial register allocation of these nodes be such that $L(u) = L(w) = r_j$ and*

$L(v) = r_i$. If all RTG paths between each pair of nodes are such that only one path does not go through a memory node, then $\{(u, v), (w, v)\}$ is a pseudo-natural group.

Proof. The proof is trivial and follows from the fact that since operation v cannot take both of its operands from the same register r_j at the same time, it has to use two paths in the RTG to bring data from register r_j. Since only one path from r_j to r_i does not go through memory, then the other path has to pass through memory and therefore, the corresponding edge is a natural edge.

Based on the lemmas above, one needs to decide which edge in group $\{(u, v), (w, v)\}$ is to be disconnected from the DAG. Loss of optimality might occur depending on which edge is selected. Unlike natural edges, breaking pseudo-natural edges might result in compromising the optimality of code generation for the component trees. However, there is a good chance that this might not happen. Pseudo-natural edges are identified using a double line segment to distinguish them from natural edge.

Example 6 Consider Lemmas 3 and 4 above and the RTG of TMS320C25 in Figure 2.9,

(3) Lemma 6 is satisfied for the case when $r_i = a$ or $r_i = p$.

(4) In this case if $r_j = p$ and $r_i = a$ only one path exists in the RTG from p to a and Lemma 7 can be applied.

After rules 1–4 of Examples 3 and 4 are applied, the expression DAG of Figure 3.13 results. Each marked edge in Figure 3.13 has on its side the number corresponding to the rule from Examples 3 and 4 used to disconnect the edge.

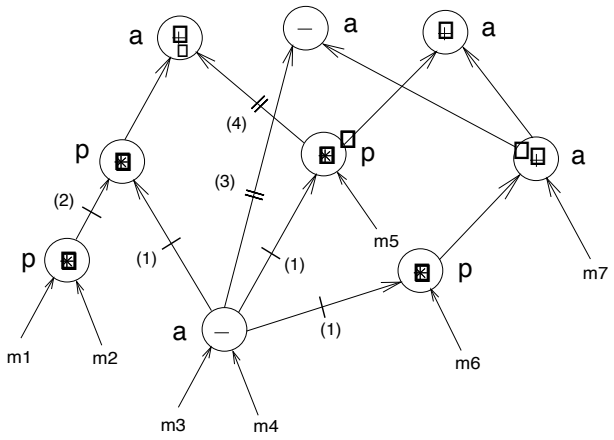


Figure 3.13: Expression DAG after natural and pseudo-natural edges have been identified.

3.5.2 Dismantling Algorithm

The task of dismantling an expression DAG may potentially introduce cyclic RAW dependencies between the resulting tree components leading to an impossible schedule. A similar problem was also encountered by Aho and Ullman [52] when they studied the problem of scheduling *worm-graphs* derived from DAGs in single-register architectures.

Consider, for example, the reconvergent paths from nodes u to v and the component trees T_1 and T_2 of Figure 3.14(a). Dismantling the DAG of Figure 3.14(a) requires that at least one of the edges of the multiple fanout nodes u and T_2 be disconnected. Assume that edges (u, T_2) and (T_2, v) have been selected as the edges to break. In this case nodes u, v and tree T_1 can be collapsed into a single component tree T_3, dismantling the DAG into trees T_3 and T_4. When an edge between two nodes is broken a *Read After Write* (RAW) edge is introduced (dark edges in Figure 3.14), in order to guarantee that the original data-dependencies are preserved by the scheduler.

In this case, the resulting RAW edges form a cycle between component trees T_3

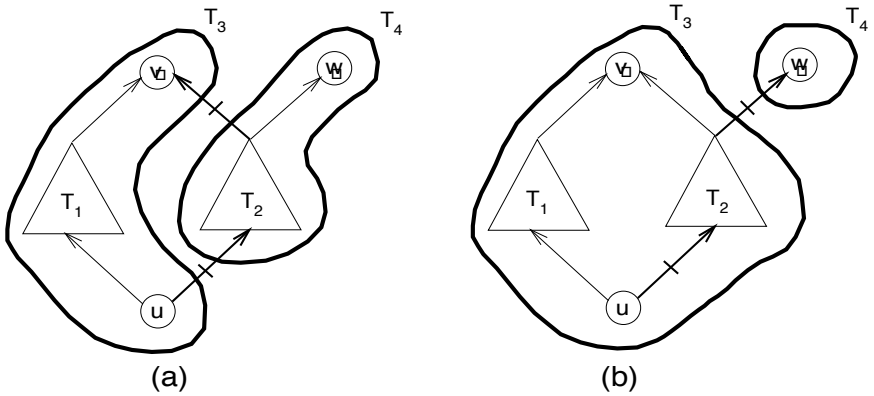


Figure 3.14: (a) Cyclic RAW dependency; (b) Constraining the tree scheduler.

and T_4, which results in an infeasible schedule for the component trees.

Notice that dismantling is also possible if edge (T_2, w) is broken instead of (T_2, v) (Figure 3.14(b)). When this occurs, RAW edge (u, T_2) is brought into the resulting component tree (T_3). As a consequence, the potential optimality of the tree scheduler algorithm in Section 3.4.3 can not be guaranteed anymore, since now it has to satisfy the constraint imposed by the new RAW edge (u, T_2) inside T_3. From the two situations analyzed above, it can be concluded that edges on both reconvergent paths have to be disconnected in order to guarantee proper scheduling of operations inside component trees and between component trees.

An algorithm which dismantles the DAG should disconnect edges by using as many natural and pseudo-natural edges as possible. Such an algorithm, called *Dismantle*, is shown in Figure 4.12. The *Dismantle* algorithm uses *MarkEdges* to mark those edges which will be disconnected from the DAG. Each node u in the DAG carries attributes $u.visit$ and $u.best$. Attribute $u.visit$ is set when node u is visited. Attribute $u.best$ is used to store the best edge found by the algorithm at that point.

MarkEdges visits each node u of the DAG. At each node it determines the type of the incident edge (v, u) and the best edge to break up to that point, which is stored in $u.best$. If $u.best$ is a marked edge, then $v.best$ takes the value of $u.best$. Otherwise,

```
Dismantle (D)
begin
    foreach u ∈ D.nodes
        MarkEdges (u);
    endfor;
    foreach e ∈ D.edges
        if e.mark
            BreakEdge (e);
        endif;
    endfor;
    foreach u ∈ D.nodes
        if |successors (u)| > 1
            count = 0;
            foreach v ∈ successors (u)
                e = (u,v);
                if count < |successors (u)|
                    count = count + 1;
                    BreakEdge (e);
                endif;
            endfor;
        endif;
    endfor;
end
```

```
MarkEdges (u)
begin
    u.visit = true;
    foreach v ∈ predecessor (u)
        save = v.best;
        e = (v,u);
        if (u.best).mark
            v.best = u.best;
        else if IsNatural (e)
            v.best = e;
            e.mark = true;
        else if IsPseudoNatural (e)
            if (u.best).mark
                v.best = u.best;
            else
                v.best = e;
            endif
        endif
        if v.visit
            (v.best).mark = true;
            save.mark = true;
        else
            MarkEdges (v);
        endif
        u.visit = false;
    endfor
end
```

Figure 3.15: Algorithm to dismantle DAG based on natural and pseudo-natural edges.

the type of the incident edge (v, u) is determined. If (v, u) is a natural edge, then this edge is marked. On the other hand, if (v, u) is a pseudo-natural edge, then this is compared with the type of $u.best$. If $u.best$ is marked, then the best edge at v is $u.best$. The reason for this is that pseudo-natural edges should have lower breaking priority when compared with natural edges, or edges which have already been marked. When a reconvergent path is found, i.e. $v.visit$ is true, paths are marked to break. Otherwise, the algorithm recurs on the predecessors of u.

After the execution of *MarkEdges*, *Dismantle* uses $BreakEdge(u)$ to break all but one of the outgoing edges from u. This guarantees that no node will have shared operands. Broken edges which are not natural nor pseudo-natural edges are identified by a dark circle mark in Figure 3.16(a). Broken edges are then disconnected from the DAG, temporary memory nodes are created and RAW edges introduced between component trees.

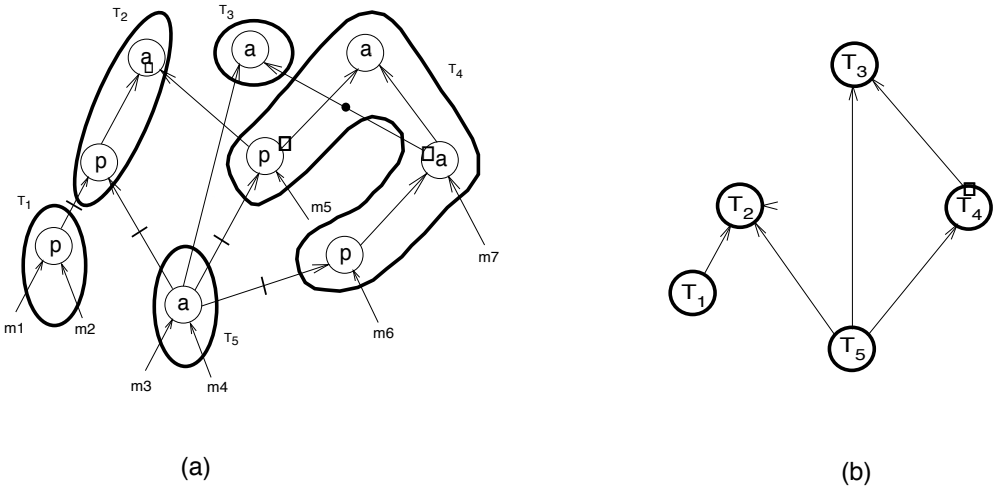


Figure 3.16: (a) Resulting component trees after dismantling; (b) Component trees graph.

After algorithm *Dismantle* is executed, the DAG is decomposed into its component trees T_i $(i = 1, 2, \ldots)$, as shown in Figure 3.16(a). The component trees form a new graph, shown in Figure 3.16(b). In the graph of Figure 3.16(b) nodes are associated

to component trees T_i, and edges are used to represent the RAW constraint edges from Figure 3.16(a). Topological ordering is then performed in order to schedule the component trees. Finally, optimal code is generated for each component tree using the algorithm proposed in Section 3.4.3.

3.6 Experimental Results

DSPstone [72] is a benchmark designed to evaluate the code quality generated by compilers for different DSP processors. This is the benchmark that will be used for the experimental evaluation of the algorithms just studied.

DSPstone is divided into three benchmark suites: *Application, DSP-kernel and C-kernel*. The Application benchmark consists of the program *adpcm*, a well-known speech coding algorithm. The DSP-kernel benchmark consists of a number of code fragments, which cover the most often used DSP algorithms [72]. The C-kernel suite aims to test typical C program statements. *DSPstone* project was supported by a number of major DSP manufacturers (Analog Devices, AT&T, Motorola, NEC and Texas Instruments). It is not an ideal benchmark, since it is heavily based on kernels and not on complete applications. Actually, this tendency in benchmarking programs for DSP applications can also be observed in other well publicized benchmarks, like the *BDT Benchmark* [73]. The reasoning behind that is based on three major facts. First, kernels play a very important role in the execution of a DSP program, more than in other application domains. Second, DSP programs are traditionally written in assembly. In this case, the reference assembly code for a complete application benchmark could take years of engineering effort [73]. Third, the large majority of the application programs developed for DSPs are proprietary. Furthermore, much of those in the public domain are designed using floating-point variables for the purpose of simulation only. Hence, they are not suitable for compiling in fixed-point machines

(e.g. programs for cellular telephone standards like GSM and IS54).

3.6.1 Expression Trees

Algorithm *OptSchedule* (Section 3.4.3) was applied to two expression trees from each DSP-kernel benchmark in DSPstone. Code was also generated for each tree using Left-first and Right-first SC schedules. The results are shown in Table 3.2. Each kernel is represented by a pair of consecutive trees.

The metric used to compare the code from different schedules was the number of cycles that takes to compute the expression tree. Given that no branch instruction is executed, this is almost the same as measuring the code size of the instructions generated for that tree, given that instructions in a DSP architecture are designed to execute in a single machine cycle. As mentioned before in Chapter 2, code size is as important metric for the DSPs (if not more important) as execution time. The reason for that has to do with the restricted on-chip ROM size available to store the program, in which case one byte can make the difference between being able to store the program or not.

Observe from Table 3.2 that algorithm *OptSchedule* produces the best code when compared with the two SC schedules, what was expected since we have proved its optimality. Notice that although SC schedules can sometimes produce optimal code, they can also generate bad quality code, as it is the case for expression tree 4. One can also verify that the same expression tree generates different code quality when different SC schedules are used. The structure of the expression tree dictates the best SC schedule, and this structure is a function of the way the programmer writes the code. The numbers surrounded by boxes in Table 3.2 are associated to expression trees for which optimal code can only be guaranteed by *OptSchedule*.

Observe that for many expression trees the use of algorithm *OptSchedule* makes no difference. Based on that, it may be felt that the final impact of *OptSchedule*

Tree	Origin	Scheduling Algorithms		
		Left-first	Right-first	*OptSchedule*
1	biquad_N	10	14	10
2		4	4	4
3	biquad_one	8	10	8
4		14	12	10
5	complex_multiply	7	9	7
6		7	9	7
7	complex_update	8	8	8
8		8	8	8
9	convolution	7	7	7
10		3	3	3
11	dot_product	7	7	7
12		2	2	2
15	fft	8	10	8
16		5	5	5
13	fir	7	7	7
14		7	7	7
17	fir2dim	7	7	7
18		7	7	7
19	lms	7	7	7
20		5	5	5
21	matrix_1	7	7	7
22		3	3	3
23	matrix_1x3	9	7	7
24		3	3	3
25	n_complex_updates	3	3	3
26		3	3	3
27	n_real_updates	7	7	7
28		3	3	3
29	real_update	3	3	3
30		3	3	3

Table 3.2: Number of cycles to compute expression trees using: Right-Left, Left-Right and *OptSchedule*; Tree numbers in a box show cases where *OptSchedule* generates optimal code which cannot be guaranteed by SC scheduling.

is negligible for a large applications. Nevertheless, it is important to mention that DSP applications demand the best possible code. The fact that inner loop kernels are extremely critical for these applications reinforces the thesis that optimality must be guaranteed. For example, consider an inner loop basic block of size 6, which is a single instruction longer than the optimal solution. Consider, for the sake of simplicity, that the number of *Cycles per Instruction* (CPI) of the instructions in this basic block is 1 cycle/instruction. Assume also that the inner loop has to be executed in at most 5.7 million cycles, in order to meet the real-time constraint demanded by the application[1]. If the extra instruction takes one cycle to execute, then the inner loop will require 6 million cycles to execute, and the application will not be able to meet its time constraints. This scenario is very typical in the design of DSP applications, and it is common for DSP programmers to struggle for weeks just to meet the code size and performance constraints.

The experimental approach above departs considerably from the one used in general-purpose computing, where *SPEC* is used to analyze large applications. It is the opinion of the author that this is not a major drawback since *OptSchedule* has been proved correct. Moreover, the contribution here goes beyond the practical benefits offered by *OptSchedule*, since it also brings new insights in the interaction between code generation and datapath topology.

3.6.2 DAG Types Distribution

A thorough understanding of the importance of expression tree code generation, for DSP applications, can be captured by measuring how frequent these data structures occur in DSP program basic blocks. Expression DAGs were classified as trees, leaf DAGs and full DAGs. Leaf DAGs are DAGs for which only leaf nodes have outdegree greater than one. A DAG was classified as a full DAG if it is neither a tree nor a leaf

[1]For simplicity, let the overhead due to loop control be zero.

DAG. The need to separate full DAGs from leaf DAGs in the classification, comes from the existence of memory-register instructions in acyclic ISA and constant leaf nodes, for which breaking the outgoing edge from the shared node implies no loss of optimality.

DSP kernel	Basic Blocks	Trees	Leaf DAGs	DAGs
biquad_N	6	5	0	1
biquad_one	34	31	2	1
complex_multiply	6	4	2	0
complex_update	5	3	1	1
convolution	25	22	3	0
dot_product	16	14	2	0
fft	31	28	2	1
fir	25	21	3	1
fir2dim	125	111	12	2
lms	36	30	4	2
matrix_1	62	56	5	1
matrix_1x3	23	22	1	0
n_complex_updates	12	11	0	1
n_real_updates	25	21	3	1
real_update	5	5	0	0

Table 3.3: Types of DAGs found in typical digital signal processing algorithms.

As one can see from Table 3.3, the classification reveals that of all 436 basic blocks analyzed 88% were trees, 9% were leaf DAGs and 3% full DAGs. From the numbers in Table 3.3 one can notice that the vast majority of the DAGs are trees or leaf DAGs (97%). Another experiment was performed, this time using the DSPstone application benchmark *adpcm*. As before, basic blocks were analyzed to determine the frequency of trees, leaf DAGs and full DAGs. In this case 94% of the basic blocks in *adpcm* were found to be trees, 3% leaf DAGs and 3% full DAGs. Although dynamic counting of the basic blocks execution profile would be required in order to provide information on the impact of this distribution on execution time, one can reasonably argue that

a large portion of the execution time in DSP programs is spent processing expression trees. Hence, tree based code generation is very suitable for this application domain. This fact has been verified a long time ago for general purpose applications, but has not been studied for DSP programs.

3.6.3 Expression DAGs

Table 3.4 lists a series of expression DAGs extracted from programs in the DSP-kernel benchmark. The largest DAG, found in each kernel, was selected for the purpose of comparison with hand-written code. Since the goal here is to study the impact of the DAG dismantling technique proposed before, only DAGs which are not expression trees were considered[2]. Hand-written assembly code (or *assembly reference code*) for each DSP-kernel program is available from the DSPstone benchmark suite [72]. Compiled code was generated for each DAG and the resulting number of cycles for a single loop execution reported in Table 3.4. Compiled code was also generated using a *Standard* heuristic, which dismantles the DAG by breaking all edges at multiple fanout nodes (column *Standard Heuristic*). Table 3.4 shows the number of processor cycles with respect to hand-written code. Notice that the overhead is only due to the DAG dismantling technique.

The average overhead when comparing the compiled code with the assembly reference code was then computed. The overhead due to the *Dismantle* heuristic (3%) was smaller than when using the *Standard* heuristic (6%). Leaf nodes were treated the same way in both heuristics. They are simply duplicated into different nodes - one for each outgoing edge. As a consequence, both heuristics have the same performance for the case of leaf DAGs. The average overhead of the *Dismantle* heuristic for the case of full DAGs was higher (7%) than for the case of leaf DAGs (3%). The discrepancy

[2]Observe that *real_update* is not listed, since it does not contain any expression DAG which is not a tree.

DAG	Origin	DAG Type	Hand-written Code	*Standard* Heuristic	*Dismantle* Heuristic
1	biquad_N	F	12	12	12
2	biquad_one	F	15	17	15
3	complex_multiply	F	17	17	17
4	complex_update	F	16	18	18
5	convolution	L	5	5	5
6	dot_product	L	7	7	7
7	fft	L	9	9	9
8	fir	L	4	5	5
9	fir2dim	L	5	5	5
10	lms	F	7	9	8
11	matrix_1	L	5	5	5
12	matrix_1x3	L	5	5	5
13	n_complex_updates	L	7	7	7
14	n_real_updates	L	9	9	9

Table 3.4: Experiments with DAGs – Leaf DAG (L); Full DAG (F). Numbers in a box mark DAGs for which the *Dismantle* heuristic results in a better code than the *Standard* heuristic.

is probably due to the existence of memory-register and immediate instructions in the processor ISA, which can have zero cost multiple fanout operands, when these are memory references or constant values.

The expression DAGs for which *Dismantle* resulted in a better code than the *Standard* heuristic have their numbers in Table 3.4 surrounded by boxes. Although the heuristic gains may seem very small, as mentioned before, the generation of high quality code for inner loops is crucial in compiling for DSPs.

3.7 The Case of Cyclic ISA

The work developed up to this point deals with code generation for acyclic ISA architectures, or architectures for which the RTG is acyclic. Although these architectures

have been largely used in designing DSPs, the recent tendency in DSP design is to adopt register-register (or *load-store*) architectural styles. Modern DSP processors like the ADSP 2100 and the Motorola 56000 are examples of such architectures. The rationale behind that is simple – *registers are faster than memory*. Although on-chip DSP memories are designed using SRAMs, which are faster than external DRAMs, the statement still holds true because of the slow down in SRAM performance due to the increasing size of on-chip programs. In order to clarify the impact of an RTG cycle in code generation, it is important to understand the reasons why cycles are introduced in a datapath, and how these cycles are used during code generation.

3.7.1 The Need for Cycles

The target DSP adopted for this study is the Motorola 56000 processor. The Motorola 56000 is a popular DSP, and its datapath is very representative of a typical cyclic ISA. The Motorola 56000 datapath is showed in Figure 2.14(a). It contains three register files: A, RX and RY. Register file A is an accumulator file containing two registers (a_0 and a_1) which is used to store the result of the arithmetic operations. Register file RX (RY) is used to store operands, and contains two registers as well – x_0 and x_1 (y_0 and y_1).

Instruction	Cost	Operands	Destination	Three-address
alu x_i, a_j	1	$x_i,\ a_j$	a_j	$a_j \leftarrow x_i\ op\ a_j$
alu y_i, a_j	1	$y_i,\ a_j$	a_j	$a_j \leftarrow y_i\ op\ a_j$
alu a_i, a_j	1	$a_i,\ a_j$	a_j	$a_j \leftarrow a_i\ op\ a_j$
mul x_i, y_j, a_k	1	$x_i,\ y_j$	a_k	$a_k \leftarrow x_i\ *\ y_j$

Table 3.5: Partial ISA of the Motorola 56000 processor.

A brief description of the Motorola 56000 instruction set is represented in Table 3.5. As discussed before in Chapter 2, the Motorola 56000 is a DLE architecture,

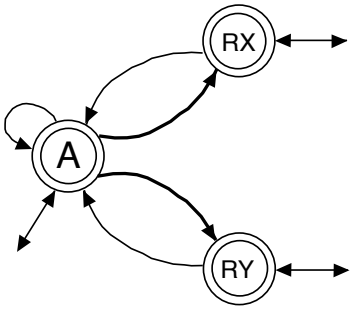


Figure 3.17: The Motorola 56K RTG representation.

i.e it allows one arithmetic operation and two memory transfer operations to occur during the same machine cycle. The code generation approach used in this thesis is divided into two phases. First, sequential code generation is performed, followed by code compaction. Since the problem studied here deals only with sequential code generation, assume that each instruction performs a single operation in each cycle.

In order to simplify the analysis, only arithmetic operations are represented in Table 3.5. These operations are divided into two groups. The first group contains all operations which use the datapath ALU, and are represented by instruction *alu* in Table 3.5. This includes operations like: addition, subtraction, shifting, etc. The main characteristic of an *alu* operation is that one operand register (e.g. a_j) must always be from the accumulator register file, which is also used to store the operation result. The other group is formed by the multiplication operation (*mul*). Observe from Figure 2.14(a) that the multiplier requires its operands to be in the registers of files RX and RY, and stores the result into an accumulator. This restriction is an immediate consequence of the dedicated datapath topology of the MAC unit.

The RTG for the Motorola 56000 is shown in Figure 2.14(b), and is repeated in Figure 3.17 for convenience. The paths in the RTG responsible for the cyclic nature of the processor ISA are shown as dark lines in Figure 3.17. Observe that they originate from the need to transfer data from the accumulator register file A to

register files RX and RY. But this transfer operation only occurs when the result of two *alu* operations stored in A are needed in RX and RY as operands for the multiplication. Since both operands of the multiplier have to be in RX and RY, the use of the dark paths in Figure 3.17 eliminates the need for instructions to move the multiplication operands into memory and after that into RX and RY. This is a very specific situation which should occur frequently in the program running on this architecture, in order to justify the existence of such paths. This insight opens up an opportunity to explore the trade-off between the availability of a path in the processor datapath, its effective usage by the application, and its impact in the design of the code generation algorithm.

3.7.2 Extending to Cyclic ISAs

Another important aspect learnt from the RTG model is that expression trees which have no allocation deadlocks will always have spill free schedules independent of the existence of cycles in the architecture RTG. This is because an expression tree which is free of allocation deadlocks will not perform any cyclic register-transfer operation. In other words, the RTG cycles in the architecture are opaque for that particular expression tree. An interesting consequence of that is the following heuristic. If it is possible to guarantee that no allocation deadlock will exist in an expression tree T after instruction selection and register allocation are performed, then an optimal schedule for T exists which is independent of the RTG structure. This can be done by the following greedy approach.

Algorithm 1 Let v be the root of an expression tree T, v_1 and v_2 its children, and T_1 and T_2 subtrees rooted in v_1 and v_2. Using a pre-order depth first traversal of T do the following:

[1] Compute for node v_1 (v_2) the set $v_1.regset$ ($v_2.regset$) containing the names of the registers used in subtree T_1 (T_2).

[2] Assign registers $L(v_1) = r_1 \notin v_2.regset$ and/or $L(v_2) = r_2 \notin v_1.regset$ respectively to v_1 and v_2, such that T is free of allocation deadlock.

Algorithm 1 can be programmed together with the dynamic programming algorithm used to perform instruction selection and register allocation, as discussed in Section 3.3. For small size basic blocks seems it is possible that a large fraction of

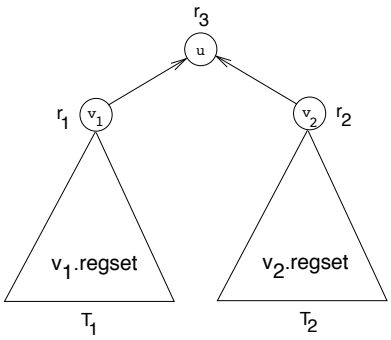


Figure 3.18: The use of allocation deadlock avoidance in register allocation.

the expression trees are naturally free of allocation deadlock and thus, are suitable for Algorithm 1. For example, in the case of the Motorola 56000, only a specific combination of multiplication and *alu* operations can lead to allocation deadlocks. All other expression trees which do not have this specific combination are free of allocation deadlocks, and these are not a few in a typical program.

Its is certainly true that the argument above is speculative and has no hard facts to support it. Nevertheless, it is reasonable to believe that the allocation deadlock avoidance technique described in Algorithm 1 can produce good results for a considerable number of expression trees in cyclic ISA. The time and future experiments will be the judge of this hypothesis.

Chapter 4

Code Generation for Address Computation

4.1 Introduction

In Chapter 3 the problem of basic block code generation for DSPs was thoroughly studied. This chapter focus on another important problem in compiling for these architectures. Address computation constitutes a large fraction of the execution time for most programs. Addressing can account for over 50% of all program bits and 1 out of every 6 instructions for a typical general-purpose program [74]. These numbers are probably much higher for the case of DSP programs, given that accessing data-streams is a very common task in such applications. Although computing the address requires a number of simple arithmetic operations, the best DSP compilers available today (and some CISC[1] compilers as well) do not generate very efficient addressing code. This chapter proposes techniques, based on addressing mode properties which can generate high-quality code for address computation in DSP architectures.

In the case of DSP applications, data is frequently stored in arrays and access is made through array indexing. Alternatively, pointers can be used to directly address

[1]Complex Instruction Set Computer

the data. Computing the address of an array element involves adding an *offset* to the *base* address of the array. The *offset* is computed as the *index* of the array multiplied by the *size* of the stored object. In any case, the generation of addressing code can be done implicitly by the compiler, when array indexing is used, or explicitly by the programmer, when the access occurs by means of pointer variables. Traditionally, many DSP architectures offer specialized *addressing mode* instructions which enable fast computation of the data address. A typical example is the auto-increment (decrement) mode (see Chapter 2 for details), in which the address register is incremented (decremented) after the access operation is completed.

Code generation techniques for address computation in general-purpose architectures have been studied before. Horwitz *et. al* [75] proposed the first algorithm for optimal allocation of index registers for addressing operations in straight line code. Hitchcock [74] studied many of the problems involved in using addressing modes for these architectures. Code generation for address computation in DSPs architectures only recently has gained some attention. The problem of allocating local variables to the *stack-frame* in DSP architectures was first formulated and solved by Bartley [76]. Since then, the solution of this problem has been extended by Liao *et. al.* [28], who named it *Offset Assignment Problem*. Offset Assignment is the problem of assigning to the local variables in a program, an offset with respect to the beginning of the stack, such as to maximize the usage of the auto-increment (decrement) addressing mode. The use of different addressing modes for DSP architectures has also been explored recently by Liem *et al.* [77].

4.2 The Address Generation Unit

Due to the hard performance constraints found in the DSP application domain, DSP architectures have hardware units, known as *Address Generation Units* (AGU), which

enable fast address computation. These units are present in all commercial DSPs, and in many CISC architectures such as the Motorola 68000 and the VAX PDP-11/780. A typical AGU contains a number of *Address Registers* (*ARs*) and an ALU that can perform basic arithmetic operations such as increment/decrement. The *ARs* are used by indirect addressing mode instructions to point to the memory position where the desired data is stored. This permits the design of instructions with different addressing modes like: *linear*, *offset-based* and *modular* (see Chapter 2 for details).

DSP architectures which have a single data memory bank, such as the TMS320C25, have only one AGU to address data in that bank. As mentioned in Chapter 2, *Dual-Load Execute* (DLE) DSP architectures have been largely used in the design of modern DSPs. The main advantage of this architecture is that it permits one ALU operation and two memory operations to occur concurrently. DLE architectures have two AGUs, one for each data bank, as shown in Figure 4.1. This architecture style enables instructions which are capable of performing three tasks in a single machine cycle: (a) execute an ALU operation; (b) load the operands used in the next instruction; and (c) compute the address of the operands for the instruction following that. Since the AGU executes in parallel with the datapath, the architecture can hide the latency of the address computation operation, while executing the current instruction.

Figure 4.1 shows a DLE architecture with two AGUs (X-AGU and Y-AGU). Assume that a program is executing in that architecture which makes use of two arrays a and b, stored in memory banks MX and MY respectively. The elements of array a are accessed using address register AR_x (X-AGU), while the elements of b are accessed through AR_y (Y-AGU). The program reads an element $a[N * i]$ of a, for each element $b[i]$ of b. The program uses offset addressing mode (the offset is N) to access a and linear (post-increment) mode to access b.

A more detailed example can give a better understanding of the problems involved in generating code for address computation. Consider, for example, the code in

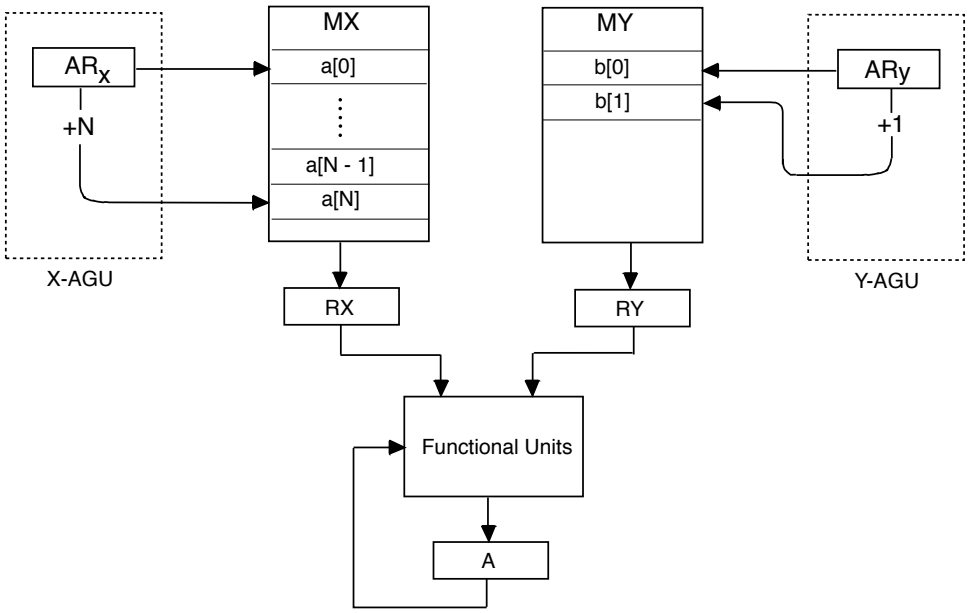


Figure 4.1: Computing the address of operands in a DLE architecture.

Table 4.1 which implements the body of a loop construct containing the statement $sum = sum >> 2 + a[i] * b[i] + a[i-1] - b[i+1]$. The three-address code corresponding to the statement is formed by a sequence of four instructions. Each instruction is divided into its operation components by vertical bars.

As before, two arrays are used in the computation of sum, array a and b, which are stored in memory banks MX (array a) and MY (array b). Assume that before instruction (1) is executed, address register AR_x (AR_y) points to the second element of array a (b), i.e. $a[1]$ ($b[1]$), and that the accumulator A contains the initial value of sum. During the execution of instruction (1) the accumulator A is shifted right by two and register RX (RY) loaded with elements $a[1]$ ($b[1]$). At the same time, address register AR_x (AR_y) is decremented (incremented). Notice that this update occurs **after** AR_x and AR_y are used to load the operands requested by the **mac** operation in instruction (2). During instruction (2) registers RX and RY are multiplied and added to the accumulator, RX is loaded with $a[i-1]$ and two is added to AR_x.

shift 2	**ld** RX, $[AR_x --]$	**ld** RY, $[AR_y ++]$		(1)
mac RX, RY	**ld** RX, $[AR_x + 2]$			(2)
add RX		**ld** RY, $[AR_y]$		(3)
sub RY				(4)

Instruction 1	Instruction 2	Instruction 3	Instruction 4
$A \leftarrow A >> 2$	$A \leftarrow RX * RY + A$	$A \leftarrow A + RX$	$A \leftarrow A - RX$
$RX \leftarrow [AR_x]$	$RX \leftarrow [AR_x]$	$RY \leftarrow [AR_y]$	
$RY \leftarrow [AR_y]$	$AR_x \leftarrow AR_x + 2$		
$AR_x \leftarrow AR_x - 1$			
$AR_y \leftarrow AR_y + 1$			

Table 4.1: Example of address computation in DLE architecture.

During the execution of instruction (3) RX (i.e $a[i-1]$) is added to A (i.e. sum) at the same time that RY loads $b[i+1]$. Finally, during instruction (4) RY (i.e. $b[i+1]$) is subtracted from A. Now it will become clear why two is added to AR_x in instruction (2), while AR_y is not modified in (3). The reason for that has to do with the values required for AR_x and AR_y in the first instruction of the next iteration of the loop. During the next iteration, the loop induction variable i is incremented (i.e. i becomes $i+1$), and therefore, the load operations in instruction (1) will read elements $a[i+1]$ and $b[(i+1)-1] \equiv b[i]$. Observe that the element pointed by AR_x just after the execution of (1) in the previous iteration was $a[i-1]$. Assume, without loss of generality, that the array elements have the same size as the machine word. In this case, in order to redirect AR_x from $a[i-1]$ in the current iteration to $a[i+1]$ in the next iteration, two has to be added to AR_x before the end of the loop. This can be done in parallel to the execution of instruction **mac**, by means of the offset addressing mode ($[AR_x + 2]$ in instruction (2)). Similarly, the value of AR_y after instruction (4) points to $b[i+1]$, since AR_y was incremented in (1). Given that in the next loop

iteration AR_y has to point to $b[i+1]$, then there is no need to update the value of AR_y before the end of the loop. The code sequence above is optimal, in the sense that no other sequence of instructions can compile the same source level statement using fewer machine cycles and registers. Achieving this quality for addressing code is not a simple task though. The next section describes an overview of the problems involved in this task, and gives an approach on how to tackle them.

4.3 The Address Code Generation Problem

Generating high quality code for address computation is a very important task for a DSP compiler. One of the major goals of the compiler, when optimizing for address computation, should be to guarantee that the AGU is effectively used by the source program. Efficient usage of the AGU requires the following two tasks to be performed well: (a) identification of an addressing mode; and (b) allocation of address registers to addressing operations.

The first problem, identifying the possibility of using an addressing mode, is the task of allocating *virtual address registers* to pointer variables and array references in the program. For example, consider a pointer variable $p++$ with the post-increment operator. It is clear that p can be allocated to a virtual post-increment register. Similarly, offset and post-increment modes can also be used to redirect pointers to array elements, as was shown before in the code of Table 4.1. Section 4.4 below describes techniques for the solution of this problem. This approach assumes that the AGU has an infinite set of *virtual address registers*. The second task in address code generation is the allocation of physical registers to the virtual address registers assigned before. This problem is well-known in the compiling literature, and a discussion of the possible solutions for the DSP case are provided in Section 4.5.

4.4 Virtual Address Register Allocation

The problem of virtual address allocation can be divided into two sub-problems. The first is allocation of address registers (*ARs*) to array indexing, which is called the *Array Index Allocation* problem. The second is the allocation of address registers to pointer variables. These problems are studied in Section 4.4.1 and Section 4.4.2 respectively. For the sake of simplicity, consider that every reference to an *AR* in this section is a reference to a *virtual AR*.

4.4.1 Array Index Allocation

It is a consensus among DSP programmers that array accesses ought to be transformed to pointer operations. The main reason for that is the inability of compilers to perform efficient allocation of *ARs* in the presence of array accesses. Although some researchers have addressed this issue before [77], transformation to pointers is still considered the technique of choice.

The problem of allocating virtual *AR* to array references is termed *Array Index Allocation Problem*. This section proposes a formulation and a solution for this problem. The goal here is to take advantage of the properties of the AGU to perform efficient reference to array elements. Auto-increment (decrement) is a common feature found in the AGU of all DSP architectures. Some DSP architectures have more elaborate versions of AGUs, which allow adding/subtracting to an *AR* other quantities besides one. These quantities are usually stored in *offset registers* (*OR*s) before the address computation is performed (see Chapter 2). Different addressing modes can be modeled by assuming that the *AGU* has just the ability to add a fixed amount *offset* to the *AR*. The quantity *offset* can be any positive or negative integer value, generated by any addressing mode (linear, offset or modular). Observe that by making $offset = 1(-1)$ this model also takes care of the auto-increment (decrement) cases.

The presence of such features in the AGU gives great flexibility to the programmer. Consider, for example, a program which uses an array of ten elements as shown in Figure 4.2. Assume that each array element is an object of size *size*. The case when $size = 1$ is very common in applications running in DSP processors. In fact, due to performance requirements, these processors are designed such that the size of the memory word matches the size of the program data[2]. If the elements of *array* are accessed sequentially, then the address of the array elements can be automatically computed in the AGU by continuously adding *size* to the AR. Given that the major-

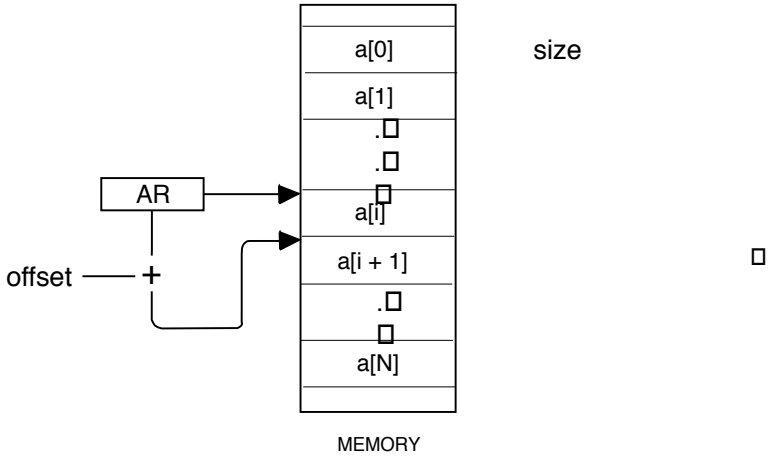


Figure 4.2: Addressing array elements.

ity of the array references are made within loop constructs, understanding how array addressing occurs within a loop should be the goal of array index allocation.

Array References in Loops

Assume that a sequence of array references (or *accesses*) occurs within the body of a loop statement. Consider also, for the sake of simplicity, that the loop body contains a single basic block and that no control statements (e.g. **if-the-else**) are allowed within the loop body. This constraint is not a requirement of the technique though. It will

[2]Which is usually 16 bits.

be shown later how to tackle the cases when control statements exist within the loop body. In the code generation approach used here, the *Intermediate Representation* (IR) of an array reference is not decomposed into its atomic operations. In other words, the references to array elements are maintained until the final schedule is performed for the program, which occurs just before the generation of the assembly code.

Consider for a while, a single loop construct where induction variable i is linearly updated by the integer quantity *step* (i.e. the loop step), and $step \neq 0$. Figure 4.3(a) is an example of such a loop, where $step = 1$ and the loop *trip-count*[3] is N. The majority of DSP programs use well-defined loop constructs like the one just described. The main reason for that is the sequential processing nature of these programs, as discussed in Chapter 1.

```
for (i = 0; i < N; i++)
  array [a * i + b] = 0;
```

(a)

```
for (i = 0; i < N; i++)
{
  OR1 ← a;
  AR1 ← array + b;
  [AR1 + OR1] ← 0;
}
```

(b)

```
OR1 ← a;                          (1)
AR1 ← array + b;                  (2)
for (i = 0; i < N; i++)           (3)
{                                 (4)
  [AR1 + OR1] ← 0;                (5)
}                                 (6)
```

(c)

Figure 4.3: (a) Array reference using an affine function for index; (b) Three-address form of the loop body after strength reduction; (c) Three-address form after code motion of invariant instructions.

[3]Trip-count of this loop is the quantity $|UpperBound - LowerBound|$.

Assume that the indexes of the array references within the loop are affine functions of the type $f(x) = a * x + b$, where a and b are integer quantities. This assumption is not a serious restriction, given that the indexes of the majority of the array accesses in DSP programs are functions of this type.

Consider now the code fragment of Figure 4.3(a). Figure 4.3(b) and (c) show equivalent three-address code for the loop body. Two common code optimizations were performed in order to convert the code from Figure 4.3(a) to Figure 4.3(c). They are *strength reduction* and *code motion* [23]. Strength reduction substitutes the multiplication $a * i$ by adding a to the address of the array element at each iteration. Register OR_1 is used for this task. First, OR_1 is loaded with a and at each iteration it is added into AR_1 (instruction (5) of Figure 4.3(b)), after the storage operation is completed[4]. Figure 4.3(b) shows the resulting loop body after this optimization has been performed, and the register AR_1 allocated to address the array element. After strength reduction is performed, code motion is used to move all loop independent statements (instructions (3) and (4) in Figure 4.3(b)) to the outside of the loop. This results in the code of Figure 4.3(c) which works as follows. Before the loop is executed, address register AR_1 points to the first array access in the loop body, corresponding to address $array+b$. Offset register OR_1 is then loaded with a, if $a \neq 1$. If $a = 1$ register OR_1 is not required and the auto-increment feature of the AGU can be directly used in AR_1. In fact, the main object of computation in a DSP application are digitized signals, which can be naturally represented using integer numbers that fit in a single DSP word, which makes auto-increment (decrement) the most used addressing mode in a DSP application.

When multidimensional array elements are present within nested loops, array accesses can usually be reduced to the simple unidimensional case with the help of induction variable elimination [23]. Consider, for example, the nested loops of

[4]As discussed in Chapter 2, the majority of the address operations in a DSP are post-modifiers.

```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      array [i][j] = 0;
```

(a)

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
  {
    AR1 ← array;
    [AR1++] ← 0;
  }
```

(b)

```
AR1 ← array;                    (1)
for (i = 0; i < N; i++)         (2)
{                               (3)
  for (j = 0; j < M; j++)       (4)
    [AR1++] ← 0;                (5)
}                               (6)
```

(c)

Figure 4.4: (a) Multidimensional array reference inside loop; (b) Three-address form of the loop body after strength reduction; (c) Three-address form of the loop body after code motion of invariant instructions.

Figure 4.4(a), where array a is laid down in memory in a row first format. Similar to the unidimensional case, strength reduction and code motion are used to optimize the addressing code. The three-address code, resulting after these optimizations are performed, is shown in Figure 4.4(b) and (c). The final code of Figure 4.4(c) executes as follows. For each iteration of the outer loop (induction variable i), the inner loop statement visits each element of row $a[i][j]$ by just post-incrementing AR_1 (instruction (5) of Figure 4.4(b)). After the inner loop is finished, AR_1 is already pointing to the first element of the next row (i.e. $a[i+1, 0]$) and thus, it is ready to iterate over the next row. Although this example is a simple case of multidimensional array reference, similar techniques can also be employed for complex array references as described in [23].

The goal of the above optimizations is to rewrite array references in the loop body,

such that they are made dependent on a single induction variable, corresponding to its closest nested loop. Once this is achieved, the task of detecting the use of certain addressing modes can start, and this is done by means of the *Indexing Graph*.

The Indexing Graph

The ability of a program to make usage of the AGU features can be measured by the *Indexing Graph* (IG). The goal of the IG is to identify opportunities for array accesses to use the address computation modes available in the processor. Once these opportunities are found, the IG can be employed to allocate an AR to each array access. Consider the loop construct of Figure 4.5(a). Assume in the following analysis, that the target *AGU* provides only auto-increment (decrement) addressing modes. This is not a restrictive assumption for DSP applications, as mentioned before. When other modes are present, code optimization techniques can be employed, which allow the IG formulation to be used as well. The code fragment of Figure 4.5(a) is a typical example of a DSP application loop, and hence, it will be used in the remainder of this section. Each array reference will be defined from now on by its *array access*.

Definition 5 *Let $access(r) = n$ be the function which maps a reference r to an array element into n ($n = 1, 2, \ldots$), where n is the order of r in the code sequence resulting after the instructions in the loop body have been scheduled. We say that n is an access of the array element referenced by r.*

Definition 6 *Let n_1 and n_2 be array accesses. Access n_1 (n_2) is said smaller (larger) than n_2 (n_1), denoted by $n_1 < n_2$ ($n_1 > n_2$), if and only if n_1 (n_2) precedes n_2 (n_1) in schedule order.*

Example 7 Figure 4.5(b) represents the loop construct of Figure 4.5(a) after all instructions in the loop body have been scheduled. In order to simplify the analysis,

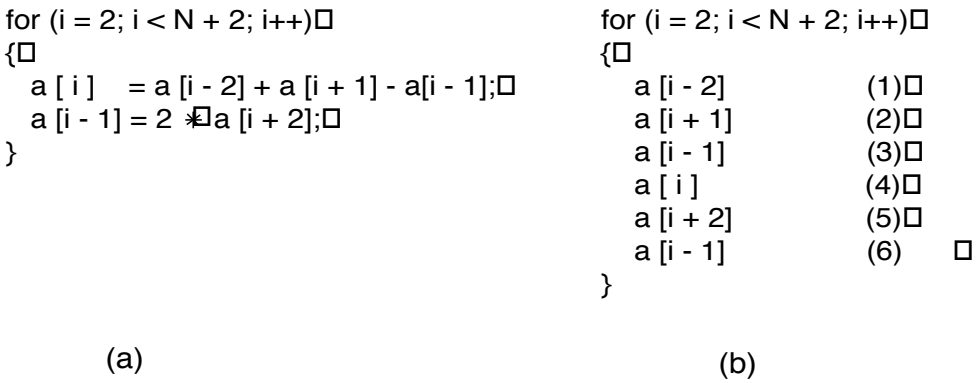


Figure 4.5: (a) Typical loop construct in a DSP application; (b) Array access sequence after scheduling the instructions in the loop body.

Figure 4.5(b) shows only the array references and not the instructions which make use of them[5]. Each time an array reference is used, a number (in parenthesis) is associated with it, corresponding to the order in which the reference occurs in the final schedule. This number is the access of that array element. For example $access(a[i+1]) = 2$.

The IG will be used to allocate *ARs* to each array access in the loop. Observe that the goal is to allocate the minimum number of ARs which can address all the array accesses within the loop. In this case, it is desirable to maximize the number of accesses that can share a single AR. In order to identify the possibility of sharing between two accesses, the concept of *indexing distance* is introduced.

Definition 7 *Let n_1 and n_2 be array accesses and step the increment of the loop containing these accesses. Let $index(n)$ be a function which takes access n and returns the index associated with that access. The indexing distance between accesses n_1 and n_2 is the positive quantity:*

$$d(n_1, n_2) = \begin{cases} |index(n_2) - index(n_1)| & \text{if } n_1 < n_2 \\ |index(n_2) - index(n_1) + step| & \text{if } n_1 > n_2. \end{cases}$$

[5]This representation will be used many times throughout this section.

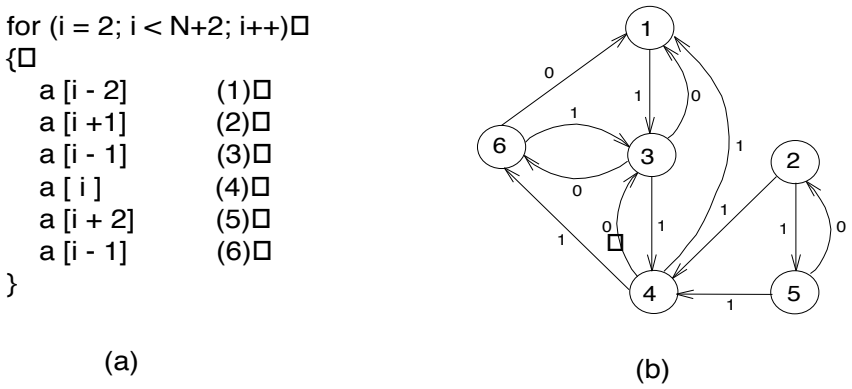


Figure 4.6: (a) Typical loop construct in DSP programs; (b) Corresponding IG.

Example 8 Consider for example the array accesses of Figure 4.6(a). In this case, as in the majority of loops in DSP programs, *step* = 1. The indexing distance $d(1,4) = |i - (i-2)| = 2$. Since the indexing distance is larger than one, no auto-increment (decrement) operation can be used to update the address register allocated to access (1), such that it ends up pointing to the data requested by access (4). On the other hand, since $d(4,1) = 1$, an auto-decrement operation can be used to redirect the address register associated to access (4) such that it points to access (1). Notice that this will occur when access (1) is reached from access (4) across consecutive loop iterations.

Now that array accesses are identified and a metric to measure their usage of the AGU is specified, the *Indexing Graph* (IG) can then be defined.

Definition 8 *An Indexing Graph (IG) is a directed graph where each node corresponds to an array access, and there exists an edge* (n_1, n_2) *if and only if* $d(n_1, n_2) \leq |step_{AGU}|$*, where* $step_{AGU}$ *is a post-modifier step in the processor AGU.*

There exists an edge (n_1, n_2) in the IG when AGU operations can be used to update the address register associated to access n_1, such that it can then point to the data associated to access n_2. The IG has two type of edges. An edge (n_1, n_2) is a

forward edge (or simply an *edge*) if $n_1 \leq n_2$, otherwise it is a *backward edge*.

Example 9 The IG of Figure 4.6(b) was built from the array accesses in the body of the loop of Figure 4.6(a). The IG is not a labeled graph, the labels on the edges in Figure 4.6(b) are only to illustrate the indexing distance from the source to the destination node of each edge. Observe that edges with indexing distance 1, like $(3, 4)$, capture the possibility for auto-increment (decrement) between array accesses in scheduling order. Backward edges, e.g. $(6, 1)$, identify auto-increment (decrement) operations which can be performed across loop iterations.

Array Index Allocation

Array Index Allocation is the problem of allocating virtual address registers to array accesses within loops, such that the total number of virtual address registers is minimized. The importance of this problem arises from the fact that the majority of array accesses in the DSP domain occur within loops which have linearly updated induction variables, and for which the *step* can be statically defined at compiling time. In order to formalize the array index allocation problem, the concepts of IG *path* and *cycle* have to be defined.

Definition 9 *A path $n_i \rightarrow n_j$ in the IG is a sequence of distinct array accesses $(n_i, n_{i+1}, \ldots, n_j)$, such that (n_k, n_{k+1}) is an edge in the IG and $n_k < n_{k+1}$, for $i \leq k \leq j-1$ and $i, j = 1, 2, \ldots$.*

Definition 10 *A cycle in the IG is a sequence of nodes $(n_i, n_{i+1}, \ldots, n_j, n_i)$ such that subsequence $(n_i, n_{i+1}, \ldots, n_j)$ forms a path in the IG, where $i, j = 1, 2, \ldots$.*

A path in the IG corresponds to the allocation of the same AR to a sequence of array accesses which can use the AGU addressing modes. Similarly, a cycle indicates

that the same AR can be used not only for accesses in program order, but also for one more access in the next loop iteration.

Definition 11 *A cover C of Indexing Graph G is a set of disjoint paths and/or cycles from G such that for all nodes $n \in G$, $n \in c_i$, c_i an element of C. The cover is disjoint, meaning that for every pair of elements c_1 and $c_2 \in C$, $c_1 \cap c_2 = 0$.*

The problem of minimizing the number of virtual address registers given an IG can be formulated as the following graph optimization problem.

[**IG Covering**] *Given an IG, determine the disjoint* **path/cycle** *cover of the graph which minimizes the total number of paths and cycles, and which has the* **smallest number of paths**. *Assume for the purpose of this problem that a node is a degenerated cycle of zero length.*

Notice that not all cycles are allowed in the definition above. According to Definition 10 an IG cycle can only contain a single backward edge. The reason for this is that cycles should allow auto-increment (decrement) operations across a single iteration and not across multiple iterations, reflecting what occurs during the loop execution. Observe that each path and cycle in the IG cover corresponds to an address register. Moreover, unlike the AR associated to a path, an AR associated to a cycle allows auto-increment (decrement) operations to occur across consecutive loop iterations. Notice also that an IG node is considered a cycle of zero length. Hence, a cover can contain elements which are single IG nodes. In this case the auto-increment operation occurring at that access updates the AR to be used by the same access in the next iteration.

Example 10 Consider the IG showed in Figure 4.7(b). Covering the IG in that case produces a two cycle cover (and no paths) which is represented in bold in Figure 4.7(b). Each cycle corresponds to a virtual address register (AR_0 and AR_1).

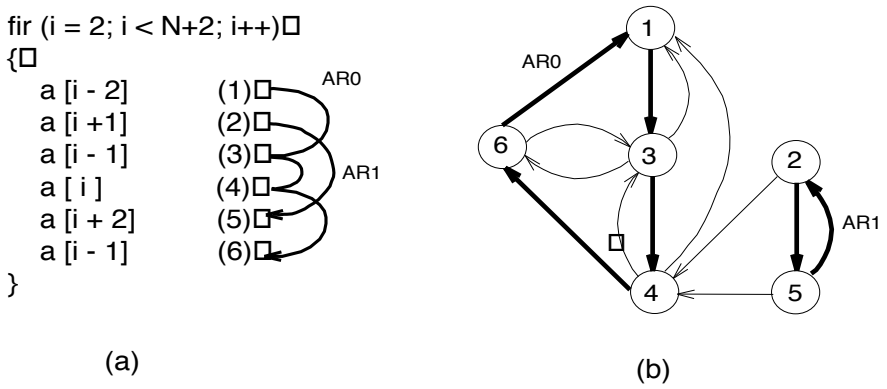


Figure 4.7: (a) Typical loop construct in DSP programs; (b) Corresponding covered IG.

Observe that the allocation of register AR_1 to cycle (2,5,2) takes care of the case of updating the address of the array reference (2) across loop iterations. The same is also true for register AR_0. The allocation of ARs across consecutive loop iterations was analyzed before, when register AR_y was used to access $b[i]$ across consecutive loop iterations (instruction (1), Table 4.1).

Observe that the solution for the IG Covering problem aims to find a cover which has the smallest number of paths (or the largest number of cycles) from all the covers which have the same (minimum) cardinality. In order to understand why, let the solution of the IG Covering problem be such that a path p results. Let $head(p)$ and $tail(p)$ be the head and the tail of p. Now assume that the indexing distance from the tail to the head of p, i.e. $d(head(p), tail(p))$, is larger than one. Therefore, neither auto-increment nor auto-decrement can be used to redirect AR from tail to head. In this case, an instruction will have to be issued in order to explicitly update the contents of the AR at the tail of p, such that it ends up pointing to the head of p.

Consider, for example, the program fragment of Figure 4.8(a), and its covered IG in Figure 4.8(c). Since the AGU provides only auto-increment (decrement) mode, and $d(3,1) = |(i+2) - (i) + 1| = 3 > 1$, then no edge exists from (3) to (1). The

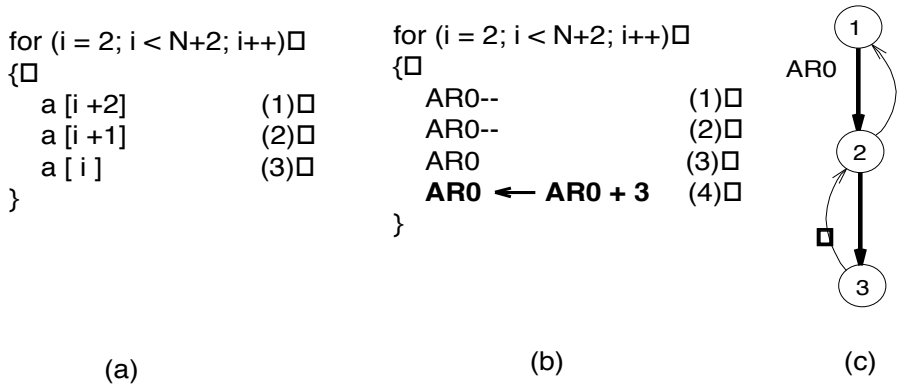


Figure 4.8: (a) The original array access sequence; (b) If only auto-increment (decrement) is allowed for the access, then we need to add 3 to redirect AR_0 to point to the first access in the next iteration; (c) The corresponding covered IG.

IG cover results in a single path $(1, 2, 3)$, which is allocated to virtual AR_0. At the end of the loop AR_0 has to be redirected to point to $a[i + 3]$ in the next iteration. But since no edge exists from (3) to (1) then an instruction (4) has to be added at the bottom of Figure 4.8(b) to explicitly do that. Therefore, the solution of the IG Covering problem should be restricted to those covers which have the smallest cardinality, in order to guarantee that the minimum number ARs are used. From all possible candidates the solution is a cover which has the smallest number of paths, since this minimizes the number of instructions required to update the ARs at the tail of the paths.

The IG Covering problem is similar to the minimum disjoint cycle cover of a graph (MDCC). The number of disjoint cycles which cover the nodes of a graph is known as the *Hamiltonian cycle index*. Determining the minimum Hamiltonian cycle index of a graph has been shown to be NP-complete [49]. Cycles in a cover for the MDCC problem, unlike cycles for the IG Covering, can contain more than one backward edge. Therefore, it is expected that MDCC is NP-hard, since it is as difficult as determining the minimum Hamiltonian cycle index.

The Simple IG Covering

Given that IG Covering is NP-hard, it is important to consider heuristics to tackle this problem. The most obvious one is to formulate the problem such that auto-increment (decrement) operations across loop iterations are not permitted, as in Figure 4.9. As a result of that, the IG becomes acyclic, and the original problem is reduced to the one of determining the minimum vertex-disjoint path covering of the graph (MDPC). Based on that one can now formulate a simple version of the IG Covering problem.

[**Simple IG Covering**] *Given an IG determine the disjoint* **path** *cover of the graph which* **minimizes the total number of paths**. *Assume for the purpose of this problem that a node is a degenerated cycle of zero length.*

The MDPC problem has been studied before by Boesch and Gimpel in [78]. The solution they propose is based on the Hopcroft-Karp $O(n^{5/2})$ solution of the bipartite matching problem [79]. The main idea is to divide each node n of graph G into two nodes n_1 and n_2. Node n_1 (n_2) is the source (destination) of all outgoing (incoming) edges from (into) n. By doing so, the resulting graph G' becomes bipartite, with n_1 and n_2 belonging to disjoint sets. When the bipartite matching algorithm is used in G', by definition, no node n in G' will have more than one outgoing (incoming) edge. Therefore, the matched edges in G' form a cover in G, which has only disjoint paths. Assume here that a node is a degenerated path of length zero. During the minimum bipartite match all edges have the same weight. In this case, the algorithm results in the minimum number of disjoint paths capable of covering G.

Example 11 Solving the MDPC for the acyclic IG of Figure 4.9 results in paths $(1, 3, 4, 6)$ and $(2, 5)$. Cycles can still be identified in this case by computing the indexing distance between the tail and the head of a path. For example, since $d(6, 1) = 0 \leq 1$ ($d(5, 2) = 0 \leq 1$), then virtual register AR_0 (AR_1) can be used, at the tail

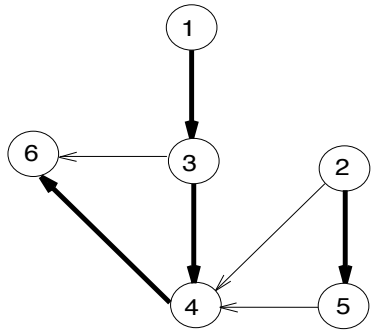


Figure 4.9: Solving the MDPC for an acyclic IG.

of its corresponding path, to point to the data accessed at the head of the path. In this case, the heuristic approach produces the same result as the exact solution in Example 10.

The solution of the Simple IG Covering problem, for the graph of Figure 4.9, was the same as the exact solution for the IG covering problem for this graph. This might not always be the case though. Nevertheless, the experiments in Section 4.6 demonstrate that Simple IG Covering produces effective improvement, and hence the need for a more expensive solution might not compensate the computational effort.

Dealing with If-then-else statements

In the previous sections it was assumed that the loop body contains a single basic block. Consider now that **if-then-else** statements are allowed within the loop, like in the program fragment of Figure 4.10(a). Observe that array accesses $a[i]$, $a[i+1]$ and $a[i-1]$ are located in three different basic blocks as shown in Figure 4.10(b). The basic block which contains access $a[i]$ (B_1) is always executed. On the other hand, basic block B_3 (B_5), containing access $a[i+1]$ ($a[i-1]$), is only executed if variable *test* is larger (smaller/equal) than zero.

The corresponding indexing graph (without backward edges) has three nodes (1,3

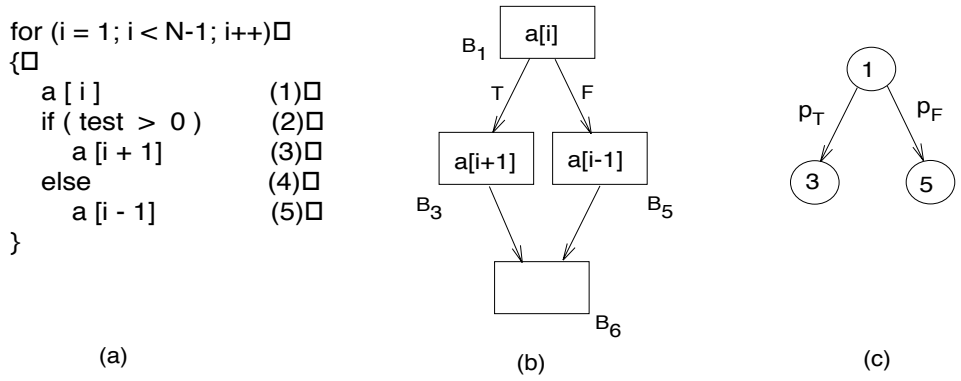


Figure 4.10: (a) Loop body containing **if-then-else**; (b) The CFG representation; (c) The corresponding IG with path probabilities.

and 5), one for each array access, and two edges $(1, 3)$ and $(1, 5)$, as shown in Figure 4.10(c). Applying IG covering to it results in one of the two edges being covered. Assume that virtual address register AR_1 is allocated to the covered edge. Therefore, AR_1 will be used for access (1). If the covered edge is $(1, 3)$ ($(1, 5)$), then AR_1 will also be used for access (3) ((5)). During access of $a[i]$ AR_1 is incremented (decremented) in order to enable AR_1 to point to $a[i + 1]$ ($a[i - 1]$) when the next basic block is executed. The IG covering algorithm has no way to distinguish which edge from these two will be covered. Nevertheless, it is reasonable to believe that the edge associated to the most frequently executed path should have higher priority during the covering. The reason is that this increases the probability that a frequently executed path will have its addressing operations allocated to an addressing register. In order to capture that, the IG covering problem can be modified, such that IG edges associated to paths in the program are annotated with the probability of the path being taken. For example, in Figure 4.10 the IG edge associated to the loop condition being true (false) is labeled with the probability p_T (p_F) of the path through B_3 (B_5) being taken. The covering algorithm is then modified, such that the list of IG edges selected for the cover are sorted in decreasing probability order. Hence, frequently exercised paths have a higher probability of being covered.

```
for ( i = 0; i <= LENGTH; i++)      (1)
{                                   (2)
    *px + + = i;                    (3)
    *ph + + = i;                    (4)
}                                   (5)
```

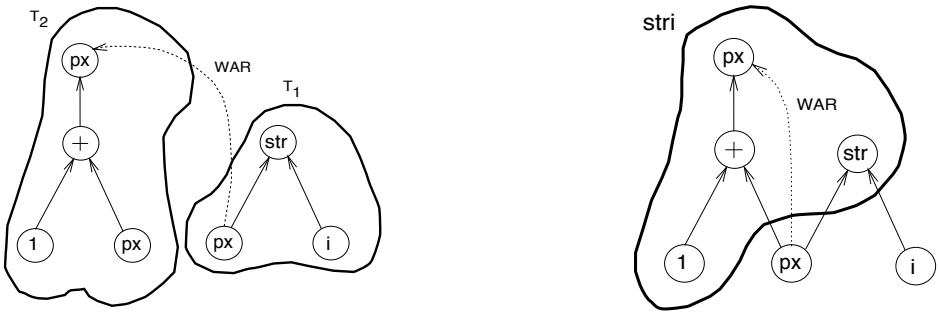
Figure 4.11: Part of the fir.c DSPstone benchmark kernel.

4.4.2 Pointer Variable Allocation

Using pointer variables to access data-stream elements is a common operation in DSP programs. Consider for example the program fragment in Figure 4.11. In this program fragment, pointer variables px and ph are used to initialize the contents of an array. Consider the expression DAG generated from statement (3). By using the tree-based code generation techniques described in Chapter 3, one can dismantle this DAG into expression trees using two different approaches, as shown in Figure 4.12(a) and (b). There, nodes labeled px are used to represent operations *read* px and *write* px, and node *str* takes the value contained at memory position i and stores it at the memory position pointed by px. A Write After Read (WAR) constraint edge is used to enforce the original post-increment behavior in the source program.

Assume as before, for the sake of simplicity, that px points to an object of the size of a machine word. It will be shown later how this can be generalized. Two approaches can be used to compute the new address of the object pointed by px.

In the first approach (Figure 4.12(a)), tree T_1 contains the operations used to perform the increment of px, and T_2 contains those required for copying variable i. The three-address representation of the instructions generated for these operations is also shown in Figure 4.12(a). Registers AR_1 and R are respectively used to store variables p_x and i. Assume, for the sake of simplicity, that each instruction takes a



T_1:
$AR_1 \leftarrow px$
$R \leftarrow i$
$*IR_1 \leftarrow R$
T_2:
$AR_1 \leftarrow px$
$AR_1 \leftarrow AR_1 + 1$
$px \leftarrow AR_1$

(a)

T_1:
$AR_1 \leftarrow px$
$R \leftarrow i$
$*IR_1 ++ \leftarrow R$
$px \leftarrow AR_1$

(b)

Figure 4.12: (a) Dismantling the expression DAG into trees; (b) Pattern matching an auto-increment operation.

single machine cycle to execute. The total cost of the instructions corresponding to trees T_1 and T_2 is six instructions. In another approach, operations *str* and increment p_x are pattern matched by a new instruction *stri* (store and increment), as shown in Figure 4.12(b). Notice that after the pattern matching operation occurs, the new instruction *stri* will have the same operands as instruction *str*. The difference is that *stri* will increment the variable *px* after the storage operation is performed. Therefore, by allocating virtual address register AR_1 to variable *px* one can make efficient usage of the auto-increment feature of the processor AGU. The total cost of the resulting three-address instructions is four instructions, which is smaller than the cost of the previous approach.

Observe that this technique can also be used for the case when the size of the pointed object is larger than a machine word. Consider for example that *px* is a

pointer to an object of size S. Assume that the processor AGU has offset register OR and address register AR, and that its instructions allow for offset addressing (see Chapter 2). Hence, AR can be loaded with the address of the first object and OR with its size S. Each $px++$ operation is then compiled into the *load/store* post-increment operation $[AR + OR]$, and every time an element is pointed by px, the address of the next element is automatically computed.

4.5 Physical Address Register Allocation

The previous section proposed techniques to allocate virtual ARs to array references and pointer variables. The goal there was to increase the usage of addressing modes available in the architecture, and to minimize the number of virtual ARs required by indirect addressing instructions.

The next task in code generation for address computation is to allocate the physical ARs available in the processor to the virtual ARs assigned before. The goal here is to minimize the usage of physical ARs. As an immediate consequence of that, pointer variables and array accesses will have a higher probability of using an address register for its auto-increment (decrement) operations, reducing the cost of memory spilling, and increasing the possibility of these operations being performed in parallel with datapath operations.

Register allocation is a well studied problem in compiler technology. The standard technique to solve this problem, known as *register coloring*, was proposed by Chaitin [47] and refined in [80, 81]. Further improvements on this technique have been recently proposed in [82]. The idea behind register coloring is to build an interference graph where each node represents a virtual register, and an edge (r_1, r_2) between virtual registers r_1 and r_2 indicates that r_1 and r_2 have intersecting lifetimes and therefore, should be allocated to different physical address registers. The problem is modeled as

a graph coloring problem, where no two nodes on the same edge can have the same color, and where a color is associated to a physical register in the processor. The number of colors available to color the nodes of the graph corresponds to the number of physical registers present in the processor. Chaitin's algorithm was used in this work to perform the allocation of physical ARs to virtual ARs.

Although the numbers indicate that the use of Chaitin's algorithm results in improved code (see Section 4.6), the existence of certain features in DSP programs seems to suggest that more attention should be given to loop statements during the allocation process. As discussed before in Chapter 1, DSP programs make extensive use of array access inside loop statements. Hence, it is reasonable to believe that the majority of the data accesses through address registers will occur within loop constructs. In this case, an approach which favors allocating registers to inner loops will certainly result in better code performance, given that inner loop code executes a larger number of times than outer loop code. The Callahan-Koblenz algorithm [48] is based on this reasoning. The idea behind the Callahan-Koblenz algorithm is a hierarchical register coloring which gives higher priority to allocate registers to inner loop variables. The main goal is to decrease the probability of spilling operations to occur inside inner loops, where the performance penalty would be proportionally larger than in outer loops. The use of Callahan-Koblenz for physical AR allocation is part of the future extensions of the SPAM compiler [26].

4.6 Experimental Results

The techniques for array index and pointer variable allocation discussed in the previous sections were tested here using kernels from the DSPstone benchmark. As mentioned before (Chapter 3), DSPstone is a well-known benchmark for DSP applications. The fact that it is based almost exclusively on kernels (unlike *SPECInt'95*)

reflects a characteristic of the DSP domain, in which kernels play an important role, and large applications are either proprietary or simulation oriented (floating-point based[6]). Other DSP benchmarks, like the *BDT DSP Benchmark* are also kernel oriented.

4.6.1 Array Index Allocation

DSP kernels from the DSPstone benchmark have been used to evaluate the improvement gained by array index allocation. Precisely due to the lack of effective compiler optimizations for this problem (like the one proposed in Section A.3.1), DSP programs have being designed such that array references are transformed into pointer operations. This was also the case for the DSP kernels available in DSPstone. Hence, in order to use DSPstone as an experimental framework, its kernels had to be rewritten into their original array style[7]. The resulting kernels were compiled using the *Texas Instruments TMS320C25 Optimizing C Compiler*, which is considered the best optimizing compiler for this processor available today (circa 1997). The relevant optimization flags of the compiler were active during compilation. The kernels were also compiled using the array index allocation technique from Section A.3.1, which was implemented inside the SPAM compiler.

The size and cycle count of the resulting code was measured for each compiler and listed in Table 4.2. The average improvement in code size and cycle count for the SPAM compiler compared to the TI compiler, were respectively 13% and 28%. The main reason why the TI compiler was consistently outperformed was due to its inability to efficiently allocate address registers to array references. It is common to find, in the TI code, recomputed address operations which could have been shared between consecutive arrays accesses. Notice that the possibility of sharing address

[6]The TMS320C25 is a fixed-point processor.

[7]Only those kernels which iterate over array elements have been included.

Item	Kernel Name	Code size		Cycle count	
		TI	SPAM	TI	SPAM
1	biquad_N	118	107	528	425
2	complex_update	85	76	121	108
3	convolution	65	56	289	273
4	dot_product	52	43	107	81
5	fir	62	56	428	379
6	fir2dim	245	221	4035	2802
7	lms	121	105	763	584
8	matrix1	125	114	17897	14924
9	matrix_1x3	46	35	118	79
10	n_real_updates	101	82	711	627
11	n_complex_updates	139	122	1603	1165

Table 4.2: Experiments with array index allocation. Programs were converted to use array references instead of pointers.

operations is captured by the IG covering algorithm in the SPAM compiler. On the other hand, the big difference in code quality between the two compilers cannot be explained only based on the use of array index allocation. The way the SPAM compiler handles stacks plays a very important role as well. The SPAM compiler tries to use static allocation of local variables as much as possible, unlike the TI compiler, which always allocates local variables to the stack. The approach to the allocation problem used by the TMS320C25 compiler misses an important feature frequently found in DSP programs. DSP applications are well defined programs which almost never use recursion, unlike general-purpose applications. Therefore, since the resulting *callgraph* is typically acyclic, variables can be statically allocated and the stack can be eliminated. As a consequence, the TI compiler needs three ARs to manage the stack, and since the TMS320C25 has 8 ARs, only five are left for addressing operations. Moreover, address registers are allocated *on the fly* and not based on its liveness, which contributes to an increased probability of spilling.

Item	Kernel Name	Code size			Cycle count		
		Unopt.	Opt.	TI	Unopt.	Opt.	TI
1	biquad_N	132	105	110	535	420	511
2	complex_update	97	74	71	156	102	89
3	convolution	68	56	59	465	373	279
4	dot_product	51	41	46	238	68	76
5	fir	74	62	79	734	392	408
6	fir2dim	214	216	199	4905	2799	2689
7	lms	121	100	111	1025	584	544
8	matrix1	160	113	106	20137	14923	14507
9	matrix_1x3	40	30	32	195	73	75
10	n_real_updates	113	80	73	813	613	575
11	n_complex_updates	191	123	135	1965	1153	1032

Table 4.3: Using pointer update optimization.

4.6.2 Pointer Variable Allocation

The approach described in Section 4.4.2 was used to compile a set of kernel programs from the DSPstone benchmark suite. The results are listed in Table 4.3. The metrics used to measure the code quality are code size and cycle count. Columns *Code Size Unopt.* (*Cycle Count Unopt.*) in Table 4.3 shows the number of words (cycles) after compiling the program using no address register optimization techniques. In column *Code Size Opt.* (*Cycle Count Opt.*) one can find the final assembly code size (number of cycles) when the approach described in Section 4.4.2 was used. In all programs, the address computation associated with the auto-increment (decrement) operations were compacted into datapath instructions. Column *Code size TI* (*Cycle Count TI*) lists the size and cycle count when the kernel is compiled using the TI compiler with the appropriate flags set. The SPAM compiler resulted in better code, considering size and cycle count, for kernels 1, 3, 4, 5 and 9. Observe that in some cases (kernels 7 and 11) although the code size for the SPAM compiler was smaller than the one produced by the TI compiler, the number of cycles was larger. The

reason for this is the performance of the SPAM compiler for some instructions inside a loop body which was worse than for other instructions outside the loop. The average improvement compared with unoptimized code was 22% in code size and 38% in number of cycles. Observe that this optimization **is not** new for compilers of architectures containing auto-increment (decrement) addressing modes, like the VAX PDP 11/780, the Motorola 68000, the TMS320C25 and others.

Chapter 5

Conclusions

Computers have been around for fifty years, and during this time compilers have evolved from basic translation tools, like assemblers, to sophisticated optimizing compilers. The early compiler research in the 70's addressed the basic problems in code generation for straight line code, by providing algorithms to improve code size during a time when memory size was a big constraint in the design of a program. The late 70's and early 80's witnessed the development of global data-flow analysis and a number of optimization techniques. The late 80's and first half of the 90's were marked by the emergence of an unified architectural model (RISC), and by new compiling techniques based on this model. The RISC architecture was the result of two decades of research in architecture and compiler design. The idea behind RISC is simple: a combination of a regular architectural style with a simplified ISA, which can be efficiently exploited by an optimizing compiler. RISC is a general-purpose architecture, i.e. an architecture devoted to run a broad range of different applications. Besides better performance for generic programs, there are also economic reasons to support the general-purpose architectural style, instead of a customized architectural solution. A high-performance general-purpose processor, which can be used by many applications, can easily amortize its design and manufacturing costs through the production of a large number of units.

Unfortunately, general-purpose architectures are not capable of meeting the constraints demanded by certain application domains. Applications which have stringent time, cost and low power constraints require specialized architecture processors to meet these requirements. Examples of these applications can be found in telecommunications and consumer electronics products like: cellular telephones, hand-held computers, video cameras, GPS units, etc. Although general-purpose architectures are sometimes capable of satisfying their performance requirements, they are far away from meeting the required levels of cost and power consumption.

The small size of the programs running on specialized processors has enabled applications to be programmed in assembly language for a long time. With the recent increase in the size and complexity of these the applications, this task has become cumbersome and prone to errors. As a consequence, the demand for programming in high-level languages is growing fast, and so is the need for efficient compilers and other supporting tools for these processors.

Specialized processors have dedicated architectural features, which require novel compiling techniques. The goal of the SPAM project is to provide this technology, and the work developed in this thesis is a contribution to this goal. This thesis studied the problems of basic block and addressing code generation for a subset of specialized processors known as DSPs. DSP architectures demand high quality code generation, i.e small code size and high performance. The route followed by this thesis starts at the early research work in code generation done during the '70s. The idea was to re-examine the basic problems in the area from the perspective of a new application domain, and to provide solutions for the problems encountered in this domain.

5.1 Thesis Contributions

There are two major contributions of this thesis. The first is an architectural model called *Register Transfer Graph* (RTG), which can capture the important features of the processor ISA. The RTG permits various architectural styles to be modeled using a unique structural representation. Based on the RTG, the processor ISA can be analyzed in terms of the interaction between the datapath topology and its relation to the code generation problem. The ISA of various DSPs were classified according to the existence of RTG cycles. Based on that, an acyclic ISA was defined to be an architecture for which the RTG has no cycles. An important fact emerged from this observation: an optimal linear-time code generation algorithm exists for expression trees, when the ISA is acyclic. In particular this resulted in the first such algorithm for a DSP processor. The RTG model also provided the insights which lead to a heuristic for dismantling basic block expression DAGs using architectural information.

The second major contribution of this thesis is an algorithm for the allocation of address registers to array references in DSP programs. The solution for this problem, proposed in Chapter 4, is an approach which eliminates the need for DSP programmers to rewrite array references into pointer arithmetic, while retaining the code quality achieved when doing so. This approach enables address operations to be shared between consecutive array accesses, even when they occur across consecutive loop iterations. This technique can be combined with address register allocation for pointers, resulting in a coherent approach for the generation of addressing code for DSPs.

5.2 Future Work

The RTG architectural model opens up new avenues for the exploration of the interaction between application, datapath design and code generation. For example, the role of cycles in RTG, and its impact in code generation, gives the processor designer a guideline that can help in the design of new architectures for a specific application domain. An analysis of the expression trees found in the program exposes the designer to the paths in the datapath which are exercised during the execution of the program. Based on that, only paths which are intensively used by the application need to be introduced, which minimizes design cost and might reduce the number of expression trees which have allocation deadlocks[1]

Observe that if the target ISA is cyclic, this does not mean that all expression trees in the program will have allocation deadlocks. In this case, a scheduling heuristic based on allocation deadlock avoidance could produce very good results. Verifying this assumption is an interesting continuation of the work developed in Chapter 3.

Another approach to handle the case of cyclic ISA might be possible by determining the required size of the register files in the architecture. The solution for this problem is particularly important for the design process of *in-house* processors, where the architecture and the compiler are designed at the same time. The key idea here is to determine the size of the register files based on the cycles of the RTG, such that no expression tree in the program had allocation deadlocks. Techniques that could guarantee this would have a major impact in the design of in-house processors.

The RTG Theorem in Chapter 3 brings to light another interesting research problem. The RTG Theorem shows that any architecture which has an acyclic ISA, must have a linear-time optimal code generation algorithm. Although this is only a sufficient condition, it opens the possibility for the existence of the following conjecture:

[1]Notice that expression trees which have no allocation deadlocks will have spill-free schedules.

Conjecture 1 *An architecture has a polynomial time expression tree code generation algorithm iff its ISA is acyclic.*

A proof for Conjecture 1 would eventually solve the problem of basic block code generation for specialized processors in general, since it implies that code generation for arbitrarily complex datapath topologies is an NP-complete problem.

Another possible extension of this work comes from the formulation of the Array Index Allocation problem. As discussed in Chapter 4, Array Index Allocation can have a more general formulation based on covering the IG using cycles and paths while minimizing the number of paths. A solution for this general formulation was not considered thoroughly. The problem of covering a graph with cycles has been studied before in other context, like in the *Chinese Postman Problem* [83]. The solution for this related problem may provide the heuristic for the solution of the generalized form of Array Index Allocation, what may result in improved code generation.

The major goal of this work was to bring code generation for DSPs into mainstream compiler technology, by addressing the basic problems and providing solutions to them.

Appendix A

The Code Generation Interface

The goal of this appendix is to describe the programming interface used in the design of the code generation modules for the SPAM compiler. It assumes some familiarity with compilers, code-generator generators, and is intended for compiler developers. The information available here was used in the design of a code-generator for the TMS320C25 DSP processor, but it can also be used in the development of code generators for other DSP processors.

This appendix is divided into four sections. Section A.1 describes Olive, the machine description language used in SPAM. Section A.2 lists the functions available in the SPAM register allocator and Section A.3 the functions used to perform virtual address register allocation. Section A.4 describes how to use SPAM to dismantle the basic block DAG using architectural information. Finally, Section A.5 shows examples of two partial Olive files, one for a typical RISC processor and another for a DSP processor.

A.1 The Olive Description Language

The SPAM compiler uses *Olive* [70] as its machine description language. Olive was designed by Steve Tjiang, and is based on *Twig* [68] and *iburg* [69]. iburg is a code-generator generator developed by David Hanson and Christopher Fraser. Twig is a code-generator generator designed by Steve Tjiang, A.V. Aho and M. Ganapathi.

Olive is not a machine description language, in the strict sense of the word, given that it only supports structural description of the target processor *Instruction Set Architecture* (ISA). Olive does not capture microarchitectural information, for example pipeline timing. Issues associated with that, like the need to fill in *load delay slots*, have to be addressed by the developer separately during the design of the new compiler. On the other hand, Olive provides the developer with a set of powerful code generation algorithms, and most of all, it defines a methodology which is very helpful in the design of the compiler code generator module.

The following sections give a broad overview of Olive, and describe how to use Olive functions in the design of the processor Olive file. Olive takes this file as input to generate a code-generator for the target processor. The resulting code-generator reads an Olive expression tree and returns machine code. Section A.1 describes the history and the advantages of using Olive when compared with other code-generator generators. Sections A.1.1 – A.1.4 describe the syntax of Olive and its embedded functions. These sections have been adapted from the document *An Olive Twig* [70].

Overview

This section describes Olive, a new version of Twig based on iburg. Twig is a tree-manipulation system intended primarily for code generation. Twig can be used to build a code generator that takes expression trees as input and emits assembly code. Andrew Appel used Twig to build several code generators in his compiler class at

Princeton [70].

Though intended for code generation, Twig's flexibility also suits other applications. Kurt Keutzer used Twig to build Dagon, a logic synthesis system used at Bell Labs. Twig's flexibility arises from dynamic programming and the generality of its cost computations, which can be practically any block of C. Dynamic programming offers a powerful method to choose between many different alternatives that arise during code generation and technology-mapping. General cost computations permitted users to specify fairly sophisticated tests. Moreover, general cost computations allow dynamic programming to be performed even with non-integer costs.

Another key feature of Twig is the ease with which tree manipulations can be interleaved with pattern-matching. Twig's pattern-action rules can rewrite subtrees that can participate in further matching.

Advantages of iburg

Unfortunately, Twig's generality hurts its pattern-matching performance. Because costs are opaque to the Twig compiler, powerful table-generation techniques such as BURS theory [84] cannot be applied. A general cost function compels Twig to perform all dynamic programming during run-time instead of at table-generation time, saddling Twig with slow run-time performance. For example, in a VAX code generator designed with Twig [68], the pattern matching accounted for over 80% of the execution time.

Several new tree-matching systems – burg, iburg – have capitalized on these deficiencies. Burg restricts costs to integers and applies BURS theory to create very fast pattern-matchers. Unfortunately, the matchers are difficult to debug as the details are buried in inscrutable tables. To make debugging easier, David Hanson designed iburg for use in his compiler class at Princeton [70]. The matcher-generator – iburg – has done away with tables, instead implementing the pattern matching directly with

IF and **SWITCH** statements.

What New Olive Offers

Olive is Steve Tjiang's attempt to update Twig so that it becomes a better tool for tree-matching. Here are the innovations in Olive:

1 *A richer specification language:* Recent systems like burg and iburg have shied away from more powerful specification languages. The developers of burg and iburg intended them to be used as a back-end for tree-manipulation systems that do offer richer specification languages [70]. While the developers' policy does make burg and iburg more generally applicable, it reduces their value as pre-packaged solutions [70]. The aim of Olive is to ease the flow of information between rules. This flow could not be specified directly in Twig. Olive treats rules more like functions with arguments and offers easier ways for one rule to invoke another; both of these features are missing from Twig. In Olive, communication between rules can occur only through global variables or special fields in the nodes of the AST.

2 *Faster pattern matching:* Olive takes advantage of the open-coding style of iburg, i.e. code the table interpretation directly as **IF** and **SWITCH** statements, minimizing the overhead of the pattern-matching.

3 *Generalized costs:* Olive maintains general costs because of their usefulness as a means to write predicates. Although this will cost somewhat in performance, that impact will be much less than for Twig, and be more similar to that of iburg.

The following sections describe the Olive language, show how to interface to the pattern-matcher, and outline the code that Olive generates.

A.1.1 Lexical Conventions

Identifiers are defined in Olive as strings of letters, digits or underscores, starting with either a letter or a percent sign (%). All Olive keywords begin with a percent sign. Olive uses the characters "; :() , =" as punctuations.

Like YACC grammar productions [85], Olive rules contain embedded C code that are to be integrated into the pattern matcher. The C code can come in two forms. It can be a correctly parenthesized C expression or it can be brace-enclosed blocks of C code as in YACC. Within the embedded code, C lexical conventions rule with the exception of special constructions that begin with a dollar sign (**$**), which are constructs with special meaning to Olive. They will be explained later in Section A.1.2.

A.1.2 Rules and Tree Patterns

In the following, EBNF is used to describe the Olive language. A **nonterm** in tree-

rule	→	**nonterm : tree [cost] action;**
tree	→	**term (tree_list)**
	|	**term**
	|	**nonterm**
tree-list	→	**tree_list, child**
	|	**child**
child	→	**tree**
	|	**_**
cost	→	**C-code**
	|	**C-expr**
action	→	**C-code**

patterns corresponds to non-terminals in the grammar. A **term** corresponds to terminal symbols. For tree-patterns, **term** symbols label tree nodes. The special identifier "**_**" denotes a *don't care* situation which is used to bypass pattern-matching for particular subtrees. A rule matches a tree if and only if two things happen:

1 The tree pattern must match the subtree. *Matching* can be defined recursively. For a tree to match the pattern

$$x\ (t_1, t_2, \ldots, t_n)$$

the root has symbol x; there are n subtrees and each subtree matches a corresponding t_i. The leftmost subtree matches t_1; the next leftmost matches t_2; and so on.

2 The **cost** part of the rule must return a finite cost.

Costs

The **cost** part computes the cost of the rule when the tree pattern of the rule matches. The **cost** part can be used as a predicate: it can return a zero cost to accept a match or it can return an infinite cost to force a mismatch.

The **cost** part of a rule is either a C expression or C code. If the **cost** part is a C expression the expression is evaluated. If it evaluates to true then the rule matches with cost zero, otherwise the rule fails to match. If the **cost** part is C code the code is executed. Olive expects the code to return the cost of the match in special variables that begin with a dollar sign (**$**), as described in the following section.

Actions and Prototypes

Every non-terminal symbol is associated with a set of functions. This set of functions have the same prototype – they return values of the same type; they have the same number of arguments and each corresponding element have identical types. The developer must supply a prototype for a non-terminal functional using a **%declare** statement.

The **code** and **action** parts of a rule can communicate with the pattern- matcher through special variables and functions whose names begin with a percent sign (`$`). The list below summarizes the conventions.

- `$n`

  The nth node in the tree pattern. Nodes in a tree pattern are numbered in the order in which they appear in a pre-order traversal. For example, in the rule

  ```
  L: A (B, C (D))
  ```

  the `A`, `B`, `C`, and `D` are numbered 1, 2, 3, and 4 respectively. The designator `$0` refers to `L`, left-hand-side of a rule.

- `$cost [n]`

  The cost of the nth node in the tree pattern. This can only be used inside the **cost** part of a rule. To return the cost, a rule should assign to `$cost [0]`.

- `$rule[n]`

  This function returns the **act** of the current rule. The **act** of a rule is an internal data structure that Olive uses to keep track of the non-terminal which was selected to match the current AST node. See action `$exec` below on how to use **act**.

- `$getcost[n]`

  The cost of the nth node in the tree pattern. This can only be used inside the **action** part of a rule[1].

- `$action[n](a, b, ...)`

  Explicitly perform the action part of the rule associated with the nth node in the tree pattern. This represents a call to the action part of the rule with arguments `a`, `b`, `c`, etc. The action part returns a value of the type specified in its prototype declaration. The action part of a rule can be executed more than once.

[1] In future versions of Olive, `$getcost` should be eliminated and `$cost[n]` used in the action part as well.

- `$immed[n,lhs](a, b, ...)`

  Explicitly perform the action part of the rule associated with the nth node in the tree pattern, which must be a "`_`". The action routine should have the same prototype as an action part of a rule whose left hand side is the non-terminal symbol `lhs`. Execution will fail if the n^{th} node does not match `lhs`. The action part of a rule can be executed more than once.

- `$exec[act](a, b, ...)`

  Explicitly perform the action part of the rule which matched the node described in **act**. The action routine should have the same prototype as the action part of the rule matched in **act**. `$exec` can be used to emit code in advance for those subtrees in which the root has been spilled into memory (see Section A.5.2).

- `$match [n, lhs]`

  Returns true if the n^{th} node matches `lhs`. The n^{th} node in the tree pattern must be a "`_`".

Declarations

Terminal and non-terminal symbols must be declared before they are used. A terminal symbol is declared using the `%term` statement. Each `%term` statement declares a list of identifiers as terminals. Olive assigns an index to each terminal symbol and creates a `#define` which can be used to refer to the terminal symbol. To declare a non-terminal symbol and to associate it with a prototype, the developer must use `%declare` statement. Each `%declare` statement looks like this:

`%declare<return type> nonterm<arguments>`

Note that angle brackets are used and not parenthesis. This is done to keep the Olive parser simple.

A.1.3 Interfacing to Olive

The user must provide two abstract data types to Olive, one for costs and one for tree nodes. This gives the user flexibility in choosing representations for costs and trees. If C++ is used then these could be classes. In C, however, they have to be provided using macro definitions and functions. This section lists the required definitions and functions for costs and trees.

Costs

Olive will support costs defined by the following components:

1 **COST** is a C type, defined either through a macro or a **typedef**.

2 **INFINITY** is the maximum attainable cost value.

3 **DEFAULT_COST** is the default cost returned by rules without a cost part.

4 **COST_LESS (x, y)** is a function that returns true if and only if the first argument costs less than the second.

The cost part of a rule set the costs for the rule by using the designator **$cost[0]**. It can also perform a **return 0;** explicitly to abort the consideration of that rule.

Trees

Olive uses the following definitions to access the trees.

1 **NODEPTR** is the type of a pointer to a node.

2 **NULL** denotes a non-existent node.

3 **GET_KIDS (r)** returns a vector of **NODE-POINTER**s that are the children of **r**.

4 **OP_LABEL (r)** returns the label of the node **r**.

5 **SET_STATE (r, s)** assigns the state label s to the node **r**.

6 **STATE_LABEL (r)** returns the state label previously assigned to the node **r**.

A.1.4 Invoking the Tree Matcher

Olive generates a routine called **burm_label** that is called to invoke the matcher, as shown below: The routine **burm_label** determines a covering for the tree, and may

```
if(burm_label(root) == 0)
     error ("matcher failed");
else
     lhs_action (STATE_LABEL (root), arguments);
```

invoke actions on parts of the tree due to immediate rules. If **burm_label** returns a non-zero value, covering was successful. If **lhs** is the non-terminal that matches at the root, one can execute the actions of the rule that form the cover by calling the routine **lhs_action**.

Olive Flags

The following are Olive command line flags:

- **-L**

  Turns *off* the emission of "**#line**" directives.

- **-I**

  Tells Olive to generate **burm_string**, **burm_files**, **burm_file_numbers**, and **burm_line_numbers** which are tables used to map rule numbers back to the file and line number where the rule is defined. These tables are used for debugging.

A.2 The Register Allocator

SPAM contains a number of modules employed to easy the design of new compilers. `TheRegAllocator` is the register allocation module of SPAM. It contains all the functions the developer will need to allocate virtual and physical registers during the design of the Olive file. The functions available in `TheRegAllocator` are:

- `AsmRegister* GetVReg (int arnum)`
  This function returns a virtual register numbered `arnum`.
- `AsmRegister* GetVReg (AsmOperandKind kind)`
  This function returns a virtual register of type `kind`.
- `AsmRegister* GetVReg (AsmRegister* ar = NULL)`
  This function returns a virtual register if `ar` is NULL, and a virtual register with the same number as `ar` otherwise.
- `AsmRegister* GetVReg(var_sym* var)`
  This function allocates and returns a virtual register for variable `var`.
- `AsmRegIndir* GetVRegIndir (int arnum)`
  This function returns a virtual address register numbered `arnum`.
- `AsmRegIndir* GetVReg (AsmOperandKind kind)`
  This function returns a virtual address register of type `kind`.
- `AsmRegister* GetVRegIndir(AsmRegIndir* ar = NULL)`
  This function returns a virtual address register if `ar` is NULL, and a virtual address register with the same number as `ar` otherwise.
- `AsmRegIndir* GetVRegIndir (var_sym* var)`
  This function returns a virtual address register for pointer variable `var`.

- `AsmRegIndir* GetVRegIndir (otree_node* node, AsmRegIndir* dst = NULL)`
  This function returns a virtual address register for `node`. Node `node` must be of type `nVAR` and the variable associated to it must be a pointer variable. The function allocates a virtual address register to the variable if `dst` is `NULL`, or uses `dst` otherwise. The type of the register (e.g. auto-increment) is defined by an annotation attached to the node.

- `AsmRegIndir* GetVRegIndir (instruction* ins)`
  This function allocates an `AsmArrayRef` to instruction `ins` (see *asm.h* file in the SPAM Manual).. This instruction must be of type `io_array`.

- `AsmRegIndir* GetVRegIndir (AsmRegIndir* ar, int offset)`
  This function returns a virtual address register with the same number of `ar`, which can perform auto-modify operations in the range ± `offset`.

- `AsmRegister* GetPReg()`
  This function returns a physical register.

- `void FreePReg (AsmRegister* r)`
  This function frees register `r`.

- `var_sym* GetTmp (type_node* ty)`
  This function returns a temporary spill location of type `ty`.

- `void FreeTmp (var_sym* tmp)`
  This function frees temporary spill location `tmp`.

A.3 The Virtual Address Register Allocator

DSP applications make extensive use of auto-increment (decrement) addressing and the SPAM compiler provides methods to identify these modes and to optimize the

generation of addressing code (see Chapter 4 for details). Address register allocation is performed using the methods available in two classes: (a) The `IndexAllocator` class; and (b) The `PointerAllocator` class. The methods associated with these classes are described in Section A.3.1 and Section A.3.3 respectively .

The allocation of ARs to pointer variables and array references is done on a procedure by procedure basis. The procedure is represented by an object of type `CFGraph*`. The nodes in `CFGraph*` are objects of the type `CFGnode` corresponding to the procedure basic blocks. Each `CFGnode*` stores a member pointer of type `CompactedInstructionList*`, which is the list of assembly instructions associated with that basic block.

The **for** statements in the source program are not dismantled into its atomic operations, since this will depend on the type of target architecture, one wants to generated code for. In this case, each **for** statement is mapped into a special node in `CFGraph`. The **CFGFor** node is used for this purpose. It contains members which preserve the information available in the original **for** statement: (a) test code; (b) loop lower bound; (c) loop upper bound; and (d) loop body. The `CFGfor` nodes provide methods to access each one of these members. Notice that strength reduction and loop invariant code motion should have been performed in order to guarantee that the code in each loop depends only on the induction variable of the inner most loop.

At this point of the code generation process, instructions use only virtual ARs, which were previously allocated to pointer variables. The virtual address registers (ARs) have annotations associated with the type of operation performed on the AR: (a) `k_index_register`, for a simple indirect access AR; (b) `k_auto_increment` for an AR with post-increment operation; (c) `k_auto_decrement` for an AR with post-decrement operation. Array references are dismantled into their basic components, with the original suif `io_array` instruction being used for this purpose. The `io_array`

are then converted into `AsmArrayRef` operands, which are used by assembly instructions after code scheduling is performed. Observe that array references which can not be computed using the processor addressing modes (e.g. $a[i^2]$) should be dismantled into their basic operations. Hence, the majority of the array references are kept intact down to assembly code. The reason for that is similar to the case of the `for` statements, i.e. the need to keep the source level information around in order to allow easy retargetability.

A.3.1 The Array Index Allocator

The methods in `IndexAllocator` (see below) implements the allocation of array index operations to virtual ARs. Currently the `IndexAllocator` only handles ARs with auto-increment (decrement) modes, but the extension to other addressing modes should not be difficult. The `AllocArrayIndex` method takes a `CFGraph*` and builds the IG using the methods described in Section A.3.1. Each node of the IG corresponds to a `AsmArrayRef` operand which is used by an assembly instruction. The algorithm proceeds performing IG covering using the Boesch-Gimpel algorithm (see Chapter 4). Each path is then assigned to a virtual ARs, using private method `AlocAddrReg` and the map `pathAR`. Observe that each path in a covered IG corresponds to an AR, which can be used to compute the address operations of successive array references present on the path. After virtual address registers are allocated to the paths, the algorithm traverses each path assigning a virtual AR, of type `AsmRegIndir`, for each `AsmArrayRef` on it. Similarly for the case of pointer variables, the `AsmRegIndir` are annotated according to the type of operation associated with the register, such as post-increment (decrement).

```
class IndexAllocator {
    PHash<Path*, AsmRegIndir*> pathAR;

private:
    void AllocAddrReg (Path* path);
    void InitAddrReg (CFGFor* node);
    void AllocArrayIndexRec (CFGBBlock* bblock);

public:
    IndexAllocator() { };
    ~IndexAllocator() { };

    void AllocArrayIndex (CFGraph* cfg);
};


class IGraph : public DiGraph {
    CompactedInstructionList* cil;

    void AddEdges (IGNode* ign);

public:
    IGraph( CompactedInstructionList* clist );
    ~IGraph() { };

    void Print (FILE* fp = stdout);
};
```

The Indexing Graph

The `IGraph` is the SPAM object corresponding to the IG. The `IGraph` is a directed graph where each node corresponds to an array reference (i.e. `AsmArayRef`), and there exists an edge (v_1, v_2) if the reference v_2 can be computed during reference v_2, using the addressing modes in the processor.

Given that the IG is a directed graph, it is inherited from the `DiGraph` class available in the SPAM data-structures (see SPAM Manual). The `IGraph` constructor takes a `CompactedInstructionList` object and builds the corresponding IG. The `CompactedInstructionList` should be formed by concatenating the pointer members `CompactedInstructionList` of the each basic block inside a **for** statement.

The construction of the the IG uses the private method `AddEdges`, and the method `IndexDistance (IGNode* ign1, IGNode* ign2)`. This method computed the indexing distance from two IG nodes `ign1` and `ign2`, which are of type `IGNode*`. The `IGraph` class also provides for a `Print` method.

```
class IGNode : public DiNode {
    AsmArrayRef* arrayRef;
    int offset;

public:
    IGNode(AsmOperand* ar);
    ~IGNode() { };

    AsmArrayRef* ArrayRef()  { return arrayRef; }
    int Offset()             { return offset;    }

    void Print(FILE* fp = stdout, int indent = 0);
};
```

Class `IGnode` represents a node in the IG. A `IGnode` is constructed using an `AsmOperand` which must be an object of type `AsmArrayRef`. Each `AsmArrayRef` contains information about the `offset` required from the current induction variable to compute the array element address. For example, if `AsmArrayRef` stores the reference $a[i-1]$ then `offset` is -1. Method `offset` from `IGNode` can be used to retrieve the array reference corresponding to this `IGNode`, while method `Offset` returns the `offset` required by this reference. The `IGnode` also provides a `Print` method.

The edges in of an `IGraph` are represented in SPAM using the `IGEdge` class. The `IGEdge` constructor, `IGEdge (IGNode* src, IGNode* dst)` takes two `IGnode` operands, `src` and `dst` which correspond to the source and destination nodes on the edge in the IG. Class `IGEdge` stores member `indexDist` which is the indexing distance between nodes `src` and `dst` on the edge. member `indexDist` can only assume values -1, 0, and 1, corresponding to the possible operations that can be computed using an AR. Methods `SetIndexDistance` and `GetIndexDistance` can be used to set and retrieve the edge indexing distance. As before class `IGedge` also provides a `Print`

method.

```
class IGEdge : public DiEdge {
    int indexDist;

public:
    IGEdge(IGNode* src, IGNode* dst);
    ~IGEdge() { };

    void GetIndexDistance (int distance) { indexDist = distance; }
    int  GetIndexDistance ()             { return indexDist;     }

    void Print(FILE* fp = stdout, int indent = 0);
};
```

A.3.2 The Pointer Variable Allocator

Data access in DSP programs are frequently performed using pointers. Auto-increment (decrement) operations are typically used to perform this task. `PointerAllocator` is a SPAM class which is used to allocate pointer variables into virtual ARs. Method `AllocPointerVar` takes the `CFGraph` corresponding to a procedure. It traverses each basic block in the `CFGraph` allocating a virtual AR to each pointer variable. If the pointer has a post-increment (decrement) operator (e.g. `p++`), `AllocPointerVar` attaches annotation `k_auto_increment` (`k_auto_decrement`) to the SUIF instruction (i.e `lod`, `str` or `memcpy`) using the pointer. A `k_index_register` annotation is attached otherwise. This annotation will be used later, during the code generation phase, to emit the appropriate virtual AR.

Private methods `MatchTreeInstr` and `MatchOperand` are used to identify the existence of auto-increment (decrement) operations. Method `MatchInstr` is employed to detect the type of the operation being performed in the pointer.

Notice that pointer allocation can be performed before assembly code generation, unlike array index allocation, which needs the a scheduled assembly code to create

```
class PointerAllocator {
    var_sym* MatchInstr (instruction* ins);
    var_sym* MatchOperand (operand op);
    var_sym* MatchTreeInstr (tree_instr* ti);

    void IncDecOptRec(CFGNode* bb);

public:
    PointerAllocator() { };
    ~PointerAllocator() { };

    void AllocPointerVar (CFGraph* cfg);
}
```

the IG.

A.3.3 Allocating Physical Registers

Once virtual ARs have been allocated to array references and pointer variables, the next task is to perform allocation of physical ARs to the virtual ARs. The developer should use the `InterferenceGraph` method to build an interprocedural interference graph were the nodes are the virtual ARs, and the edges define intersecting liveness between them. The `InterferenceGraph` is a general package in SPAM (see SPAM Manual) which allows the developer to build interference graphs for different types of objects that are used in a `CompactedInstruction`. Actually the developer has to provide two functions. One function which takes a `CompactedInstruction` and returns a bitset corresponding to the the objects that are defined during the execution of that instruction. The second function, does the opposite, it takes a `CompactedInstruction` and returns a bitset associated to the objects which have been destroyed during the execution of the instruction. Example of valid objects are variables, memory positions and virtual ARs. Once the interference graph is built, class `ColorAssignment` can be used to perform register coloring. The number of colors used to color the graph is the number of ARs available in the processor. After that, virtual ARs from each

`CompactedInstruction` are substituted by its corresponding physical ARs and the spilling operations introduced wherever required.

A.4 Building and Dismantling Expression DAGs

This appendix describe the steps the developer should follow in order to build and dismantle the expression DAG using the SPAM compiler. After the expression DAG is dismantled into its component trees, then code generation for the expression trees is used as described in Section A.5.

The `ProcessProc` code sequence presented below generates assembly code from an unoptimized `tree_node_list` schedule. Now it will be described how an optimized schedule may be generated by constructing and dismantling an expression DAG. Consider the modified `ProcessProc` code sequence below:

```
 1  TwifProc* ProcessProc( tree_proc* tp ) {
 2      // Build the TWIF procedure.
 3      TwifProc* twifp = new TwifProc( tp );
 4
 5      // Build the CFG for this TWIF procedure.
 6      twifp->BuildCFG();
 7      EpilogueLabel = twifp->Epilogue()->Label();
 8
 9      // Build expression DAGs for each basic block in this procedure.
10      BuildDAGs( twifp->CFG() );
11
12      // Perform dead-code elimination on each DAG.
13      LocalOpt* localOpt = new LocalOpt( twifp->CFG() );
14      localOpt->DeadCode();
15
16      // Generate assembly code for each basic block.
17      GenerateCode( twifp->CFG() );
18      return twifp;
19  }
```

Those portions of this code sequence which are related to building and dismantling the DAG are described in Section A.4.1 and Section A.4.2 respectively. In general,

the `ProcessProc` works as follows. Line 10 invokes function `BuildDAGs`, which recursively traverses the *Control-Flow Graph* (CFG) so as to reach every basic block, and construct an expression DAG for them. Lines 13 and 14 utilize the `LocalOpt` class to perform local dead code elimination on each expression DAG just constructed. Line 17 invokes `GenerateCode` which generates assembly code for each basic block. `GenerateCode` applies method `DoSchedule` to generate an optimized `tree_node_list` schedule for each basic block. This is done as follows. The expression DAG is dismantled using the traditional heuristic of breaking multiple fanout nodes. A `TreeDAG` (see *treedag.h* in SPAM Manual) is constructed from the dismantled expression DAG. A `TreeDAG` is a DAG where each node contains the component expression trees of the original expression DAG, and the edges define the dependencies between these component trees. Finally, the `TreeDAG` is scheduled using a topological ordering of its nodes, and code is generated for each `TreeDAG` node, using the code generator from Section A.5. The result is a `tree_node_list` for the basic block where *common subexpression* and *dead code* are eliminated.

A.4.1 Building the Expression DAG

`BuildDAGs` may be written as shown below. In line 16, the `BuildExpDAGs` method is invoked, which constructs the expression DAG for the current basic block. The developer is responsible to provide method `BuildExpDAGs` to assemble the expression DAG.

A.4.2 Dismantling the Expression DAG

Dismantling an expression DAG generally results in loss of optimality. The restricted instruction selection caused by separating potential patterns, and the insertion of spill and reload code when edges are broken, usually end up in code sub-optimality.

```
 1  void BuildDAGs( CFGraph* cfg ) {
 2      // Iterate through every node of the CFG.
 3      CFGraphForeachNode( cfg, node ) {
 4        // Check for embedded CFGs.
 5        if ( node->Kind() == CFGNODE_FOR ) {
 6          CFGFor* forNode = (CFGFor*) node;
 7
 8          // Recursively examine the landing pad and body CFGs.
 9          BuildDAGs( forNode->LPad() );
10          BuildDAGs( forNode->Body() );
11          continue;
12        }
13
14        // CFG node is a basic block -- build a DAG for it.
15        CFGBBlock* bb = (CFGBBlock*) node;
16        bb->BuildExpDAGs();
17      } CFGraphHcaerofNode;
18  }
```

In some cases, various properties of the target machine architecture may be used to guide the dismantling process, such that much of the sub-optimality is avoided. The following section describes how the developer may utilize properties of the architecture to improve the dismantling process. The TMS320C25 processor will be used as an illustrative example.

Changing SPAM Code

In order to implement new dismantling techniques, one must first create a class that inherits from `ExpDAG`, as shown below. This derived class, `TMS320ExpDAG` redefines the `Dismantle` function.

```
class TMS320ExpDAG : public ExpDAG {
 public:
  void Dismantle();
};
```

Since the `DoSchedule` method must be modified for each architecture, a class that inherits from `CFGBBlock` must be created. The `TMS320BBlock` class, shown in Figure A.1, is such a derived class and will subsequently be used as the *generator*

class for each procedure, i.e. while constructing the CFG, every basic block will be of type TMS320BBlock. For each generator class defined, two sets of methods are required. For each constructor in the base class, the generator class must have a corresponding constructor. For each `NewBBlock` method in the base class, the generator must have a corresponding `NewBBlock` method. Both sets of methods are shown in the TMS320BBlock declaration of Figure A.1.

In order to specify that all basic blocks constructed must be of type TMS320BBlock, line 3 of `ProcessProc` must be modified as follows:

```
TwifProc* twifp = new TwifProc( tp, &TheTMS320BasicBlock );
```

By modifying method `GenerateCode` in the base SPAM code (see SPAM Manual), as shown below in line 5 of `DoSchedule`, the developer forces the developer-written function `TMS320BBlock::DoSchedule` to be invoked, rather than the corresponding method associated with `CFGBBlock` (i.e. method `CFGBBlock::DoSchedule`). Now `DoSchedule` (shown below) invokes the `TMS320ExpDAG::Dismantle` routine in order to perform the dismantling heuristic provided by the developer, instead of the heuristic in `ExpDAG::Dismantle` which comes with SPAM, and which is based on breaking all outgoing edges from a node.

Architectural Based Dismantle

Dismantling DAGs is a task that can be useful in designing code generation for embedded processor architectures. The developer is responsible for the design of the dismantle function though. In the following, the parts of the algorithm used to dismantle DAGs for the TMS320C25 are described. The `TMS320ExpDAG::Dismantle` method is a rather long algorithm. For the sake of simplicity, only the most important parts of this algorithm are highlighted here. Similar for other algorithms in this document the developer should refer to the appropriate chapters for a complete understanding of the approach. Chapter 3 describes the ideas used in the implementation

```
 1   char* targetTMS320 = ‘‘tms320’’;
 2   class TMS320BBlock : public CFGBBlock {
 3    public:
 4     TMS320BBlock()
 5        : CFGBBlock() { }
 6     TMS320BBlock( label_sym* lab, CFGraph* pcfg = 0 )
 7       : CFGBBlock( lab, pcfg ) { }
 8     TMS320BBlock( char* n, CFGraph* pcfg )
 9       : CFGBBlock( n, pcfg ) { }
10     TMS320BBlock( tree_node_list* l, label_sym* lab, CFGraph* pcfg = 0 )
11       : CFGBBlock( l, lab, pcfg ) { }
12     TMS320BBlock( tree_node_list* t, sym_node_list* labs, CFGraph* pcfg )
13       : CFGBBlock( t, labs, pcfg ) { }
14     TMS320BBlock( sym_node_list* labs, CFGraph* pcfg ) : CFGBBlock( labs, pcfg ) { }
15     ~TMS320BBlock() { }
16
17     CFGBBlock* NewBBlock( label_sym* lab, CFGraph* pcfg = 0 )
18       { return new TMS320BBlock( lab, pcfg );     }
19     CFGBBlock* NewBBlock( char* n, CFGraph* pcfg )
20       { return new TMS320BBlock( n, pcfg );       }
21     CFGBBlock* NewBBlock( tree_node_list* l, label_sym* lab, CFGraph* pcfg = 0 )
22       { return new TMS320BBlock( l, lab, pcfg );  }
23     CFGBBlock* NewBBlock( sym_node_list* labs, CFGraph* pcfg = 0 )
24       { return new TMS320BBlock( labs, pcfg );    }
25     CFGBBlock* NewBBlock( tree_node_list* l, sym_node_list* labs, CFGraph* pcfg = 0 )
26       { return new TMS320BBlock( l, labs, pcfg ); }
27
28     char* Target() { return targetTMS320; }
29     void DoSchedule();
30   };
31   extern TMS320BBlock TheTMS320BasicBlock;
```

Figure A.1: TMS320BBlock class is inherited from CFGBBlock.

```
1   void TMS320BBlock::DoSchedule() {
2     // Check if expression DAG has already been built.
3     if ( dag ) {
4       // Type-cast the DAG to the new derived type.
5       TMS320ExpDAG* tmsExpDAG = (TMS320ExpDAG*) dag;
6
7       // Dismantle the DAG using developer-written heuristics.
8       tmsExpDAG->Dismantle();
9
10      // Construct a tree-DAG for the dismantled DAG.
11      TreeDAG* tdag = new TreeDAG( tmsExpDAG );
12      delete schedule;
13
14      // Generate the optimized tree_node_list schedule from the tree-DAG.
15      schedule = tdag->Schedule();
16      delete tdag;
17    }
18  }
```

of the `TMS320ExpDAG::Dismantle` method.

The `TMS320ExpDAG::Dismantle` algorithm starts by breaking all multiple fanout nodes in the DAG which are leaf-nodes. Leaf nodes represent constant and variables in the source program. Breaking outgoing edges from constant and primary input nodes are similar tasks. The code below shows the case when the nodes are constant nodes. The constant nodes of a DAG are stored into the `ExpDAG` object and can be accessed using iterator `ForeachConstantNode` (line 2). Similarly the iterator `ForeachPrimaryInput` can be used to access the primary input (i.e. program variables) in the DAG.

The algorithm traverses each constant node `conNode`. If `conNode` has only one outgoing edge then there is no need to break its outgoing edges and the algorithm proceeds. Otherwise `SpillNode` (line 4) is executed to find a node where to spill `conNode`. Since `conNode` is a constant, `SpillNode` returns this node as `spillNode`. The same also happens for the case of primary input nodes. Next, the iteration macro `ExpDAGNodeForeachOprOutEdge` takes each operation edge out of `conNode` (i.e. `edge`) and breaks it. This is done by means of method `SpillEdge`. This method takes as

```
1    // Break edges from multiple fanout constant nodes.
2    ForeachConstantNode( conNode ) {
3      if ( conNode->NumOprFanout () > 1 ) {
4        ExpDAGNode* spillNode = conNode->SpillNode();
5
6        ExpDAGNodeForeachOprOutEdge( conNode, edge ) {
7          ExpDAGNode* newCon = edge->SpillEdge( spillNode );
8
9          ExpDAGNodeForeachConInEdge( conNode, edge ) {
10            InsertEdge( edge->SrcNode(), newCon, edge->Type() );
11         } ExpDAGNodeHcaerofConInEdge;
12
13         ExpDAGNodeForeachConOutEdge( conNode, edge ) {
14           InsertEdge( newCon, edge->DstNode(), edge->Type() );
15         } ExpDAGNodeHcaerofConOutEdge;
16
17       } ExpDAGNodeHcaerofOprOutEdge;
18       if ( !conNode->num_operators() )
19          delete RemoveNode( conNode );
20     }
21   } HcaerofConstantNode;
```

argument the new source node of the broken edge, i.e. node `spillNode` (which is the same as `conNode`). Then, it executes the following tasks: (a) disconnects *edge* from `conNode`; (b) creates a new constant node `newCon` which will be the new source of `edge`. In this case `newCon` is just a clone of `spillNode` (i.e. `conNode`). Next, iterator `ExpDAGNodeForeachConOutEdge` (line 13) is used to redirect all input constraint edges from `conNode` into `newCon`. Finally, `conNode` is removed from the DAG.

Natural edges are edges that can be disconnected from a DAG without impacting the quality of the code after the code generation is performed. The approach used to disconnect *natural edges* from the DAG is similar to the one just described above.

In this case each internal node `intNode` of the DAG is visited using iterator `ForeachInternalNode`. Similarly as before only those multiple fanout nodes (for which `NumOprFanout` > 1) are of any interest. In the case of internal nodes, method `SpillNode` does the following: (a) creates a spilling (primary output) node (`spillNode`) out of `intNode` using a temporary variable `tmp`. The developer then has to provide

```
1    // Break natural edges.
2    ForeachInternalNode( intNode ) {
3      if ( intNode->NumOprFanout() > 1 ) {
4        ExpDAGNode* spillNode = intNode->SpillNode();
5
6        ExpDAGNodeForeachOprOutEdge( intNode, edge ) {
7          ExpDAGNode* dst = edge->DstNode();
8
9          if ( (spillNode != edge->DstNode()) &&
10              ((IsAcc(intNode) && IsPreg(dst)) ||
11              (IsPreg(intNode) && IsPreg(dst))) )
12            edge->SpillEdge( spillNode );
13       } ExpDAGNodeHcaerofOprOutEdge;
14     }
15   } HcaerofInternalNode;
```

functions to identify natural edges. In the TMS320C25 natural edges exist from nodes which store into the `acc` to nodes which store into `preg`, and from those nodes which store into `preg` to nodes which store back into `preg`. Functions `IsAcc` and `IsPreg` should be defined by the developer to identify if the operation in a DAG node store its result into `acc` or `preg`. These functions return true if this occur and false otherwise. If an edge is identified as natural then method `SpillEdge` is used to break `edge` as before. Again this method takes `spillNode` as an argument. Since `intNode` is an internal node then `SpillEdge` does the following: (a) creates a primary input node to read `tmp` using the variable name (`tmp`) in `spillNode`; (b) breaks `edge` and makes `spillNode` the source node of `edge`; (c) creates a constraint *RAW* edge from `spillNode` to this input node, which is used later in order to enforce the scheduler.

Similar DAG dismantling algorithms, for other DSP architectures, can also be designed by the developer using similar techniques as those just described above.

A.5 Examples

This section gives two small examples on how to use Olive to design code generators. Section A.5.1 describes an example for a generic non-superscalar RISC architecture using the Sethi-Ullman algorithm, while Section A.5.2 considers the case of the TMS320C25 DSP processor.

A.5.1 RISC Example

RISC architectures are today the *de facto* architectural style used in the design of modern microprocessors [9]. They usually have a single register file which typically contains at least 32 registers. The ISA is homogeneous, i.e. any instruction can use any combination of registers as operands and destination of the operation[2]. For the sake of simplicity the RISC microarchitecture considered here is not superscalar.

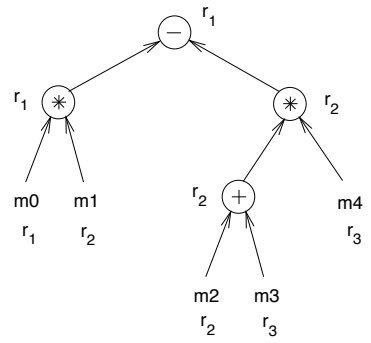


Figure A.2: Allocation of an expression tree for a RISC architecture.

Register Allocation and Scheduling Algorithm

The Sethi-Ullman algorithm [44] can be used to perform the tasks of register allocation and scheduling in a RISC architecture. The idea is to determine the best scheduling

[2]The exception to that is register r_0, which is usually assumed to be zero.

such as to minimize the number of registers required to execute the instructions associated to the tree operations.

The algorithm works as follows. At each node of a tree the number of registers used in the left (N_{left}) and right subtrees (N_{right}) are compared. If they are the same then the number of registers used by the current node is just that number plus one. Otherwise, the number of registers used by the current node is $min\ (N_{left}, N_{right}) + 1$. Consider for example the expression tree from Figure A.2. When the Sethi-Ullman algorithm is applied to it, the schedule is such that registers r_1 and r_2 are first loaded with memory variables m_1 and m_2, and the left subtree multiplication is performed. The result of the multiplication can be stored into register r_1, given that this register is not needed anymore, after the multiplication is performed. The schedule proceeds to the right subtree. The addition operation uses registers r_2 and r_3, storing the result into r_2. Following that register r_3 is used to load m_4, and the result of the multiplication is stored into r_2. Finally, the subtraction operation at the root of the tree is scheduled, storing its result into r_1. The final schedule requires only three registers live at the same time.

The RISC Olive File

Implementing the Sethi-Ullman algorithm using Olive to generate code for the tree above is a very straightforward task. A single non-terminal is assigned (`reg`), to represent a register available in the processor register file. Three terminals `nADD`, `nMUL`, `nSUB` and `nVAR` are used to represent addition, multiplication, subtraction and load operations. For the case of this example, consider a local register allocation strategy, in which registers are allocated in a tree basis, using the Sethi-Ullman algorithm.

The RISC Olive file is described below. The description does not take care of the case when memory spill occurs. This is simple to implement, and can be done using class `mset` and iterator `SetForeachItem`, as it is described in the DSP example

(Section A.5.2).

The first rule (lines 1 – 31) is used to match the `ADD` operation. In the cost part of that rule (lines 2 – 14) the cost of the subtree rooted at that node (i.e. `$cost[0].cost`) is computed as the sum of the number of cycles used by the `nADD` instruction (i.e. 1 cycle), plus the number of cycles required to compute the left (`$cost[2].cost`) and right subtrees (`$cost[3].cost`). The number of registers used by the current subtree (`$cost[0].nregs`) is determined comparing the number of registers used by the left subtree (`$cost[2].nregs`) with the number of registers used by the right subtree (`$cost[3].nregs`). This is done using the Sethi-Ullman algorithm. If a left (right) first schedule is required, then bit (`$cost[0].dir`) is set to `LEFT` (`RIGHT`). The action part of the first rule (lines 15 – 31) is executed after the pattern matching has been performed, and the fields `nregs` and `dir` of each node are known. In that section the code associated to the subtrees of the current node are scheduled according to the value of bit `$action[0].dir` set previously during the pattern matching phase. Olive command `$action[2]` schedules the code of the left subtree by executing the C statements in the action part of the rule matched by the left child of the current node. The action part of each matched node returns the register allocated to that node, like `rl` (line 30), and frees the register allocated to one of the children, e.g. the right child (`rr`) in (line 29), after emitting the instruction which used them (line 28). In other words, the register used by one of the child (e.g. `rl`) is kept to store the result of the current operation.

```
 1 reg : nADD (reg, reg)
 2 {
 3     $cost[0].cost = 1 + $cost[2].cost + $cost[3].cost;
 4
 5     if ($cost[2].nregs >= $cost[3].nregs) {
 6         $cost[0].dir = LEFT;
 7         $cost[0].nregs = 1 + $cost[2].nregs;
 8     }
 9     else {
10         $cost[0].dir = RIGHT;
```

```
11          $cost[0].nregs = 1 + $cost[3].nregs;
12      }
14 } =
15 {
16      AsmRegister* rl, rr;
18      if (LEFT) {
19          rl = $action[2]();
20          rr = $action[3]();
21      }
22      else {
23          rr = $action[3]();
24          rl = $action[2]();
25      }
26      AsmOperation* ao;
27      ao = new AsmOperation (aMUL, rl, rr, rl, NULL);
28      cil->Emit (new CompactedInstruction (ao, NULL));
29      TheRegAllocator.FreeReg (rr);
30      return rl;
31 };
32
33
34 reg : nMUL (reg, reg)
35 {
36      $cost[0].cost = 1 + $cost[2].cost + $cost[3].cost;
37
38      if ($cost[2].nregs <= $cost[3].nregs) {
39          $cost[0].dir = LEFT;
40          $cost[0].nregs = 1 + $cost[2].nregs;
41      }
42      else {
43          $cost[0].dir = RIGHT;
44          $cost[0].nregs = 1 + $cost[3].nregs;
45      }
46 } =
47 {
48      AsmRegister* rl, rr;
49      if (LEFT) {
50          rl = $action[2]();
51          rr = $action[3]();
52      }
53      else {
54          rr = $action[3]();
55          rl = $action[2]();
56      }
57    AsmOperation* ao;
58      ao = new AsmOperation (aMUL, rl, rr, rl, NULL);
59      cil->Emit (new CompactedInstruction (ao, NULL));
60      TheRegAllocator.FreeReg (rr);
```

```
61     return rl;
62 };
63
64
65 reg : nSUB (reg, reg)
66 {
67     $cost[0].cost = 1 + $cost[2].cost + $cost[3].cost;
68
69     if ($cost[2].nregs <= $cost[3].nregs) {
70         $cost[0].dir = LEFT;
71         $cost[0].nregs = 1 + $cost[2].nregs;
72     }
73     else {
74         $cost[0].dir = RIGHT;
75         $cost[0].nregs = 1 + $cost[3].nregs;
76     }
77 } =
78 {
79     AsmRegister* rl, rr;
80     if (LEFT) {
81         rl = $action[2]();
82         rr = $action[3]();
83     }
84     else {
85         rr = $action[3]();
86         rl = $action[2]();
87     }
88     AsmOperation* ao;
89     ao = new AsmOperation (aSUB, rl, rr, rl, NULL);
90     cil->Emit (new CompactedInstruction (ao, NULL));
91     TheRegAllocator.FreeReg (rr);
92     return rl;
93 };
94
95
96 reg : nVAR
97 {
98     $cost[0].cost = 1;
99     $cost[0].nregs = 1;
100 } =
101 {
102     AsmRegister* r = TheRegAllocator.GetReg ($1->variable());
103     AsmOperation* ao;
104     ao = new AsmOperation (aLD, r, $1->variable(), NULL);
105     cil->Emit (new CompactedInstruction (ao, NULL));
106     return r;
107 };
```

Instructions are considered here as operations, and constructor `AsmOperation` is used to create them. This constructor takes as arguments the SPAM assembly opcode for the operation (i.e. aADD in this case), and the registers it uses as operands and result. Operations are compacted into a `CompactedInstruction`, by means of constructor `CompactedInstruction` which takes a variable length argument list containing operations. The compaction of operations into a `CompactedInstruction` occurs after sequential code generation is performed using Olive. The compacted instructions generated from the expression trees of the current basic block are appended into list `cil` using function `Emit`. The `Emit` function is a member of the class `CompactedInstructionList`.

The other rules for the `nMUL` and `nSUB` operations are almost identical to the `nADD` just described. If the developer wants he (she) can create a function to merge the common tasks present on these rules. The rule for the load operation (lines 93 – 102) is a special case. It uses method `GetPReg` from `TheRegAllocator` to get a physical register for the variable stored in terminal `nVAR`.

A.5.2 DSP Example

DSPs are irregular architectures which contain a number of small register files. The ISA of these processors are usually heterogeneous, in the sense that instructions require operands to be stored in specific register files. Code generation for DSPs is a hard and not well understood problem. Not many DSP architectures have well-defined algorithms for instruction selection, register allocation and scheduling. One of the exceptions is the Texas Instruments TMS320C25 processor. This is the architecture which will be used in the following sections.

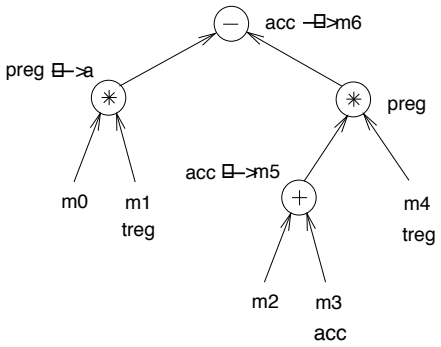


Figure A.3: Allocation of an expression tree for a DSP architecture.

Register Allocation and Scheduling Algorithm

The TMS320C25 is a memory-register heterogeneous architecture. It contains an acyclic datapath, what permits optimal sequential code generation in $O(n)$ using the *RTG algorithm* (see Chapter 3). The idea is to first recursively schedule subtrees for which the root stores into memory. Since the TMS320C25 has an acyclic datapath the remaining subtrees are free of allocation deadlocks and can be scheduled without spills. The tree of Figure A.3 shows the registers used by the instructions after register allocation and scheduling have been performed.

The DSP Olive File

Three register classes (or register files) are used in the TMS320C25: a, p and t. Each class contains a single register. These classes are respectively associated to non-terminals `acc`, `preg` and `treg` in the Olive file. Non-terminal `lword` is used to match local variables represented by terminal `nVAR`. The tree of Figure A.3 requires three operation terminals: `nADD`, `nSUB` and `nMUL`. During the pattern matching phase the cost of each operation (e.g. `$cost[0].cost`) is computed based on the cost of the current operation and on the cost of its subtrees. For example, the cost of the `nADD` operation in line 3 is determined by adding the cost of matching that instruction (2

cycles, 1 cycle to set the extension mode for the accumulator in line 11) plus the cost of its children subtrees rooted at `acc` and `lword`.

The RTG algorithm requires subtrees for which the root was allocated to memory to be scheduled ahead. In order to achieve that, each subtree stores a set of pointers to its member nodes which where allocated to memory. This information is kept in the field `mset` of the node (e.g. `$cost[0].mset` in the nADD node). Field `mset` is a set data type from the class `Set` (see *set.h* in SPAM Manual). The `mset` of each node `mset` is computed as the union of the `mset` of its children. The union operation is performed by the + operator. For example, in line 4, the union of `$cost[2].mset` and `$cost[3].mset` is stored into set `$cost[0].mset`.

The RTG algorithm states that any node which stores in memory, or which is the root of an expression tree (e.g. an assignment node) has to schedule its subtrees rooted in memory in advance. Consider, for example, the rule in lines 78 – 94, which matches an store operation. When the action part of that rule is executed the `SetForeachItem (elem, set)` construct (see class `Set`) is used to iterate over all elements `elem` from set *set*. The Olive function `$getcost[0].mset` is used to get the `mset` of the current node[3]. As mentioned before, `mset` stores pointers to all those nodes allocated to memory which are contained in the current subtree. The Olive function `$exec[elem]` executes the action part of the rule which matched `elem` (line 85), before scheduling code for the children of the current node. In order to keep track of the memory position used for the operation at the node represented by `elem`, this position is stored into field `tmp` of that node, and can be accessed later by using `$getcost[].tmp`.

```
1 acc : nADD (acc , lword)
2 {
3     $cost[0].cost = 2 + $cost[2].cost + $cost[3].cost;
```

[3]Function `$getcost[]` is for the action part of a rule, the same what `$cost[]` is for the cost part.

```
 4     $cost[0].mset = $cost[2].mset + $cost[3].mset;
 5 } =
 6 {
 7     $action[2](cil);
 8     $action[3](cil);
 9     AsmOperation* ao;
10     ao = new AsmOperation (aSSXM,NULL);
11     cil->Emit (new CompactedInstruction (ao, NULL));
12     if ( CanBeStatic(OliveCurrProc) && TwifDoStaticAlloc )
13         ao = new AsmOperation (aADD, immed($3->variable(), NULL)
14     else
15         ao = new AsmOperation (aADD, new AsmVariable($3->variable(), NULL);
16     cil->Emit (new CompactedInstruction (ao, NULL));
17 };
18
19
20
21 preg : nMUL (lword , treg)
22 {
23     $cost[0].cost = 1 + $cost[2].cost + $cost[3].cost;
24     $cost[0].mset = $cost[2].mset + $cost[3].mset;
25 } =
26 {
27     $action[3](cil);
28     $action[2](cil);
29     AsmOperation* ao;
30     if ( CanBeStatic(OliveCurrProc) && TwifDoStaticAlloc ) {
31         if ($1->IsSigned())
32             ao = new AsmOperation (aMPY, immed($2->variable(), NULL);
33         else
34             ao = new AsmOperation (aMPYU, immed($2->variable(), NULL);
35     }
36     else {
37         if ($1->IsSigned())
38             ao = new AsmOperation (aMPY, new AsmVariable($2->variable(), NULL);
39         else
40             ao = new AsmOperation (aMPYU, new AsmVariable($2->variable(), NULL);
41     }
42     cil->Emit (new CompactedInstruction (ao, NULL));
43 };
44
45
46
47 acc : nSUB (acc , preg)
48 {
49     $cost[0].cost = 2 + $cost[2].cost + $cost[3].cost;
50     $cost[0].mset = $cost[2].mset + $cost[3].mset;
51 } =
```

```
52 {
53     $action[2](cil);
54     $action[3](cil);
55     AsmOperation* ao;
56     ao = new AsmOperation (aSETPM, 0, NULL);
57     cil->Emit (new CompactedInstruction (ao, NULL));
58     ao = new AsmOperation (aSPAC, NULL);
59     cil->Emit (new CompactedInstruction (ao, NULL));
60 };
61
62
63   acc : preg
64   {
65     $cost[0].cost = 2 + $cost[1].cost;
66     $cost[0].mset = $cost[1].mset;
67   } =
68   {
69     $action[1](cil);
70     AsmOperation* ao;
71     ao = new AsmOperation (aSETPM, 0, NULL);
72     cil->Emit (new CompactedInstruction (ao, NULL));
73     ao = new AsmOperation (aPAC, NULL);
74     cil->Emit (new CompactedInstruction (ao, NULL));
75   };
76
77
78   stackw : acc
79   {
80     $cost[0].cost = 1 + $cost[1].cost;
81     $cost[0].mset = $cost[1].mset;
82   } =
83   {
84     SetForeachItem(it, $getcost[0].mset) {
85         $exec[it](cil);
86     } SetHcaerofItem;
87
88     $action[1](ail);
89     $getcost[0].tmp = GetTmp(type_unsigned);
90     ao = new AsmOperation (aLDPK, immed($getcost[0].tmp), NULL);
91     cil->Emit (new CompactedInstruction (ao, NULL));
92     ao = new AsmOperation (aSACL, immed($getcost[0].tmp), NULL);
93     cil->Emit (new CompactedInstruction (ao, NULL));
94   };
```

After that the algorithm proceeds to schedule code for the subtrees of the current node. The order in which this occurs depends on the relation, in the RTG, between

the registers which were allocated to the children nodes. The RTG algorithm requires that if all memory rooted subtrees have been scheduled, then there cannot be a node allocated to `acc` in the subtree rooted by a node allocated to `preg`. For example, in Figure A.3, if the subtree rooted at the addition is scheduled first then the result of that operation is transferred from `acc` into memory position m_5. As a consequence of that, the right subtree, which is rooted in a node allocated to `preg`, will not contain any node allocated to `acc`. Consider now the action part of the rule from lines 47 – 60. The left child of the `nSUB` operation was allocated to `acc`, while its right subtree was allocated to `preg`. Assuming that all memory rooted subtrees have been scheduled, then the right subtree will not contain a node allocated to `acc`. Therefore the algorithm can first emit the instructions from the left subtree, by executing `$action[2]`, and this will leave a value live in `acc`. When the algorithm schedules the right subtree rooted at `preg` (by executing `$action[3]`), no instruction will make use of `acc`, according to the RTG algorithm, and then no spill operation is required. Finally, after the instructions associated to the children of a node are scheduled then the instruction corresponding to that node is emitted using the `Emit` function.

Bibliography

[1] A. Peleg, Wilkie S., and Weiser U. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):25–38, January 1997.

[2] The visual instruction set (VIS) on-chip support for new media processing. Technical Report Whitepaper 95-022, Sun Microsystems, 1996.

[3] E. Killian. Extending the MIPS instruction set for digital media and emerging applications. In *Microprocessor Forum 1996*, October 1996.

[4] A.V. Oppenheim, editor. *Digital Signal Applications*. Prentice-Hall, 1978.

[5] L. R. Rabiner and Schafer R. W. *Digital Representations of the Speech Waveform*, chapter Chapter 5. Prentice-Hall, Inc., 1978.

[6] G.K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):31–44, April 1991.

[7] D.L. Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):47–63, April 1991.

[8] V. Bhaskaran and Konstantinos K. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, 1995.

[9] J.L. Hennessy and D.A. Patterson. *Computer Architectures: A Quantitative Approach*. Morgan Kaufmann, 1996.

[10] P. Athanas and Harvey F.S. Processor reconfiguration through instruction-set metamorphosis. *Computer*, pages 11–18, March 1993.

[11] R.D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[12] R. Razdan and Smith M. D. A high performance microarchitecture with hardware-programmable functional units. In *Micro 27*, pages 172–180, November 1994.

[13] Gokhale et al. SPLASH: A reconfigurable linear logic array. In *International Conference on Parallel Processing*, pages 526–532, August 1990.

[14] M. Gold and Bindra A. DSP world rocked core. *Electronic Engineering Times*, pages 11–18, March 1996.

[15] (IDC) Study Group. Technical report, International Data Corp., April 1996.

[16] Recent IC announcements. Technical report, Microprocessor Report, August 1995.

[17] J. Turley and Lapsley P. New 56301 DSP doubles 24-bit performance. Technical report, Microprocessor Report, December 1995.

[18] L. Gwennap. Improved cost model puts Pentium at $180. Technical report, Microprocessor Report, September 1994.

[19] E. A. Lee. Programmable DSP architectures: Part I. *IEEE ASSP Magazine*, pages 4–19, October 1988.

[20] S.C. Lawrence, A.C Payne, and Levergood T.M. Are DSP chips obsolete? Technical Report Series CRL 92/10, Digital Equipment Corporation, Cambridge Research Lab, November 1992.

[21] The DSPnet. http://www.dspnet.com.

[22] EECS UC Berkeley. CPU info & system performance summary. http://infopad.eecs.berkeley.edu/CIC/summary.

[23] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1988.

[24] G. De Micheli. *Hardware/Software Codesign: Application Domains and Design Technologies*. Kluwer Academic Publishers, 1995.

[25] P.G. Paulin, C. Liem, T. May, and Sutarwala S. DSP design tool requirements for embedded systems: A telecommunications industrial perspective. *Journal of VLSI Signal Processing*, 9:23–47, March 1995.

[26] The SPAM Project. http://ee.princeton.edu/spam.

[27] The SUIF Project. http://suif.stanford.edu/suif.

[28] S. Liao, S. Devadas, K. Keutzer, and A. Wang. Storage assignment to decrease code size. *ACM Transactions on Programming Languages and Systems*, 18:235–253, May 1996.

[29] A. Sudarsanam, S. Malik, S. Tjiang, and S. Liao. Optimization of embedded DSP programs using post-pass data-flow analysis. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1997.

[30] A. Sudarsanam and S. Malik. Memory bank and register allocation in software synthesis for ASIPS. In *International Conference on Computer Aided Design*, pages 388–392, 1995.

[31] G. Araujo, S. Malik, and M. Lee. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proc.* 33^{rd} *Design Automation Conference*, pages 591–596, June 1996.

[32] G. Araujo, A. Sudarsanam, and Malik S. Instruction set design and optimizations for address computation in DSP processors. In 9*th International Symposium on Systems Synthesis*, pages 31–37, November 1996.

[33] E. A. Lee. Programmable DSP architectures: Part II. *IEEE ASSP Magazine*, pages 4–14, January 1989.

[34] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee. *DSP Processor Fundamentals: Architectures and Features*. IEEE Press, 1996.

[35] P. Lapsley, J. Bier, and E. A. Lee. Buyer's guide to DSP processors. *IEEE ASSP Magazine*, pages 4–14, January 1989.

[36] Motorola. *DSP56000/DSP56001 Digital Signal Processor User's Manual*, 1990.

[37] Analog Devices. *ADSP-2100 Family User's Manual*, 1995.

[38] Texas Instruments. *TMS320C2x User's Guide*, 1990.

[39] NEC. *μPD77016 User's Manual*, 1993.

[40] Texas Instruments. *TMS320C5x User's Guide*, 1993.

[41] Texas Instruments. *TMS320C54x User's Guide*, 1995.

[42] D. Landskov, S. Davidson, B. Shriver, and P.W. Mallet. Local microcode compaction techniques. *Computer Surveys*, 12:261–294, September 1980.

[43] J.A Fisher. Trace scheduling: A technique for global microcode compation. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[44] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[45] G. Hirohisa, T. Ikezawa, and Tanabe T. A 32 bit floating point digital signal processor FDSP-4 and its application to communication systems. In *Proc. of Globecom*, pages 12.1.1 – 12.1.5, 1987.

[46] W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1), March 1995.

[47] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[48] D. Callahan and B Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 192–202, June 1991.

[49] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, New York, 1979.

[50] J.L. Bruno and R. Sethi. Code generation for one-register machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[51] R. Sethi. Complete register allocation problems. *SIAM J. Computing*, 4(3):226–248, September 1975.

[52] A.V. Aho, S.C. Johnson, and J.D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[53] B. Prabhala and R. Sethi. Efficient computation of expressions with common subexpressions. *Journal of the ACM*, 27(1):146–163, January 1980.

[54] J.L. Bruno and R. Sethi. The generation of optimal code for stack machines. *Journal of the ACM*, 22(3):382–396, July 1975.

[55] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[56] A.V. Aho, S.C. Johnson, and J.D. Ullman. Code generation for machines with multiregister operations. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, pages 21–28, January 1977.

[57] A.W. Appel and K.J. Supowit. Generalizations of the Sethi-Ullman algorithm for register allocation. *Software – Practice and Experience*, 17(3):417–421, June 1987.

[58] J.R. Goodman and A.W. Hsu. Code scheduling and register allocation in large basic blocks. In *Proceedings of the 1988 Conference on Supercomputing*, pages 442–452, July 1988.

[59] D.G. Bradlee, Eggers S.J., and R.R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[60] E.G. Coffman and R. Sethi. Instruction sets for evaluating arithmetic expressions. *Journal of the ACM*, 30(3):457–478, July 1983.

[61] P. Marwedel. Tree-based mapping of algorithms to predefined structures. In *Int.Conf. on Computer-Aided Design*, pages 586–593, 1993.

[62] C. Liem, Trevor M, and Paulin P. Instruction-set matching and selection for DSP and ASIP code generation. In *European Design and Test Conference*, pages 31–37, 1994.

[63] D. Lanner, M. Cornero, G. Goosens, and H. De Man. Data routing: a paradigm for efficient data-path synthesis and code generation. In *High-Level Synthesis Symposium*, pages 17–22, 1994.

[64] B. Wess. On the optimal code generation for signal flow computation. In *Proc. Int. Conf. Circuits and Systems*, volume 1, pages 444–447, 1990.

[65] B. Wess. Automatic instruction code generation based on trellis diagrams. In *Proc. Int. Conf. Circuits and Systems*, volume 2, pages 645–648, 1992.

[66] Marwedel and Goosens, editors. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, Massachusetts, 1995.

[67] Texas Instruments. *Digital Signal Processing Applications with the TMS320 Family*, 1990.

[68] A.V. Aho, M. Ganapathi, and S.W.K Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Prog. Lang. and Systems*, 11(4):491–516, October 1989.

[69] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator. *Journal of the ACM*, 22(12):248–262, March 1993.

[70] S. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[71] C.M. Hoffman and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1992.

[72] V. Zivojnovic, J.M. Velarde, and C. Sclåager. DSPstone, a DSP benchmarking methodology. Technical report, Aachen University of Technology, August 1994.

[73] P. Lapsley and G. Blalock. How to estimate DSP processor performance. *IEEE Spectrum*, pages 74–82, July 1996.

[74] C.Y. III Hitchcock. *Addressing Modes for Fast and Optimal Code Generation*. PhD thesis, Carnegie-Mellon University, December 1987.

[75] L.P. Horwitz, R.M. Karp, R.E. Miller, and S. Winograd. Index register allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[76] D. H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Software Practice and Experience*, 22(2):101, February 1992.

[77] C. Liem, P. Paulin, and A. Jerraya. Address calculation for retargetable compilation and exploration of instruction-set architectures. In *Proc.* 33^{rd} *Design Automation Conference*, pages 597–600, June 1996.

[78] F.T. Boesch and J.F. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *Journal of the ACM*, 24(2):192–198, April 1977.

[79] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartide graphs. *SIAM J. Comput,*, 2(4):225–230, December 1973.

[80] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proc. of the ACM SIGPLAN'89 on Conference on Programming Language Design and Implementation*, pages 98–105, June 1982.

[81] F.C. Chow and J. L. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, 12(4):501–536, October 1990.

[82] G. Lal and A.W. Appel. Iterated register coalescing. *ACM Trans. Programming Language and Systems*, 18(3):300–324, May 1997.

[83] J. Edmonds. The Chinese Postman Problem. *Operations Research*, 13(1):373, 1965.

[84] A. Balachandran, D.M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[85] Yacc – yet another compiler compiler. Technical Report Computing Science 32, AT&T Bell Laboratories.