CODE OPTIMISATION USING DISCRETE OPTIMISATION TECHNIQUES

by

TRISTAN DIDIER ALEXANDRE DOPLER

DISSERTATION

submitted in fulfilment
of the requirements for the degree

MASTER OF SCIENCE

in

COMPUTER SCIENCE

in the

FACULTY OF SCIENCE

JOHANNESBURG

at the

UNIVERSITY OF JOHANNESBURG

SUPERVISOR: PROF. T.H.C. SMITH

CO-SUPERVISOR: MR. A. HARDY

JULY 2005

Abstract

Keywords

- Combinatorial optimisation
- Compilers (Computer programs)
- Scheduling

The topic for this dissertation is the optimisation of computer programs, as they are being compiled, using discrete optimisation techniques. The techniques introduced aim to optimise the runtime performance of programs executing on certain types of processors.

A very important component of this dissertation is the movement of complexity from the processor to the compiler. Therefore both computer architecture and compilers are important supporting topics. The data output of the compiler is processed using information about the processor to produce execution information which is the goal of this dissertation.

Concepts related to instruction level parallelism are covered in two parts. The first part discusses implicit parallelism, where parallel instruction scheduling is performed by the processor. The second part discusses explicit parallelism, where the compiler schedules the instructions. Explicit parallelism is attractive because it allows processor design to be simplified resulting in multiple benefits.

Scheduling the instructions to execute while adhering to resource limitations is the area of focus for the rest of the dissertation. In order to find optimal schedules the problem is modelled as a mathematical program. Expressing instructions, instruction dependencies and resource limitations as a mathematical program are discussed in detail with several algorithms being introduced. Several aspects prevent the mathematical programs from being solved in their initial state, therefore additional techniques are introduced.

A heuristic algorithm is introduced for scheduling instructions in a resource limited environment. The primary use of this heuristic is to reduce the computational complexity of the problem. However, this heuristic algorithm can be used to generate good schedules on its own.

Finally information regarding a practical implementation of a compiler that implements the introduced techniques is introduced as well as experimental results. The experimental results are generated from a series of test programs illustrating the complete process and the computational complexity of the algorithms employed.

Contents

Contents **i**

1 Introduction **1**
1.1 Runtime optimisation . . . 2
1.2 Optimisation during compilation . . . 2
1.3 Goals . . . 3
1.4 Chapter overview . . . 3

2 Computer architecture **6**
2.1 Introduction to the components of a computer . . . 6
2.2 Instructions as an interface to the processor . . . 7
2.3 Approaches to work space implementation . . . 10
2.4 Components of a modern processor . . . 13
2.5 Superscalar processing . . . 16
2.6 Implicitly parallel instruction computing . . . 16
2.6.1 Runtime scheduling/queueing . . . 17
2.6.2 Out-of-order execution . . . 20
2.6.3 Branch prediction . . . 22
2.6.4 Implicit parallelism in practice . . . 24
2.7 Explicitly parallel instruction computing . . . 25
2.7.1 Vector processors . . . 26
2.7.2 Very large instruction word processors . . . 27
2.7.3 Branch predication . . . 29

3 Compilation **31**
3.1 Compilers in tool chains . . . 31
3.2 Structure of a compiler . . . 32
3.3 Lexical analysis . . . 33
3.4 Syntax analysis . . . 34
3.5 Semantic analysis . . . 36
3.6 Intermediate code generation . . . 38
3.6.1 Generating data dependency graphs . . . 40

UNIVERSITY OF JOHANNESBURG

3.7 Code optimisation 42
3.8 Code generation 44
3.9 Automating the compiler creation process 44

4 Instruction scheduling for code optimisation **46**
4.1 Formulating the scheduling problem as a mathematical program 47
4.2 Minimising the makespan 49
4.3 Justification for the use of continuous variables for instruction times 50
4.4 Resource constraints 52
4.5 Disjunctive method for eliminating parallelism with a single execution unit . . . 53
4.6 Locating resource contention 55
4.6.1 Generating constraints from resource contention sets 57
4.6.2 Adaptations for multiple identical resources 60
4.6.3 Absolute difference approach to constraint relaxations 61
4.6.4 The mathematical program for the example 62
4.7 An alternative method for a special case 64
4.8 A branch and bound approach to limiting parallelism 68

5 Implementing instruction scheduling **73**
5.1 Binding a variable to the absolute difference of two variables 73
5.1.1 Why the upper bound technique will not work 74
5.1.2 The difficulty in enforcing the absolute difference as an upper bound . . . 75
5.1.3 Using 0-1 variables to implement the upper bound 76
5.2 Exceptions to the requirement for both upper and lower bounds 77
5.3 Determining a suitable value for the large constant value M 78
5.3.1 Constraints 4.6 & 4.7 78
5.3.2 Constraints 4.8 & 4.9 79
5.3.3 Constraints 5.8 & 5.9 80
5.4 Dealing with differentiated execution units 82
5.5 Adding bounds to reduce computational time 84
5.6 Generating heuristic solutions 86

6 Parallelism in array boundary assertions **90**
6.1 Array boundaries and exploitation 90
6.2 Dangers of unauthorised loads and stores 93
6.3 Evaluating load/store addresses 94
6.4 Boundary assertions on the instruction level 95
6.5 Parallelism in boundary evaluations 97
6.6 Remaining drawbacks 98
6.7 Boundary check performance 99

7 Implementing an optimising compiler **101**
7.1 Data structures and representation 101
7.2 Describing the CPU 104
7.2.1 Description section 104
7.2.2 Registers section 105
7.2.3 Logical section 106
7.2.4 Physical section 107
7.3 The compiler core 108
7.3.1 DDG generation 110
7.4 Targeting and simplifier stages 111
7.5 Scheduling and assembly program generation 114
7.5.1 MILP generator 115
7.5.2 Solution processing 117

8 Experimental results **119**
8.1 Test results 120
8.2 Impact of solver options 123
8.3 Further work 126
8.4 Conclusions 127

A Demonstrating implicit parallelism **128**

B Boundary evaluation test cases **131**

C CPU definition file **132**

D Source code for the experimental results **139**

Bibliography **141**

Index **144**

UNIVERSITY
OF
JOHANNESBURG

Chapter 1

Introduction

Most of the improvements made to computer processors are in order to improve the performance of the processor. The usefulness of a computer is determined by the amount of time it requires to perform the given task, which is the execution of program blocks. Many problems are only limited by the time the computer requires to perform the task. Take for example weather prediction, the accuracy of a prediction is largely determined by the amount of computational time available. It is for this reason that performance is the paramount aspect of a computer.

UNIVERSITY
OF
JOHANNESBURG

In order to improve computer program performance two different approaches can be taken. Either the computer must execute the instructions in the program at greater speed or fewer/cheaper instructions, that will have the same end result, must be used in the program.

Increased performance through computer improvements is a runtime optimisation. Runtime optimisation is a grouping of optimisation techniques that are performed during the execution of the program. Improvements to the computer is a runtime optimisation because the performance gains are only available while using the faster computer. The opposite of runtime optimisation is optimisation during compilation which is where an optimisation is applied once during the compilation of the program.

1.1 Runtime optimisation

The performance of computer programs can be improved by reducing the time necessary for a group of instructions to execute. Improvements can either be through reducing the duration of individual instructions or by allowing multiple instructions to execute at the same time.

A significant advantage of improving performance in this way is that existing programs automatically benefit. Users do not have to update programs in any way in order to reap the benefits of improved performance. It can also hide increasing complexity from the developer since the complexity is kept within the processor.

1.2 Optimisation during compilation

The efficiency of computer programs has been improved over the years by improvements in compiler technology and improvements in application implementation (like using better algorithms). Improving programs through improvements in compiler technology is attractive in that it affects all programs being compiled. Additionally because these improvements are in software they are essentially a once off expense required during development of the program.

Compiler writers never want a processor to perform work at runtime that could have been performed at compile time. The reasoning is simple, if the work is performed during compilation then it only needs to be performed once, during the development of the program. This concept can be taken a step further by moving tasks traditionally associated with the processor to the compiler.

Scheduling of instructions and the allocation of processor resources (they are related tasks) are typically performed using a queueing (runtime) model by the processor as it executes the program. These tasks can be performed during compilation, greatly simplifying the processor that such programs run on. In addition, scheduling performed during compilation may yield better results than scheduling during execution (queueing).

Kästner and Langenbach [19] present techniques for finding good instruction schedules, during compilation, using integer linear programming. They report that it was effective even though their approach was limited to finding local optima.

1.3 Goals

The aim of the dissertation is to present a selection of algorithms and methods for the compilation of programs for explicitly parallel processors. The primary area of focus is the scheduling of instructions in a resource limited environment through the use of mathematical programming especially discrete optimisation techniques. Resource limitation significantly complicates scheduling, however it is necessary in order to produce instruction schedules for real world computers. The emphasis is on finding schedules with the shortest possible execution times, other factors such as code size are not considered.

An overview of computer architecture, with an emphasis on instruction level parallelism, is included. It is present because it necessary for the reader to understand the restrictions that computer programs operate within. It also serves to inform the reader about runtime scheduling and its relative complexity.

Similar treatment is given to the topic of compiler construction. It is necessary for the reader to understand how a program is processed and represented by a compiler. The compilers representation of a program undergoing compilation is used extensively in several chapters.

A critical aspect for the effectiveness of the techniques presented is that there is sufficient parallelism in the program undergoing compilation. A technique for increasing the parallelism in certain cases is introduced. The technique is an example of how parallelism can sometimes be found even if it is not immediately apparent.

The techniques introduced and discussed will be employed in a compiler with the intention of providing experimental results. A key benefit in acquiring experimental results is information regarding computational complexity. Practical issues regarding implementation will also be exposed.

1.4 Chapter overview

Chapter 2. Computer architecture
The second chapter discusses computer architecture of modern processors from a performance point of view. Issues like data buses, caching and memory paging are also covered in a supporting role. The latter half of the computer architecture chapter focuses

on instruction level parallelism. Both approaches to instruction parallelism, implicit and explicit, are covered.

Implicit parallelism is covered for basic principles of instruction level parallelism. Explicit parallelism is discussed because the optimisation methods introduced target processors using this approach. The topic is covered in some detail because of the effect of computer architecture on the creation of optimisation models.

Chapter 3. Compilers
The third chapter gives a brief description of the steps taken by a compiler to convert a textual source code program to a directed graph that is used by the optimisation stage. Some additional topics are covered that do not relate directly to the creation of a directed graph. Only techniques used for the implemented compiler are discussed in this chapter therefore it does not replace authoritative texts on the subject. Very little information regarding parsing is given in this chapter due to the use of compiler construction tools (which are briefly mentioned).

Chapter 4. Code optimisation through instruction scheduling
This chapter focuses on the use of mathematical programming techniques for finding an optimal schedule for the instructions that together form a program. The program is described in the form of a dependency graph from which a mathematical program must be generated. Additional information is used regarding the target processor. Efficiency plays an important role in the techniques introduced because of the computational difficulty of solving mathematical programs. The model introduced in this chapter neglects several implementation issues which are resolved in the following chapter.

UNIVERSITY
JOHANNESBURG

Chapter 5. Techniques and tools to aid implementation
This chapter deals with barriers preventing the implementation of the model introduced in the previous chapter. An additional area of focus in this chapter is the reduction in computational time required to solve the resulting mathematical program. A fast heuristic algorithm is introduced as part of reducing the computational time required. However, the heuristic algorithm can also be used to generate schedules without the use of mathematical programming.

Chapter 6. Improving parallelism in assertions
An analysis of parallelism in respect to array boundary assertions takes place in this chapter. Array boundary assertions have significant importance with regards to security and application correctness. The methods introduced improve the parallelism of such asser-

tions and thereby potentially improve the performance of the program. The limitations of these methods are also discussed especially because the introduced methods are only applicable to certain types of processors (but should be possible on all EPIC processors).

Chapter 7. Implementation of a compiler and optimiser
The compiler implemented to demonstrate the introduced techniques is documented in this chapter. Information regarding the structure and inner workings of the compiler is available as well as some configuration information. The most important configuration information is the format of the processor definition file which is covered in full detail. The compiler implements a subset of the C programming language.

Chapter 8. Experimental results
Experimental results produced by applying the compiler and optimiser to several small C programs are introduced and discussed. An emphasis is placed on Information regarding optimal schedules, heuristic schedules and computational difficulty. Included in the discussion is the impact of the solver configuration on the computational requirements. The chapter ends with a short conclusion

UNIVERSITY
OF
JOHANNESBURG

Chapter 2

Computer architecture

Computer processors are found in a wide variety of applications, therefore the requirements for a computer will vary greatly. Key factors in processors include power requirements, cost and performance. It is not practical to have a processor that matches all of these requirements so tradeoffs have to be made. In practice a particular processor will focus on a single aspect. Processors that focus on performance are the target of the material in this dissertation hence the information regarding computers is biased in that direction.

UNIVERSITY
OF
JOHANNESBURG

2.1 Introduction to the components of a computer

Every modern computer has three key components. An input/output or I/O system to move data in and out of the processor. Memory in which to store both instructions and data for the computer. And finally a processor that executes instructions to process data. This basic architecture of a computer is referred to as the Von Neumann architecture, it is discussed in detail by Cragon [10].

When the Von Neumann architecture was first introduced most computers used the Harvard architecture [10]. The main difference between the Von Neumann and Harvard architectures is that in the Von Neumann architecture instructions and data are stored together while in the Harvard architecture data and instructions are stored separately. Storing the instructions and data together has many advantages including simplified design of the processor. It also allows for modern operating environments because the operating system can treat programs as data and therefore alter it if necessary. However, the Harvard archi-

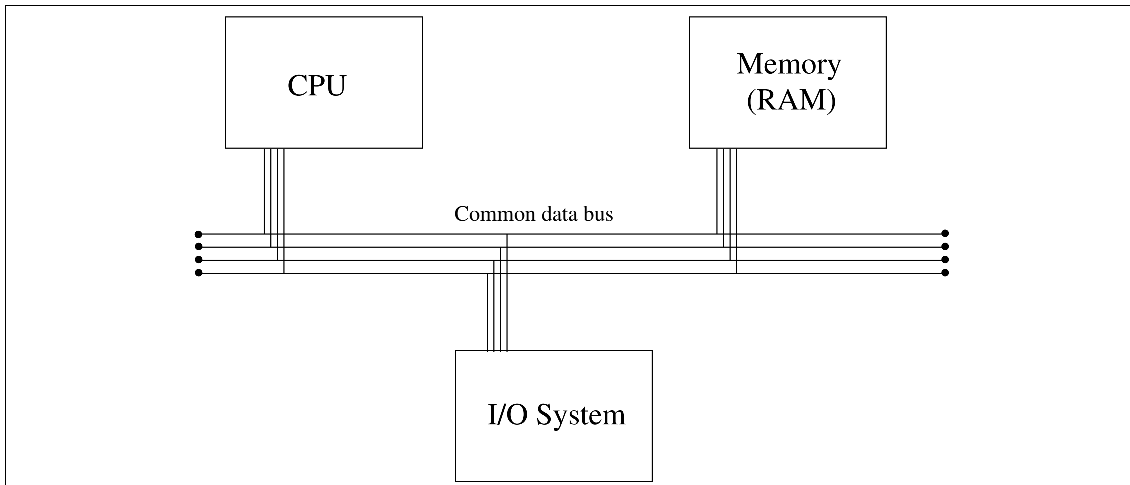


Figure 2.1: *The system bus and the devices that are connected to it.*

tecture has not completely disappeared, the concept of separate storage for instructions and data has resurfaced in cache designs (discussed in a later section).

In practice the three components in the computer have to be joined by a *bus* that facilitates the movement of data from one component to another. The bus of the computer is becoming more and more important as both the quantity of data increases as well as the ability of the processor to process data at greater speed. Figure 2.1 illustrates the role that the bus plays connecting all three of Von Neumann's computer components. The usefulness of a unified instruction and data memory is enhanced by the requirement of a single bus for both instruction memory and data memory.

UNIVERSITY
JOHANNESBURG

2.2 Instructions as an interface to the processor

Each instruction tells the processor to perform a single operation. For example an instruction could tell the processor to add two numbers together or alter the behaviour of the processor in some way. The executable component of a computer program (it may also have static data) is just a collection of instructions. The order in which the instructions must execute is very significant, therefore instructions are executed one after another as they appear in the memory of the computer. The processor has an internal register that keeps track of which instruction is executing by recording its memory address and incrementing that address each time an instruction executes. The register that records the address of the executing instruction is called the *instruction pointer* or *instruction counter*.

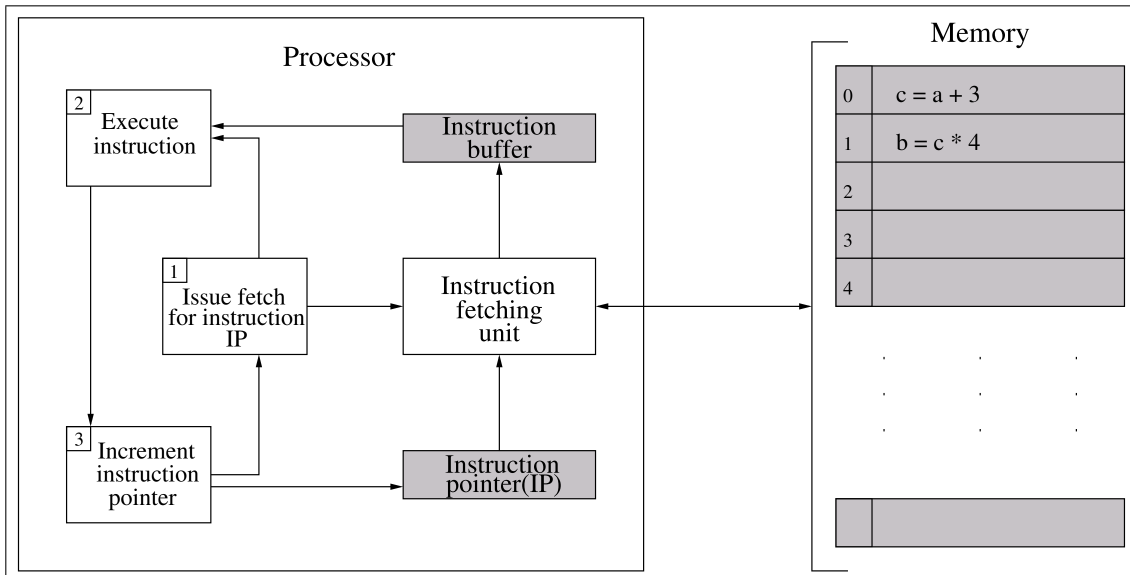


Figure 2.2: *This figure illustrates the relationship between instruction fetching and the instruction pointer. The three numbered blocks on the left represent the three different phases in each cycle. Blocks with a shaded background represent storage, including the memory which contains two instructions.*

Suppose we wanted a program that evaluates the expression $b = (a + 3) * 4$, we would have to break this into two or more instructions on most processors. The two processor instructions might be something like $c = a+3$ and $b = c*4$, assuming that c is a temporary variable. Figure 2.2 illustrates the role of the instruction pointer in the execution cycle in conjunction with Table 2.1 which lists the steps taken for the two instructions in the example.

Instruction pointer (IP)	Phases of execution	Instruction buffer
0	(1) Fetch	
0	(2) Execute	$c = a + 3$
0	(3) Increment	$c = a + 3$
1	(1) Fetch	$c = a + 3$
1	(2) Execute	$b = c * 4$
1	(3) Increment	$b = c * 4$
2	(1) Fetch	...

Table 2.1: *The list of steps taken in the cycle shown in Figure 2.2.*

According to Von Neumann's model there are three basic types of instructions that are required to implement a fully programmable computer.

- Instructions that copy data to and from the memory, I/O devices and the work space (space for temporary variables, etc) of the processor. In many processor architectures instructions cannot directly access the RAM (random access memory) of the

computer and therefore a pair of specialised data movement instructions are used to copy values between memory and the work space of the processor. A *store* instruction is used to write a value from the work space to memory. Similarly a *load* instruction reads a value from memory into the work space. Instructions used with hard-coded constants called *immediates* are sometimes associated with data movement instructions because they copy data from memory into the work space of the processor. However, unlike a load instruction the data is encoded in the instruction itself.

- *Arithmetic logic unit* (ALU) instructions perform mathematical operations such as addition and multiplication as well as boolean operations. Comparison instructions (relational operators), like equal to and greater than, also fall into this category. The range of ALU instructions offered varies widely from architecture to architecture and is often dependent on the workloads that the processor is intended for.

- Instructions that alter the flow of the program. When discussing the instruction pointer the point was made that instructions execute one after another in the order that they appear in memory. If this situation remained unchanged then each instruction would only execute once in the execution of the program. Core programming elements like loops or conditional language elements would be impossible to implement. To allow instructions to be repeated or executed conditionally it is necessary to change the value of the instruction pointer during execution. For example if the 10th instruction changes the value of the instruction pointer to 2 then all of the instructions from 2 to 10 will execute again and again, creating a loop. This sort of instruction is called an *unconditional branch* or more commonly a *jump*. If the value of the instruction pointer is changed only if a specified condition evaluates to true then it is called a *conditional branch*, or often it is just called a *branch*.

Implementing a work space for temporary values is more complex than it appears at first glance. The performance of work space access is critical to the performance of the processor because the work space is a hot spot of activity. As a result several different approaches have been taken for implementation of a suitable work space for the processor. The approach taken for implementing a work space will have a significant effect on the representation or structure of an instruction.

2.3 Approaches to work space implementation

Storing all data outside the processor in main memory, as shown in Figure 2.3, is elegant, however it has two major drawbacks. The first drawback is *latency*. Latency is the quantity of time between an action being initiated and its completion. When referring to memory access, latency is the amount of time between the processor requesting data from memory and actually getting it. Figure 2.1 illustrates the shared nature of the system bus which is inherently slower than the internal busses of the processor. The second drawback is that a comparatively large number of bits are required for memory addresses and therefore would drastically increase the size of the program. For example most desktop or server processors use a 32 bit or 64 bit memory address space. A single instruction may require more than one address, therefore the total number of bits required just for addresses would be huge. There are ways to reduce the size of addresses in specific cases, however the problem of latency remains. Since there is no practical way of solving the latency problem, RAM as a work space is not an option.

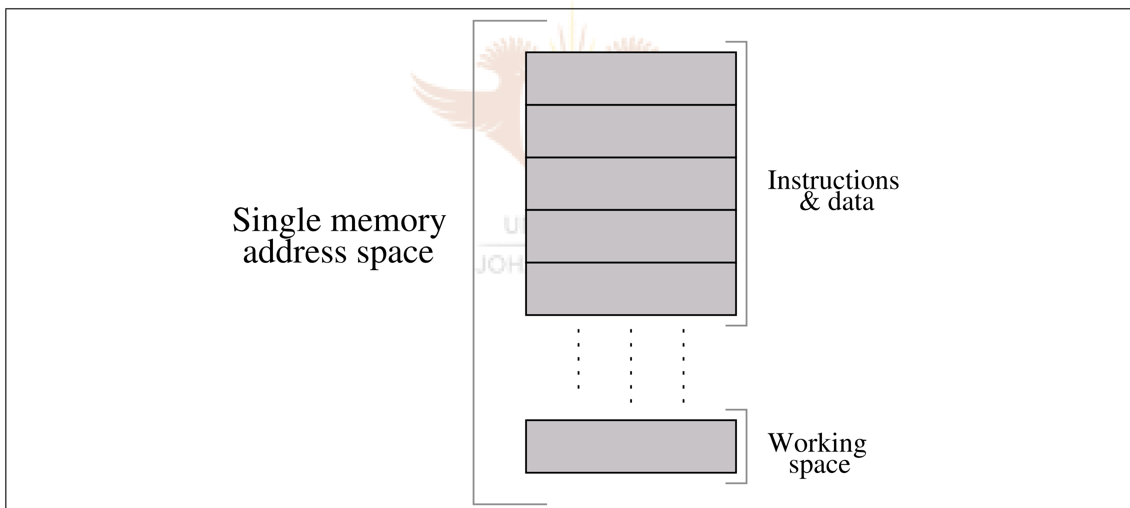


Figure 2.3: *A single unified address space for both the data/instruction area as well as the work space.*

The main problem in using general memory as a work space is the problem of latency. A major contributor to the latency problem is that the RAM is external to the processor. Therefore the logical move is to place some memory on the processor itself with a dedicated internal bus. To further reduce the latency the processor memory could be static RAM (or SRAM for short) which has a far lower latency than dynamic RAM (DRAM is the memory technology generally used to implement the main external memory). This move will alleviate the problem of latency, however there is still the problem of a large address space. Because this on-chip memory can only be accessed by parts of the pro-

cessor, it can have its own addressing scheme as shown in Figure 2.4. This reduces the problem of large addresses in instructions. The disadvantage of on-chip memory is that space is at a premium on the processor, so very little memory can be placed on the processor. Having a large amount of on-chip memory would also require the use of large addresses in order to be able to access all the registers. Registers is the term given to on-chip memory that has been divided into non-overlapping cells that can be accessed simultaneously. For example the PowerPC architecture has 64 general purpose registers. To address each register independently a 6 bit address is required since $64 = 2^6$. If the register address space were integrated into the computer's main memory address space then addresses would need to be 32 bit or 64 bit addresses instead.

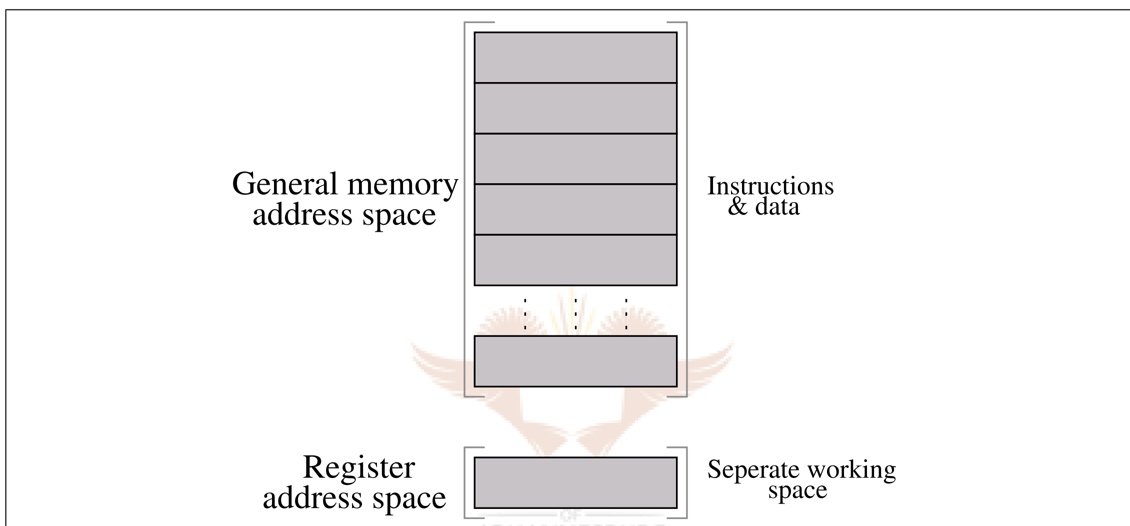


Figure 2.4: *Separate address spaces for the data/instruction area and the work space.*

The downside of SRAM is that a single bit of SRAM requires far more components than a bit of DRAM. The large on-chip space requirement means that only a very small quantity of SRAM can be provided. Architectures that use this approach are called general purpose register architectures.

An *accumulator* is a specialised type of register, the name comes from its purpose which is to accumulate the result of an instruction. Accumulator architectures have a single general purpose register called an accumulator. The IA-32 architecture (also known as the x86 architecture) is a hybrid of an accumulator architecture and a general purpose register architecture. This sort of hybrid is called an extended accumulator architecture and is described by Patterson and Hennessy [25]. Registers in extended accumulator architectures are bound by all sorts of restrictions in regard to what instructions they can be used with and in what sort of role.

Take for example an integer multiply instruction on an IA-32 processor. It must use the AX register as one of the two multiplicands and the result will be stored in a combination of the DX and AX registers [18]. The lack of flexibility inherent in accumulator architectures has made it unpopular in modern processor designs. It was originally more popular when transistors were at a premium because a simple accumulator architecture requires a few less transistors than a general purpose register architecture.

An underlying problem with both general purpose register and accumulator architectures is that they have a fixed amount of storage. If a program needs a larger work space than the processor can provide then it must fall back to some other mechanism like using main memory. The process of deciding which variables to assign to registers or accumulators and which to assign to memory is called *spilling*. If a variable is not assigned to a register then it has been *spilled*. Choosing which variables to spill is a difficult problem on general purpose register processors and even more so on extended accumulator processors.

Another technique, still popular in virtual machines, is not to have any registers at all (with one exception). A stack data structure that resides in memory is used instead. On such architectures there are three different ways of reading or writing data, the first is a *push* which writes an element to the top of the stack and a *pop* which reads and removes an element from the top of the stack. The third method is to load and store to elements on the stack relative to an address stored in a specialised register called a *base pointer*. The number of elements on the stack does not change when using a load or store as opposed to a push or pop.

UNIVERSITY
JOHANNESBURG

Stacks are popular on general purpose register architectures as well as accumulator (extended or otherwise) architectures. On these architectures the stack is used as a work area to spill variables into when there are not enough registers available. Base pointer relative addressing is when a specified address is added to the value that resides in the base pointer register to produce a final memory address. By using base pointer relative addressing, addresses in instructions are kept small, unlike addressing memory directly with absolute addresses. A side effect of using a stack based work space is that memory management is greatly simplified when elements can only be added or remove from the top of the stack.

2.4 Components of a modern processor

An *execution unit* is a functional unit of the processor that performs the actual execution of an instruction. A single execution unit may support multiple types of instructions, however the execution unit can only execute one instruction at a time, regardless of type. Multiple identical execution units may be present so that multiple instructions can be executed in parallel.

Modern operating systems want to prevent different applications from interacting with one another without the operating systems' permission. To prevent applications from being able to access one another's code or data the concept of a *segment* or *memory space* was introduced. Each application executes within a unique data segment which means that the application can only access its own data. Addressing is relative to the segment so it appears to an application that it is the only application executing on the computer and therefore it cannot even address another application's memory space. It is through the segment feature that applications are sand-boxed, or in other words prevented from affecting other applications.

Mapping pages (or frames) of physical memory to segments (called logical memory) is usually done through the use of a paging mechanism. The physical memory of the computer is divided up into pages of a fixed size, for example 8 Kb. A lookup table within the RAM of the computer, called a page map, keeps track of which physical pages are mapped to the current logical memory space and includes information about the page. Performance would be problematic if the processor had to consult the page map during every memory access. Part of the page map is cached within the processor to bring performance to an acceptable level. This cache and the supporting mechanism is called a *translation look-aside buffer* or TLB. If information regarding a needed page is not present in the TLB then the TLB is updated with information from the page map, this update is called a *table walk*. The use of pages has many advantages for the operating system discussed in detail by Patterson and Hennessy [25].

Permissions for specific types of access can be imposed on individual physical pages. Depending on the processor family, it is usually possible to specify if the application can read, write or execute data on a per page basis. The permissions regarding the page are referred to as *page rights*. Page rights are part of the information given for each page that was mentioned in the preceeding paragraph.

The component of the operating system that loads an application will load the instructions

and the data separately. It can then specify that the application may only read and execute instructions by toggling the permissions for the page map entries that corresponds to the page that the instructions are stored in. Pages containing application data may only be read from or written to since applications only execute instructions from the instruction pages. If an application attempts to perform a disallowed action then the operating system will terminate the application. This behaviour is exploited in Chapter 6 which deals with parallelism in terms of buffer boundary checks.

Fast access to data is critical for a processor to realize its performance potential. If the processor had to perform a round trip to RAM every time a memory address was accessed it would become a major bottle-neck in terms of program performance. In order to reduce the round trip time (latency), caches were added to facilitate fast lookups of commonly accessed addresses. The principle behind a cache is to duplicate some data from RAM in a fast yet small memory space within or near the processor. Information regarding the details of cache operation can be found in Patterson and Hennessy [25].

Multiple caches are layered with the caches nearer to the processor's computational section being smaller than caches further away. Figure 2.5 illustrates a processor with two levels of cache (Cache L1 and Cache L2) as well as external RAM. The read latency increases for every level in the memory hierarchy.

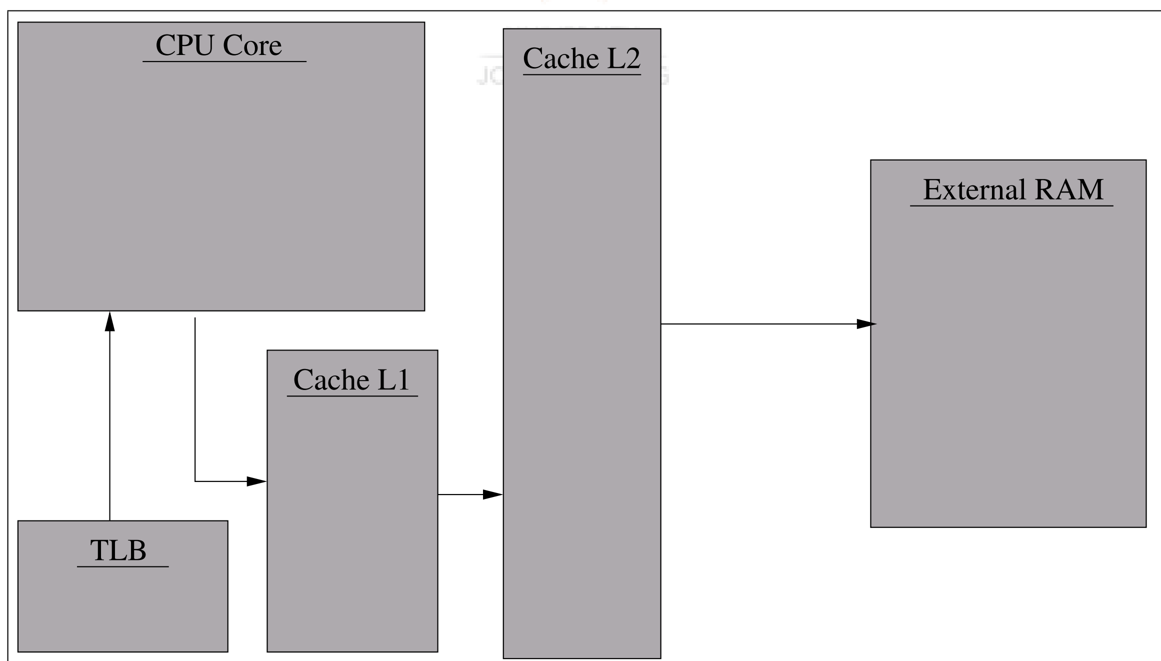


Figure 2.5: *An example of a computer's memory systems with two layers of cache between the processor and the external RAM. Notice that the L2 cache is larger than the L1 cache.*

For example, a value needs to be read from memory. The MMU (*memory management*

unit) will issue a request to the L1 (level 1) cache, if the address is not cached then the L1 cache will pass the request on to the L2 cache. If the L2 cache has a copy of the data requested, it will return the data without checking the external RAM. If the data was not found then the request would be passed on. In practice there can be more than three levels of cache.

Not only do caches retain local copies of data they also *prefetch* data. Prefetching takes advantage of the temporal locality of memory reads by reading more data than just the data requested. The additional data is also stored in the cache because the likelihood of the additional data being requested from the same region is very high.

In practice the cache nearest the processor core is split into two halves, each half has a separate controller. One half is used to cache instructions and the other half is used to cache data. The processor accesses instructions and data separately and has different implementation requirements hence the use of the split approach. RAM cached by the data cache can be both read from or written to. Memory cached by the instruction cache can only be read from, since the program cannot change its instructions at run time. A processor that employs a separate instruction cache and data cache is referred to as using a Harvard architecture cache.

UNIVERSITY JOHANNESBURG

Table 2.2 lists several instructions that are used in examples in the remainder of this chapter along with their descriptions. All of the instructions (except NOP) have one or more operands. Registers begin with the letter r, offsets are constant values and addresses are label references.

Instruction	Description
MOV $rsrc$, $rdest$	Copy value $rsrc$ to $rdest$.
MOV offset($raddress$), $rdest$	Copy value from memory address $raddress+$ offset to $rdest$. The value specified by offset is added to $raddress$ to produce a final address.
MUL $rsrc1$, $rsrc2$, $rdest$	Multiplies $rsrc1$ and $rsrc2$ leaving the result in $rdest$.
ADD $rsrc1$, $rsrc2$, $rdest$	Adds $rsrc1$ and $rsrc2$ leaving the result in $rdest$.
XOR $rsrc1$, $rsrc2$, $rdest$	Performs exclusive-or of $rsrc1$ and $rsrc2$ leaving the result in $rdest$.
BRANCH $rcondition$, $address$	If $rcondition \neq 0$ then jump to $address$.
NOP	No operation.

Table 2.2: *A list of basic instructions, with descriptions, used in examples in this chapter. Each row starts with the instruction name followed by its operands and a description. Both MOV instructions use a load/store execution unit (LSU) and the MUL, ADD and XOR instructions use an ALU execution unit.*

Some architectures employ a translation layer that converts each program instruction into one or more internal processor instructions. In the processor there is a table containing internal instructions for each program instruction. The content of this table is called *microcode* hence processors that employ this technique are microcode processors. Microcode has a significant effect on parallelism and is discussed in the relevant sections.

2.5 Superscalar processing

In an effort to improve performance of computer programs, parallelism has been implemented at an instruction level. *Instruction level parallelism* (ILP) attempts to execute instructions in parallel and therefore reduce the amount of time required to execute the program code. Processors that execute multiple instructions simultaneously are called *superscalar* processors.

Processors implementing parallelism can be divided into two separate categories, either SIMD or MIMD (definitions for various parallelism models can be found in [30]). SIMD stands for *single instruction multiple data* which means that a single instruction can perform the same operation on multiple unrelated pieces of data. MIMD, *multiple instruction multiple data*, is where multiple different instructions can execute in parallel. MIMD is the more flexible of the two approaches since it can exploit the same parallelism possibilities as SIMD as well as others that SIMD cannot.

UNIVERSITY
JOHANNESBURG

From an instruction set perspective, methods for implementing instruction level parallelism fall into two categories, implicit parallelism and explicit parallelism. In the first approach the processor finds the parallelism in the program unlike the second approach where parallelism information is encoded into the instruction.

2.6 Implicitly parallel instruction computing

Implicitly parallel instruction sets have the advantage that parallelism features can be added to a new processor without the need to change the instruction set. Therefore existing programs will execute on the newer processors and will also benefit from improved performance. The IA-32 architecture is an example of a processor family that originally did not support instruction level parallelism. Parallelism features were added to later

models resulting in significant performance improvement without the need to recompile applications. Patterson and Hennessy discuss superscalar architecture and implicitly parallel instruction computing in detail [25].

2.6.1 Runtime scheduling/queueing

When a program executes on a non-superscalar processor each instruction starts executing when the previous instruction stops executing. In a superscalar processor the execution of instructions overlap, therefore an instruction can start executing before the previous instruction has stopped executing. Once the processor has decided that an instruction may start executing it assigns the instruction to an execution unit that is not currently in use. As stated earlier each execution unit can only support certain types of instructions, therefore the processor must match the instruction to a suitable execution unit.

In general for an instruction to begin execution the following conditions must be met.

1. An execution unit capable of executing the instruction must be available, or in other words, not currently in use. This is fairly logical since each execution unit can only execute one instruction at a time.

UNIVERSITY OF JOHANNESBURG

2. The data inputs and outputs of the instruction being inspected in conjunction with the instructions currently executing must comply with Bernstein's conditions [30]. Each of Bernstein's conditions must be satisfied if it is safe for two instructions to execute in parallel. The instruction being inspected is checked against each instruction executing. If a single condition is not met then parallel execution cannot take place. Let instruction i be the instruction being inspected and instruction j an instruction currently executing. The three conditions are listed below where I_i is a set representing the input of instruction i and O_i is a set representing the output of instruction i.

   (a) $I_i \cap O_j = \emptyset$, the input of instruction i must not intersect with the output of instruction j. In this condition is not satisfied then instruction i is dependent on an output of instruction j.

   (b) $I_j \cap O_i = \emptyset$, the input of instruction j must not intersect with the output of instruction i. This circumstance is impossible in a queue because it would mean that a currently executing instruction is dependent on an instruction that has

not yet executed. The order ensures that instruction j will execute before instruction i and therefore instruction j cannot be dependent on instruction i. In a later section we will look at out-of-order execution where this circumstance is possible because, as the name suggests, the instruction order is no longer guaranteed.

(c) $O_i \cap O_j = \emptyset$, the output of instruction i must not intersect with the output of instruction j. If the output of an instruction will overwrite the output of another instruction then the order of the instructions is important. It is necessary to ensure that once both instructions have completed execution, the intended output remains, therefore the pair of instructions are order dependent.

3. The processor must be certain that the instruction being inspected will execute. The processor cannot determine which instructions will follow a branch instruction until the branch instruction has completed execution. Hence no instructions past the branch can be executed.

4. Certain types of instructions have specialised or complex behaviour and therefore are only safe to execute after all of the currently executing instructions have finished executing. These instructions vary widely from processor to processor as they are usually involved with management tasks or communicating with hardware. If the instruction being inspected is one of these instructions then no parallelism or limited parallelism may be implemented.

UNIVERSITY OF JOHANNESBURG

A simple example program is given in Table 2.3. Each instruction executes in the given order to comply with Bernstein's conditions when checking candidate instructions against currently executing instructions. In this example some parallelism is possible.

1	MOV	$0(r1)$,	$r2$	
2	MOV	$4(r2)$,	$r3$	
3	MUL	$r2$,	$r4$,	$r5$
4	ADD	$r3$,	$r5$,	$r6$
5	XOR	$r2$,	$r4$,	$r7$

Table 2.3: *Register $r1$ and register $r4$ have predefined values, the other registers are uninitialised.*

In Figure 2.6, which corresponds to Table 2.3, instruction 2 and instruction 3 can execute in parallel however instruction 4 is dependent on the output of instruction 3 and therefore can not execute in parallel. In this situation the processor waits until the output from instruction 3 is available before attempting to execute new instructions. Instruction 2 executing at the same time as instruction 3 does not change the order in which the instructions

execute. Note that instruction 5 is only dependent on instruction 1 however it cannot execute in parallel with instruction 2 and instruction 3. Execution of this instruction was prevented due to the requirement that all instructions execute in the order that they appear in the program.

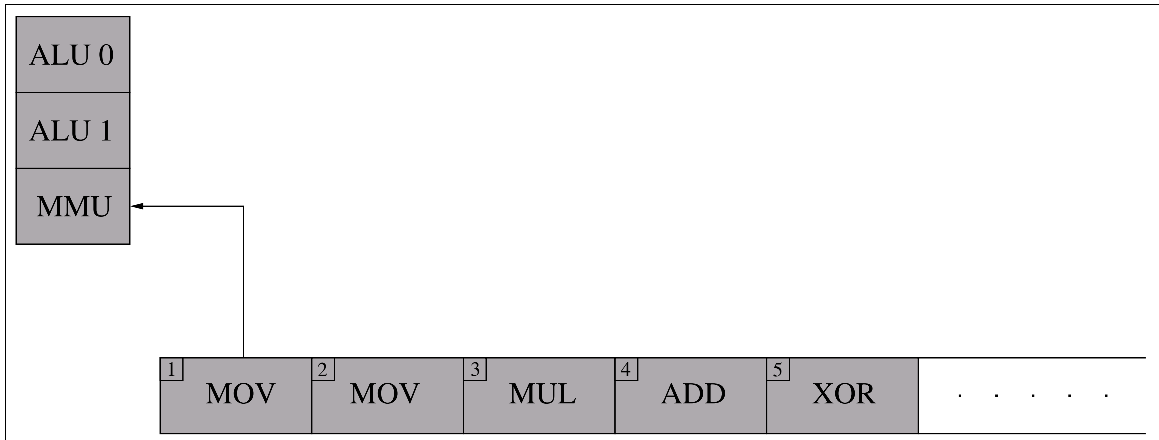


Figure 2.6: *This figure shows the first instruction of the program given in Table 2.3 executing. The blocks on the left are execution units while the numbered blocks on the bottom are instructions. Instruction 2 cannot execute because it is dependent on instruction 1. Even if instruction 2 was not dependent on instruction 1 parallel execution could not take place because there is no free MMU execution unit available.*

Figure 2.7 shows the execution of the program given in Table 2.3 after instruction 1 has completed execution. This allows instruction 2 and instruction 3 to execute because the dependence has been satisfied without changing their order. Instruction 4 cannot execute because it is dependent on the output of instruction 2.

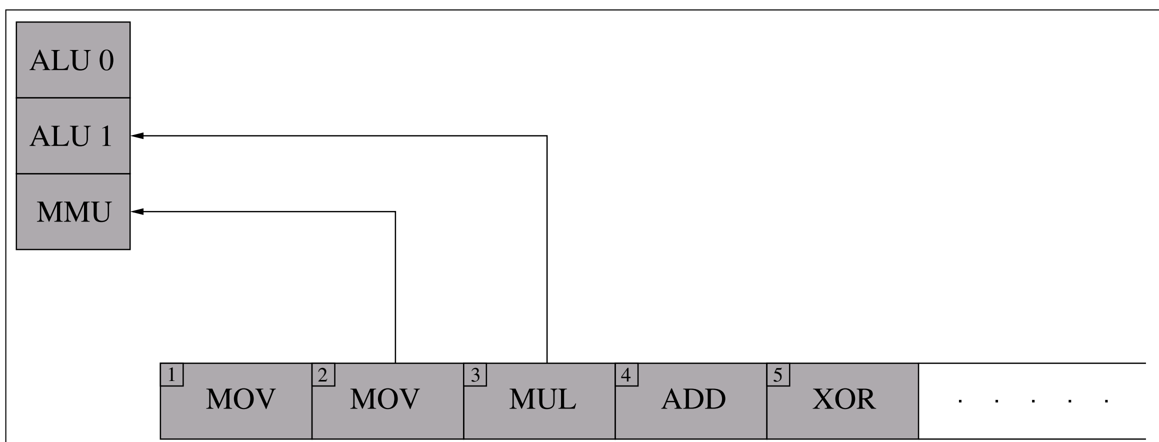


Figure 2.7: *Both instruction 2 and instruction 3 executing in parallel. Instruction 4 is dependent on instruction 2 and therefore cannot execute even though an execution unit of the correct type is present and available.*

2.6.2 Out-of-order execution

As demonstrated in Figure 2.7 the order of the instructions can have a significant effect on the exploitable parallelism of a program. The more obvious solution would be to ensure that the compiler that generates the program instructions determine the best possible order. In practice there are several reasons why instruction ordering in programs may not be optimal.

1. Some processor architectures currently in use were developed before superscalar principles became viable. Compilers for these processors simply could not take superscalar execution into account. Fortunately most programs can simply be recompiled with an up to date compiler targeting the processor in question.

2. The exact duration of some instructions on some processors cannot be determined at compile time therefore affecting the accuracy of proposed schedules. Microcode interpreting processors (like IA-32) are particularly bad in this respect because the internal instructions executed can vary for the same high-level instruction. Some instructions also have *opt-outs* or in other words some instructions can take short cuts in execution resulting in a shorter execution duration. In general, RISC processors have relatively short duration instructions that are very consistent in their execution duration in order to maximise the superscalar behaviour of the processor. It should be noted that most processors have at least one instruction that has a variable duration. The duration of a load instruction can vary depending on how *far* the data is from the processor core. For example the duration of a load instruction increases for each step through the memory/cache layers, i.e. from Cache L1 to Cache L2 to RAM. The duration required to fetch data from RAM cannot be accurately predicted on many computers. However in practice statistical averages for load instructions have proven to be fairly reliable, especially in programs where there is a fair degree of parallelism [25].

3. On microcode translating processors each program instruction will be translated into one or more of the processor's internal instructions. Each of these micro instructions (sometimes called *micro-ops* or μops) are executed separately, therefore a single program (non-micro-op) instruction may use multiple execution units. In these circumstances the compiler has limited control over the execution unit assignment compounded by the fact that processor manufacturers do not, in general, release information regarding their microcode.

UNIVERSITY OF JOHANNESBURG

4. Finding a good schedule is fairly easy, however finding the optimal schedule is very difficult. Chapter 4 focuses on finding optimal instruction schedules and Chapter 5 includes a simple heuristic (among other things) for finding good but not necessarily optimal schedules.

In Figure 2.7 an opportunity to improve parallelism was not exploited. If we switch the order in which instruction 4 and instruction 5 (in Table 2.3) execute then more parallelism will be possible. This is possible because the instruction dependencies allow the instructions to be swapped and an execution unit is available to execute the instruction in the new schedule. That is to say the level of parallelism can be improved by altering the order in which the instructions execute. Since the instructions will no longer execute in the specified order this approach is called *out-of-order* execution.

Instead of changing the order of the instructions in the stream the processor changes the order in which it processes the instructions. Hence the name out-of-order execution. If an execution unit is not in use, then the processor looks ahead in the instruction list for a suitable instruction skipping non-executable instructions. Determining if an instruction is suitable on an out-of-order processor is very complex on the hardware level, however it is a fairly simple extension to the conditions presented for in-order processors.

In Section 2.6.1 the conditions for execution of the next instruction, on an in-order processor, were introduced. One of those conditions was that another set of conditions, Bernstein's conditions, also had to be satisfied. Each of Bernstein's conditions checked the instruction under inspection (instruction i) against each instruction that was executing at the time (instruction j). On out-of-order processors the instruction under inspection must be checked against each instruction currently executing as well as each instruction that was skipped. Checking instruction i against skipped instructions prevents instruction i from executing if it is dependent on a skipped instruction.

Figure 2.8 illustrates the program segment given in Table 2.3 executing on an out-of-order processor. In this example instruction 5 executes in parallel with instruction 2 and instruction 3, unlike the in-order example shown in Figure 2.7.

Although out-of-order execution is fairly simple on a conceptual level it is notoriously difficult to implement in hardware. Studies on different parallelism models for embedded processors have also linked out-of-order execution and, to a lesser extent, implicit parallelism to high power consumption. A review of several different processes implementing a variety of parallelism techniques is presented by Kozyrakis and Patterson [21].

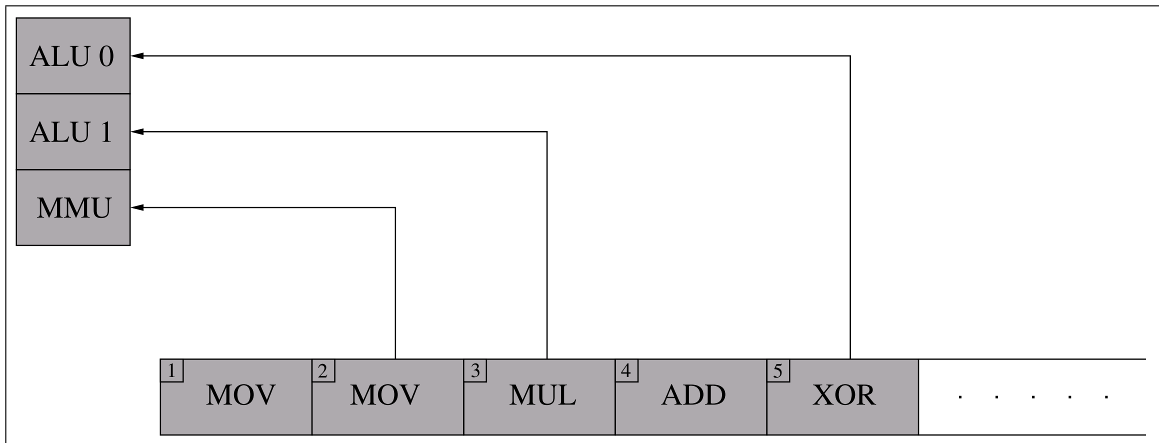


Figure 2.8: *This example demonstrates the same program executing as in Figure 2.7 except that the processor implements out-of-order execution. This allows instruction 5 to execute in parallel with instruction 2 and instruction 3.*

Differences exist between out-of-order implementations based on how far ahead the processor will look for instructions. For example the Pentium-4 implementation of the IA-32 architecture can look ahead by more than a hundred instructions (described in an Intel document [8]). Large scale out-of-order implementations are limited by the frequency of conditional branch instructions. Remember (from Section 2.6.1) that as soon as a branch instruction is encountered no more instructions are processed because the result of the branch is not yet known. Therefore in order to maximise parallelism some mechanism to extend parallelism beyond branches is required.

UNIVERSITY OF JOHANNESBURG

2.6.3 Branch prediction

In order to extend parallelism beyond branch instructions the processor attempts to guess which direction a branch instruction will take. The first difficulty in utilising branch prediction is that a percentage of the predictions will be wrong. If a prediction is wrong then the effects of the instructions following the incorrectly predicted branch must be cancelled once the error has been discovered. Branch prediction is part of a group of techniques referred to as *speculative execution*. Speculative refers to the aspect that they all have in common which is that they all speculate which instructions will execute.

Predictions regarding branch instructions are derived through the use of heuristic learning functions which are updated each time a branch instruction completes execution. Only once a branch instruction has completed execution is the real (not predicted) action of the branch known. A small quantity of information is stored regarding the behaviour of a

branch after it has been encountered for the first time. Since the space available to track branch instructions is limited, information regarding branch instructions will be replaced from time to time. The process of branch information replacement is in some ways similar to memory page replacement. Disposable information replacement is focused on finding which entry to discard to make space for a new entry in limited storage situations. Several different criteria can be used to make a decision with varying results.

1	MOV	$0(r1)$,	$r2$	
2	MOV	$4(r2)$,	$r3$	
3	MUL	$r2$,	$r4$,	$r2$
4	BRANCH	$r4$,	jump_destination	
5	XOR	$r2$,	$r3$,	$r2$
	jump_destination:			
6	ADD	$r2$,	$r3$,	$r2$

Table 2.4: *An example program similar to the one given in Table 2.3 except that it includes a branch instruction.*

Table 2.4 introduces a program where a branch instruction is present and two different instructions that can execute depending on the outcome of the branch. Figure 2.9 illustrates the execution unit loading problem when a branch instruction is encountered as there are two possible instruction streams. The branch prediction system of the processor would, using a heuristic algorithm, decide which instruction to execute (which is either instruction 5 or instruction 6). If, for example, it predicted that the branch condition would fail then it will execute instruction 5. If this prediction was incorrect then it would have to reverse the result of instruction 5 and start executing instruction 6 instead.

Branch prediction can greatly improve the parallelism of a program, however its effectiveness is tightly linked to the accuracy of the heuristic function used. Some branches cannot be predicted and in some cases the branch prediction algorithm may even be counter productive in that it may generally make the wrong decision.

A very simple prediction algorithm would simply repeat the last action that a branch took. This heuristic algorithm would work well in many circumstances. For example the branch instruction at the end of a for loop will jump on all iterations except for the last. If the correct action to be taken alternates on each iteration then this heuristic algorithm would always make the wrong choice. In practice the first case is far more common, therefore branch prediction is generally very useful.

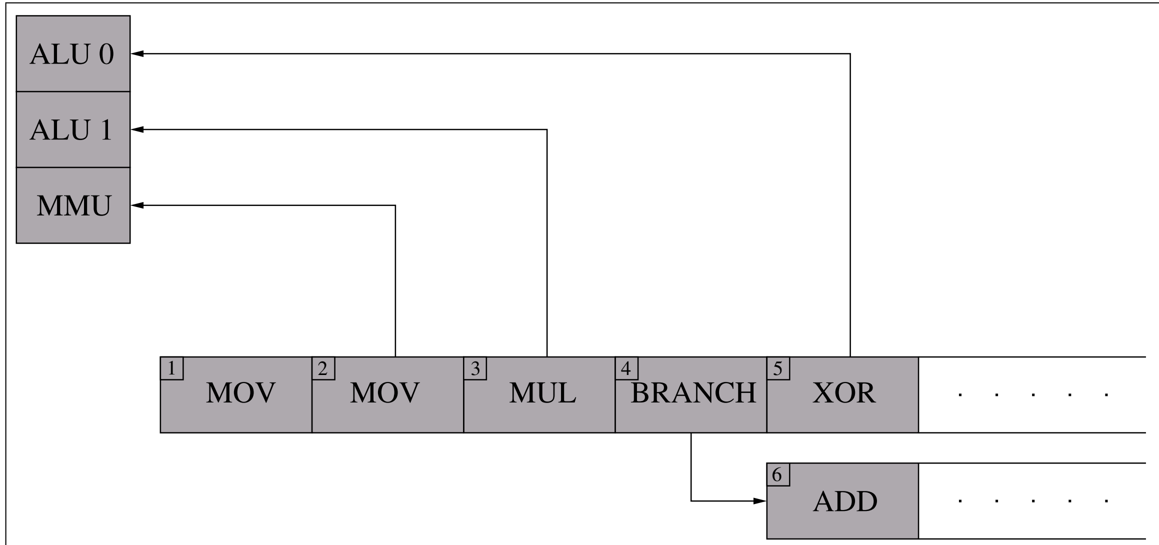


Figure 2.9: *The processor predicts which direction the branch instruction will take and executes instruction 5 in parallel with instruction 2 and instruction 3.*

2.6.4 Implicit parallelism in practice

To illustrate the effectiveness of instruction level parallelism on implicitly parallel processors several experiments can be performed. If two loop bodies contain the same number and type of instructions logic dictates that they will require a similar amount of time to execute. On non-superscalar processors this may be true, however on processors that implement parallelism the dependencies between the instructions play a big role.

Four benchmarks were conducted through the use of a custom program in order to demonstrate the effectiveness of instruction level parallelism. The four benchmarks are grouped into two pairs. One pair to evaluate multiplication and another pair to evaluate division.

Multiplication and division were chosen because they are relatively long duration instructions on the evaluated processors. Using instructions with a long duration minimises the effects of the other necessary instructions.

Each pair of operator tests consists of two benchmarks, one benchmark where instructions can execute in parallel and another where they cannot. These experiments have little meaning other than to illustrate implicit instruction level parallelism.

Details on how the benchmarks were performed as well as the source code for the benchmarks is presented in Appendix A. Table 2.5 shows the results of the four benchmarks on three different processors (three different processors that were available for testing).

Processor type	Operation	Non-parallel rate	Parallel rate	Ratio
UltraSparc-II at 296Mhz	Multiplication	43.84	66.67	1:1.52
	Division	5.263	5.263	1:1.00
Pentium-4 at 2.8Ghz	Multiplication	222.2	666.7	1:3.00
	Division	22.66	78.43	1:3.46
Celeron at 2.4Ghz	Multiplication	157.4	436.4	1:2.77
	Division	28.99	86.96	1:3.00

Table 2.5: *Instruction processing rates where parallelism is possible and where parallelism is impossible. All rates are in millions of operations per second.*

Looking at Table 2.5 several aspects regarding the numbers of execution units in the various systems becomes visible. Note that the parallel and non-parallel rates for the division instruction on the UltraSparc-II processor are identical. This would indicate that it only has one execution unit supporting divide instructions. This is to be expected considering that it is several years older than the other processors. On average the parallel rates are substantially greater than the non-parallel rates illustrating the effects of implicit parallelism.

2.7 Explicitly parallel instruction computing

UNIVERSITY
JOHANNESBURG

A great deal of the work performed by out-of-order processors could be performed at compile time. In Section 2.6.2 the point was made that in ideal circumstances there would be no need for out-of-order execution at all since the instruction scheduling could be done at compile time. The simplification could be taken a step further by encoding parallelism information into the instructions as opposed to analysing the instructions during execution. Processors that take this approach are referred to as *EPIC* processors where EPIC stands for *explicitly parallel instruction computing*.

The principle behind EPIC is that it moves complexity off the processor and into the compiler. EPIC processors are characterised by having highly predictable behaviour and simple instruction sets. That is to say aspects that are important for building accurate schedules, such as instruction durations, are highly predictable on these processors. An added advantage of scheduling the instructions at compile time is that the performance of the resulting program is very consistent.

There are several variations in terms of EPIC processors and their instruction sets. An EPIC instruction set will usually have some way of describing the order in which the

instructions must be executed as well as the execution unit assignments. EPIC processors/instruction sets fall into three different groups based on the approach taken.

2.7.1 Vector processors

Vector processors implement a SIMD processing scheme meaning that parallelism is usually limited to a single instruction processing separate pieces of data. A single vector instruction will perform multiple identical operations on different data. For example a single vector instruction might implement two multiplications. Since each multiplication requires separate data, four registers would be required for such a multiplication. In practice this would cause trouble since some vector instructions perform many parallel operations, as many as 16 separate operations in some circumstances. To reduce the number of registers referenced the data is loaded into special vector registers that can contain multiple independent values. For example a single 128 bit vector register may contain four separate 32 bit floating point values.

Vector processors have the most restrictive parallelism model of the three EPIC approaches discussed. Vector processors have a long history in the supercomputing industry because of the massive computational power of these systems when solving certain types of problems. A significant disadvantage of vector processors is that they are only useful for solving certain types of problems, where data can be grouped in vectors, such as physical simulations or linear systems.

Compilers that target vector processors are fairly effective provided the programs being compiled are suitable for processing on vector computers. The process used by compilers to *vectorise* programs is described by Zima and Chapman in [33].

In recent times several manufacturers have added vector instructions to computers intended for the desktop market, however it has proven difficult for compilers to exploit these instructions. The vector instructions have been added in order to boost the multimedia capabilities of these computers. However, unlike the supercomputer vector-processors, these desktop processors do not have *strided* load instructions.

A normal load instruction reads a single value from memory and places it in a specified register. A strided load reads multiple separate pieces of data from memory in a single operation and usually places them into a single vector register. It is called a strided load because the load instruction skips a specified amount of space between each value in

memory. Compiler techniques used for pure vector processors rely heavily on being able to specify a stride for a load instruction. Since desktop processors do not support strides, compilers have difficulty in using the vector instructions efficiently, if at all.

1	MOV	$0(r1)$,	$r2$	
2	MOV	$4(r2)$,	$r3$	
3	MUL	$v1$,	$r2$,	$v2$
4	XOR	$r2$,	$r3$,	$r2$

Table 2.6: *An example program similar to the one given in Table 2.3. v_1 and v_2 are both vector registers. Instruction 3 is a vector multiply instruction that multiplies both fields within the vector register v_1 with a single value r_2 resulting in v_2.*

Table 2.6 shows a program containing vector instructions. Figure 2.10 illustrates the execution unit loading and parallelism of the vector instruction.

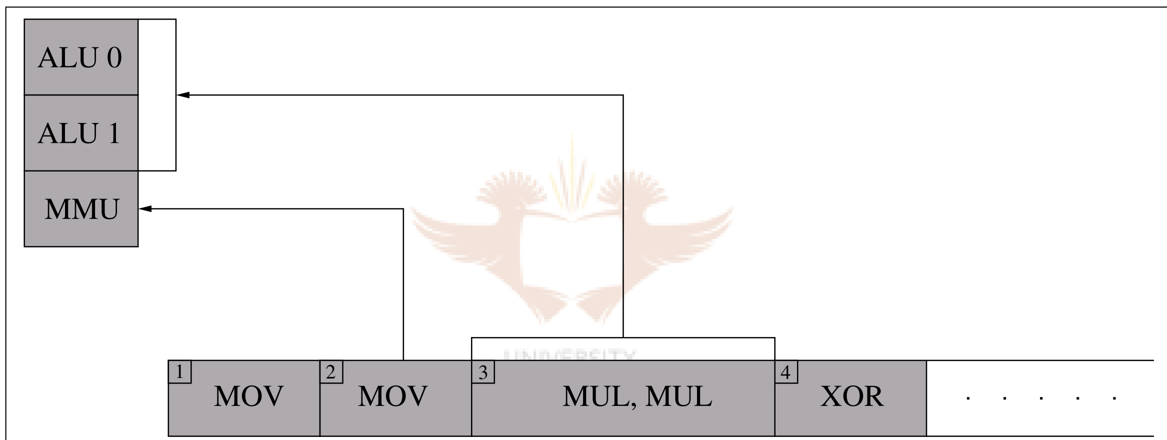


Figure 2.10: *This figure shows the execution unit loading for the instructions including the vector instruction which is instruction 3. Note that the single instruction loads two execution units.*

An example of a pure vector processor is the NEC SX-6 which powers the Earth Simulator supercomputer (used for earth sciences simulations). All of the SX processors from NEC are vector processors and have a long history of use in scientific applications and have demonstrated the effectiveness of vector processors.

2.7.2 Very large instruction word processors

Very large instruction word (VLIW) processors have the simplest instruction sets of all EPIC processors. In traditional instruction sets each instruction is decoded separately however in a VLIW processor several instructions are bundled together. This bundling

approach is not unique to VLIW processors, however in VLIW processors the bundle has special properties. Each instruction bundle has an instruction for each execution unit, therefore the number of instructions in the bundle is equal to the number of execution units. A formal definition for the VLIW technique is provided by Cosnard and Trystram [9]. VLIW processors implement a MIMD approach to parallelism.

	ALU 0	**ALU 1**	**MMU**
1	NOP	NOP	MOV $0(r1)$,$r2$
2	XOR $r2$,$r4$,$r7$	MUL $r2$,$r4$,$r5$	MOV $4(r2)$,$r3$
3	ADD $r3$,$r5$,$r6$	NOP	NOP

Table 2.7: *This is the same program as the one given in Table 2.3 except that it is executing on a VLIW processor. Each column contains the instructions for the relevant execution unit. Each row is a bundle of instructions. Notice that the order of the instructions has changed by executing the XOR instruction before the ADD instruction.*

Table 2.7 demonstrates what the program given in Table 2.3 would look like if the target processor is a VLIW processor instead of using implicit parallelism. NOP instructions are used to fill up execution slots where no instruction can be executed.

In the example given in Table 2.7 an assumption is made regarding the duration of the instructions; all of the instructions in a bundle are of identical duration. In practice differing instruction durations is implemented by inserting NOP instructions where an execution unit is still busy with a previous instruction.

UNIVERSITY OF JOHANNESBURG

Table 2.8 demonstrates a program for a VLIW processor where the multiplication instruction requires two clock cycles to complete execution. Supporting instructions with very long durations is clearly difficult in VLIW instruction sets as they can greatly increase the number of bundles required. Additionally programs can be quite large when compiled for VLIW processors because of the large number of NOP instructions that may be required.

	ALU 0	**ALU 1**	**MMU**
1	NOP	NOP	MOV $0(r1)$,$r2$
2	XOR $r2$,$r4$,$r7$	MUL $r2$,$r4$,$r5$	MOV $4(r2)$,$r3$
3	ADD $r3$,$r5$,$r6$	NOP* (Still busy with MUL)	NOP

Table 2.8: *This program is identical to the one in Table 2.7 except that the processor requires 2 cycles to perform a multiplication.*

The most commonly available VLIW processor system is Sony's Playstation 2, at the heart of which is a complex logic package that includes two very similar VLIW processors (according to Sony [2]). Most instructions in this processor require a single clock cycle to execute limiting the number of NOP instructions necessary. Since multimedia

applications are the only type of application the system is used for, the use of VLIW is actually very effective. Information regarding effectiveness of instruction level parallelism techniques when applied to specialised workloads can be in found in Sankaralingam et al. [27].

Intel IA-64 architecture processors (I.E. Itanium) use a variation of VLIW design. The design is more complicated than a simple VLIW design because it introduces restrictions on the types of instructions that can be combined in a single bundle. Phillips and Texas Instruments have both released embedded media processors that employs EPIC technology. Both the Itanium processor and the Phillips processor are discussed by Patterson and Hennessy [25]. Undoubtedly other variations of EPIC exist, for example embedding a unique identifier for the execution unit into the instruction. Regardless of the approach taken, flexible EPIC schemes have similar requirements in terms of compiler technology and programmer methodology.

2.7.3 Branch predication

In the section on implicit parallelism the concept of branch prediction was introduced as a method for increasing the parallelism of a program. Branch prediction is only effective on processors that employ an implicit parallelism approach. However a similar concept can be applied to EPIC processors and, in the spirit of EPIC, a part of the work involved can be offloaded to the compiler. This approach is called branch predication (not prediction). Patterson and Hennessy discuss branch predication as well as branch prediction [25].

The branch prediction concept introduced the idea that the changes a group of instructions make to the state of the system could be revoked. The group of instructions are the instructions that were predicted to execute after a branch instruction. Branch predication is where the compiler creates the instruction groups at compile time and the processor just manages the switching between them. A very important advantage of this scheme is that instructions known to execute and instructions predicted to execute can be interleaved in a schedule.

Table 2.9 illustrates the interleaved nature of instructions belonging to two different predicates. Even though the instructions execute at the same time some of the instructions can be discarded. Typically a predicate branch instruction would follow that would reference the predicate codes attached to the instructions. A normal branch would also have to be supported to allow loops to be implemented. The combination of the predicate branch

	ALU 0	**ALU 1**	**MMU**
1	NOP	NOP	MOV[0] $0(r1)$,$r2$
2	XOR[0] $r2$,$r4$,$r7$	MUL[1] $r2$,$r4$,$r5$	MOV[0] $4(r2)$,$r3$
3	ADD[0] $r3$,$r5$,$r6$	NOP* (Still busy with MUL)	NOP

Table 2.9: *This program includes a predicate identifier in square brackets for each instruction except for NOP since it does not execute. The multiplication in bundle 2 is part of a different predicate to the rest of the instructions.*

instructions and the normal branch instructions is a fairly good replacement for branch prediction.

Branch predication features allow compilers to extend parallelism beyond branches with the added advantage that the compiler can also employ its understanding of the program in doing so. Variations and implementation details of the branch predication principle are introduced by August et al. [3]. A study on the effectiveness of multiple variations of branch predication was published by Tyson [29].

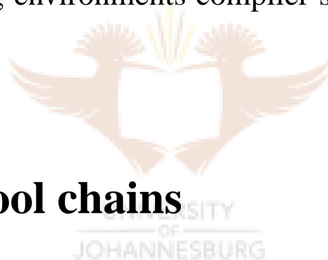
Instruction level parallelism is an important method for increasing processor performance. EPIC processors offer the possibility of performance increases beyond what is possible by just increasing the frequency of the processor. The explicit parallelism approach has been proven by the success of vector processors and less limited explicit parallelism models have even more possibilities which are yet to be fully explored.

UNIVERSITY
OF
JOHANNESBURG

Chapter 3

Compilation

Compilation is the process of converting program source code into individual instructions that a computer processor can understand. Due to the complexity of compilers and their interaction with operating environments compiler software is divided into separate programs.



3.1 Compilers in tool chains

A tool chain consists of one or more programs that are required or aid in the development of a program. The specifics regarding tools in the tool chain differ from system to system. On some systems the tools may be combined into a monolithic program but this limits reuseability and flexibility so this approach is generally seen in a bad light.

The following list describes a number of tools that are standardised on Unix systems (governed by the IEEE POSIX P1003.2 standard). For each tool a short description is given as well as the executable name.

- Compiler, converts a program described by high level source code to assembly code (in Unix the command is **cc** for the default C language compiler).
- Assembler, converts a textual assembly language program into an object file which contains the binary representation of the program (in Unix the command is **as**).

- Linker, combines multiple object files and libraries to produce a final executable file (in Unix the command is **ld**).

An added advantage of this task isolation is that it prevents any single tool from becoming too large and complex. Since this dissertation is concerned with compilation, only the compiler will be discussed. Information regarding assemblers and, to some extent, linkers is presented by Aho et al. [1].

3.2 Structure of a compiler

Compilation consists of multiple phases where each phase converts the current form of the program representation to another form or, alternately, adds additional information to the current form as a result of analysis. Although each phase is discussed separately a compiler implementation could perform the tasks associated with each phase at the same time as opposed to performing them in separate passes. The wisdom in combining the phases is however a bit more questionable. Large scale integration would limit the compiler's ability to optimise the program as well as limiting the flexibility and portability of the compiler.

Each of the phases required for compilation is listed below:

- lexical analysis
- syntax analysis
- semantic analysis
- intermediate code generation
- code optimisation
- code generation

The structure of a compiler and the necessary phases are discussed in detail in Aho et al. [1]. Each of these phases are discussed in the remainder of this chapter.

3.3 Lexical analysis

Lexical analysis is the process of converting the text of the program source code into a stream of *tokens*. Lexical analysis is also responsible for storing string literals, numbers and other syntactical information in the symbol table for use in later phases. This entire process is often referred to as *scanning*. Individual source code elements are identified by applying pattern matching rules to an input stream of characters. The most popular way of representing these patterns is in the form of regular expressions.

In general a token can be defined as an identifier for a particular type of program element. For example a 1 might represent the string 'for' and a 2 an open brace, '{'. String literals, numbers, etc. are also represented by tokens however they have additional attributes that provide information regarding the particular instance. In the case of a string literal this additional information could be an index into a table containing the actual contents of the string.

Source code elements that have no meaning to the compiler, for example comments, are removed in this phase. Miscellaneous other tasks are sometimes performed in this phase, such as keeping track of line numbers and macro expansion.

A set of lexical rules have been provided as an example in Table 3.1. Many books [12] have in depth discussions on the topic of regular expressions as they are used in a wide range of applications.

Lexical element	Regular expression	Match example
Assign	"="	=
Identifier	*[a-z][a-z0-9]**	variable1
Add	"+"	+
Multiply	"*"	*
Constant	*[0-9]+*	100

Table 3.1: *An example of several regular expressions and the strings they will match.*

In Table 3.1 two lexical elements are of particular importance. The "Identifier" element states that the first character must be a letter from a to z. This restriction is expressed through the term "[a-z]". The second term "[a-z0-9]*" states that there must be zero or more following characters with the restriction that they are letters from a to z or numbers from 0 to 9. The "Constant" element is similar to the "Identifier" element in that it matches one or more characters that range from 0 to 9.

Clearly, it is far easier to analyse a program in the form of tokens than in the original text. These tokens form a list or stream that is fed into the syntax analyser.

3.4 Syntax analysis

The *syntax analysis* phase is concerned with converting the incoming stream of tokens into a tree that describes the structure of the program that the tokens were created from. This process is commonly referred to as *parsing*. An additional function of syntax analysis is to determine if the program being parsed is structurally correct.

The structural rules for the language being compiled are referred to as a *grammar* and in the case of most languages can be applied in a context free manner. A grammar consists of a set of rules where each individual rule contains one or more clauses. Each clause consists of tokens and references to other rules. In the case of a real-world parser, clauses also have actions associated with them. These actions perform some function when the tokens in the clause associated with the action is matched with the exact sequence of tokens. Typically the action is to introduce a node to the tree-based representation of the program that is being created.

UNIVERSITY
JOHANNESBURG

A grammar can be represented as a finite state machine but this is not commonly done as it provides little help in the job of parsing or understanding the grammar. A more useful representation is that of a directed cyclic graph. However, it is difficult to work with a grammar in this format, so grammars are usually defined as a set of interrelated rules.

A simple example of a grammar is given below.

A, B and C are tokens (otherwise known as terminals) for the grammar below. The **D** and **E** symbols are used to represent rules (otherwise known as non-terminals). The vertical pipe symbol '|' is used as an OR operator. We start at **E** because it is the root of our grammar. Remember that a grammar is a directed cyclic graph, hence the use of graph terminology.

Non-terminals have been written in bold font where as terminals are not.

- **D** → B B C | C B A
- **E** → A **D** B

The above grammar can parse both of the streams given below

1. A B B C B
2. A C B A B

Assume that we are trying to parse the first stream, we start with the A token which we match with the A in the E rule. The next token in the stream is B, which must match a token from either of the two clauses of rule D. In the current example it matches with the B in the first clause and therefore continues matching the following B and C in the clause / stream at which point it continues matching tokens from the E rule. The process is shown in the table below.

Stream token	**Rule**	**Clause**	**Clause token**	**Match status**
A	E	[A]**D**B	A	TRUE
B	D	CBA	C	FALSE
B	D	[B]BC	B	TRUE
B	D	B[B]C	B	TRUE
C	D	BB[C]	C	TRUE
B	E	A**D**[B]	B	TRUE

Table 3.2: *Grammar matching example for example stream 1. If an input token matches a token in the appropriate clause then the token from the clause is enclosed in brackets in the 'Clause' column.*

UNIVERSITY OF JOHANNESBURG

The above technique works well so long as the decision about which clause to use for a particular rule can be made with only the first terminal token of that clause. Grammars that conform to this requirement as well as being parsed in a left to right fashion are referred to as *LL(1)* grammars. Typical *LL(1)* grammars are more complex than the one given in this example and may therefore need to be processed before use [1]. After processing, an *LL(1)* grammar will always be parsable using the technique described above. Grammars exist that do not conform to this requirement and are called *LL(k)* or *LR(k)* grammars, where k refers to the number of tokens look ahead that are required to make the clause decision. Fortunately most commonly used languages have an *LL(1)* grammar. This includes the C programming language, which is the programming language that the compiler implementation supports.

Given a simple expression

$$a = a + b * 3$$

the syntax graph generated by the syntax analysis phase would look something like Figure 3.1.

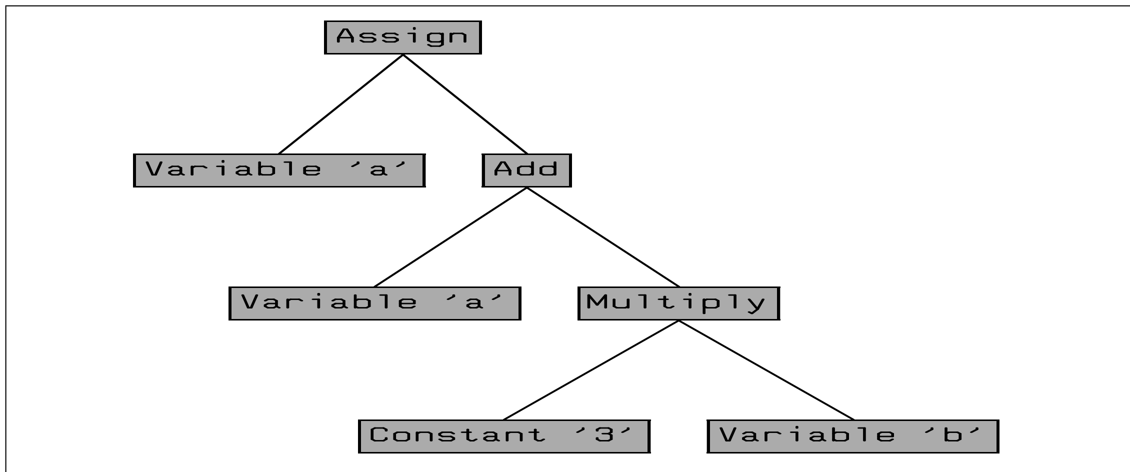


Figure 3.1: *Syntax tree example: Each node refers to a particular syntactical element. A keyword that identifies the particular type of token / syntactical element has been placed at the centre of each node. The arcs that join the nodes indicate the flow of information, for example the result of the multiplication forms one of the two operands for an addition.*

The purpose of each type of node in Figure 3.1 is given below

Node type	Description
Assign	Set the variable referenced by the left child to the value of the right child
Variable	A reference to a variable. The string in quotes is the name of the variable.
Add	Adds the left child's value with the right child's value and returns the result up the tree to the parent.
Multiply	Multiplies the left child's value and the right child's value returning the result up the tree to the parent.
Constant	Represents a constant numeric value that was specified in the source code.

Table 3.3: *Descriptions for the token / syntactical elements found in Table 3.1.*

3.5 Semantic analysis

Semantic analysis is concerned with the propagation and analysis of data types in the program utilising the tree structure that was generated by the syntax analysis phase. The two most important aspects of this phase are determining the data types of nodes in the tree structure as well as type checking the program according to the rules of the language regarding the use and compatibility of different data types.

Propagating data type information in the tree representation of the program is important

so that the correct behaviour can be applied when generating the actual computer instructions. For example, different instructions are used when performing the same operation on integer and floating point data.

The other important aspect of the semantic analysis phase is type checking. This aspect is concerned with locating and reporting the use of incompatible data types.

Conversion between values of differing data types requires special attention. An example is assigning the contents of a floating point variable to an integer variable. It cannot be done by simply copying the bits (for example using a MOV instruction in an x86 processor) from one memory location to another. In such circumstances an additional data type conversion node is necessary.

Values of some data types cannot be converted to other data types and therefore the compiler must inform the user by way of an error message. For example a pointer to integers is incompatible with a pointer to floats. The language rules used may also prohibit other uses of specific data types (as opposed to conversions). For example only integer data types can be used as array indices in C/C++.

Some of the nodes in the syntax graph have data types associated with them due to their nature. For example a string literal can only be of a string type. The core process in the semantic phase of compilation is the propagation of data types through the syntax graph. What is meant by this statement is that the data type information that is already present must be propagated through the graph as required. A simple example of this would be an addition operator. The data type of the instruction is determined by the data types of its operands.

Two approaches exist for propagating the data type information. The top down approach refers to the relationship whereby a child node inherits the data type of its parent. This has the effect of the data type propagating down the tree. The other option is the bottom up approach where a node's data type is determined by the node's children. Data types move up the tree when this approach is used.

The data type propagation methods are applied to sub trees from the main tree depending on the semantics of the language. For example, in the C/C++ language, each statement would be a sub tree and so in each the statement the data types to be determined are independent of other statements. The exact rules governing data type propagation is completely language dependent.

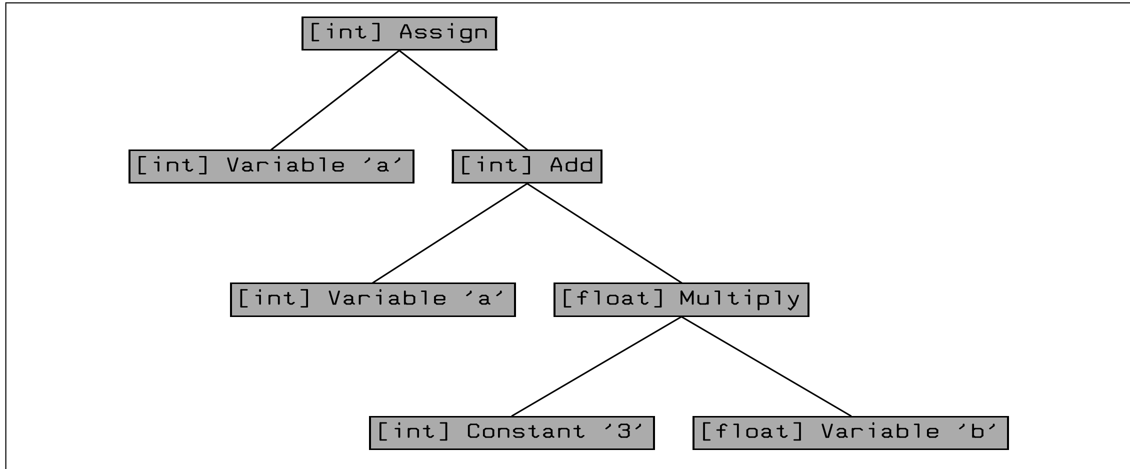


Figure 3.2: *The syntax tree example with semantic information added in brackets. It is assumed that variable 'a' was declared as an integer, variable 'b' as a float and the numeric constant '3' is an integer. Please note that this example does not strictly comply with the rules on data type propagation for the C/C++ languages.*

The example shown in Figure 3.1 has been passed through the semantic phase and the result is shown in Figure 3.2. The data type for each node can be found in the square brackets preceeding the node's label. This syntax tree with semantic information is then fed into the next stage of compilation. A program in this form could be executed by an interpreter.

UNIVERSITY OF JOHANNESBURG

3.6 Intermediate code generation

Intermediate code is a machine-independent format for representing compiled programs. Intermediate code can, in-fact, be seen as assembly code for a virtual computer. The intermediate code will be converted to a real architecture at a later point in the compilation process.

It should be noted that not all compilers utilise an intermediate code generation phase, it does however offer certain advantages over direct conversion from a syntax tree to machine specific assembly code. These are namely that the compiler can easily and efficiently be re-targeted for different architectures or models within an architecture family. Certain aspects of code analysis, debugging and optimisation are also simplified due to the intermediate code representation and therefore not having to cope with real world machine idiosyncrasies.

Although there are no specific rules regarding the form of intermediate code implementation, the commonly used forms generally fall into two categories that are determined by the number of addresses used in each instruction. An instruction is a single non-divisible unit of work, an example of this would be an addition operation as it is implemented as a single instruction in most processors.

Node type	Action
Assign	N.child(0).execute(); N.child(1).execute(); N.address = generate(ASSIGN, N.child(0).address, N.child(1).address);
Add	N.child(0).execute(); N.child(1).execute(); N.address = generate(ADD, N.child(0).address, N.child(1).address);
Multiply	N.child(0).execute(); N.child(1).execute(); N.address = generate(MULTIPLY, N.child(0).address, N.child(1).address);
Variable	N.address = identities_table_lookup(N);
Constant	N.address = constants_table_lookup(N);

UNIVERSITY JOHANNESBURG

Table 3.4: *Actions that can be used to convert a syntax tree like that of Figure 3.2 to intermediate code. N refers to the current node, N.child(*) refers to the child(*) of the current node. 'generate' creates an intermediate code instruction. In the example N.child(0) and N.child(1) would be two separate child nodes.*

In order to convert a syntax tree to intermediate code, the syntax tree is traversed using a custom traversal. This is implemented by having a set of actions for each type of node in the syntax tree. The traversal begins at the top of the tree by executing the actions for that type of node. The actions for a node will invoke the actions of the child nodes as well as generate the appropriate instruction(s). When the actions end for the node at the root of the tree the traversal is complete. Table 3.4 contains possible actions for the different types of nodes used in the syntax tree in Figure 3.2.

identifier	instruction	operand 1	operand 2	result
0	Multiply	variable[1]	constant[0]	t_1
1	Add	variable[0]	t_1	t_2
2	Assign	variable[0]	t_2	t_3

Table 3.5: *Quadruple representation of the syntax tree in Figure 3.2. Please note that 'variable[*]' and 'constant[*]' refer to the symbol table and constant table entries respectively.*

identifier	instruction	operand 1	operand 2
0	Multiply	variable[1]	constant[0]
1	Add	variable[0]	*result of instruction 0*
2	Assign	variable[0]	*result of instruction 1*

Table 3.6: *Triple representation of the syntax tree in Figure 3.2. Please note that 'variable(*)' and 'constant(*)' refer to the symbol table and constant table entries respectively.*

The intermediate code operations that were generated now exist in a table in an order that will correctly execute. The result of the intermediate code generation phase is shown in Table 3.5. This representation is one of the two representations mentioned earlier and is referred to as the quadruple representation (or three address code). Quadruple refers to the fact that four columns are required for the table (the first column is just an index sequence), that is to say a single entry has three addresses, 'operand 1', 'operand 2' and 'result'.

The latter of the two representations for intermediate code is commonly referred to as the triple representation (or two address code) and as the name suggests it has one less column than the quadruple format. The on-going example in this format is given in Table 3.6. The triple representation is a simple adaptation of the quadruple representation that reduces resource usage and more importantly simplifies access and analysis.

As stated earlier it is possible to skip intermediate code generation and generate architecture specific code immediately. The techniques required in order to do so are the same as the ones introduced in this section which makes sense when you consider that intermediate code can be seen as the assembly code for a virtual yet realistic architecture.

3.6.1 Generating data dependency graphs

An alternative to the triple and quadruple representations is to create a data dependency graph (DDG). A data dependency graph is a type of directed acyclic graph where the nodes are instructions and the arcs represent dependencies. Unlike the other representations a DDG can represent both data dependencies as well as dependencies that do not involve data. Unlike the tree structures seen up to now, an instruction in a DDG can have more than one parent and more than one child.

The DDG representation, although less commonly used, is more suitable for optimisation than a list based representation like triples or quadruples. An additional advantage of

creating a DDG is that it trivialises the removal of common sub-expressions which is an important optimisation step.

As with the list based representations a table containing a program segment for each type of node in the syntax tree is created. The program segment generates one or more instructions that implement the function associated with the node. Since the DDG is a representation for intermediate code the instructions generated are intermediate instructions as opposed to processor specific instructions.

This is where the similarity to a list based representation ends. Before a new instruction can be added we must first check that there is not an identical instruction that has already been generated. We first need to determine what exactly is meant by identical. Two instructions are identical if they are both of the same type and have identical descendents in the syntax graph representation of the program. If two instructions are identical then the result of the earlier instruction can be reused.

Clearly there are many exceptions where this reuse technique cannot be used. For example if two identical call instructions exist in a program segment then only one would be generated. This however would be incorrect as only one call would be executed instead of twice as was expressed by the program. In order to avoid this sort of problem it is necessary to provide a customised implementation for each type of node in the table described earlier.

UNIVERSITY
OF
JOHANNESBURG

Another type of problem exists where two instructions load data from the same variable and therefore are deemed to be identical and reuse occurs. On its own this is not a problem and is in fact required in order to remove common sub-expressions. However if an instruction performs a store on the variable and occurs after the first load but before the second then the second load cannot reuse the result of the first load. To do so would mean that the second load would return old and possibly invalid data.

In order to avoid this sort of problem we need to maintain a list of reusable instructions separate from the DDG undergoing construction. We could then remove instructions from the list should they become invalid. This solves the problem described with the multiple variable access because an existing load instruction can be remove from the list when a store instruction is processed.

The use of a separate list has an added advantage. Only the list has to be searched when looking for an existing identical instruction instead of traversing the entire DDG. A hash table is the ideal data structure for the reusable instruction list since it allows for fast

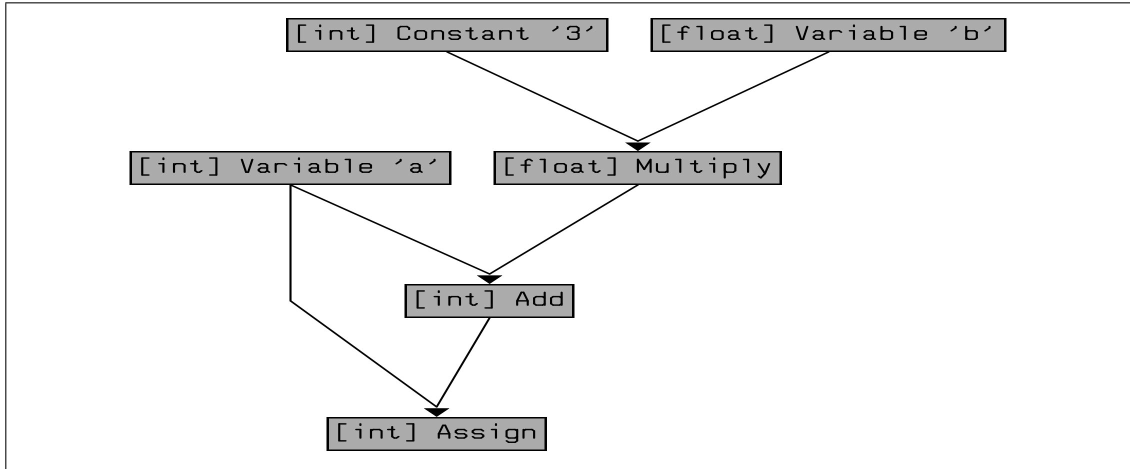


Figure 3.3: *A DDG created from the syntax tree with semantic information shown in Figure 3.2. Each arc indicates data dependence where the arrow points from the data producer.*

searching, insertion and deletion. Aho et al. [1] discuss the creation of data dependency graphs as well as suitable data structures.

The semantic graph form of the program given in Figure 3.2 was converted to a data dependency graph. The result is shown in Figure 3.3. In practice the generated DDG would possibly be closer to a real instruction set even though it is an intermediate code representation.

UNIVERSITY OF JOHANNESBURG

3.7 Code optimisation

Two types of program optimisation exist. Programs can either be optimised for size or they can be optimised for speed. This dissertation is only concerned with optimising for speed so none of the material will take size into account.

Code optimisation is intertwined with code generation because different optimisation techniques are applied on both the intermediate code as well as on the final assembly code. Hence optimisation may occur both before and after code generation takes place.

The topic of code optimisation is vast as there are an innumerable number of different optimisations that can be performed. A few different categories of optimisations are introduced in this section however it is by no means a thorough discussion of what can be done.

1. *Common subexpression removal* is concerned with identifying duplicate expressions and the removal of duplicates. This type of optimisation was discussed in the section detailing the generation of data dependency graphs.

2. *Copy propagation* is where unnecessary copying of values between variables is eliminated. This is achieved by placing resulting values directly in destination variables instead of first placing them in temporary variables or registers.

3. *Code motion* is the process of moving instructions, when possible, out of a loop body and therefore reducing the number of times the instructions must be executed.

4. *Induction variable removal* can be employed in loops where multiple variables are present and are incremented within the loop. In such circumstances some of the variables can be eliminated.

5. *Reduction in strength* is where one or more instructions are replaced by other instructions that perform the same task yet are computationally cheaper or offer a higher level of parallelism.

6. *Instruction scheduling* has become important with the wide spread use of processors that execute instructions in parallel. With the introduction of EPIC processors this type of optimisation has become even more significant. Many older books on compilers do not have any significant information on this type of optimisation as it a relatively new field. This dissertation focuses primarily on instruction scheduling.

Before any optimisation can occur the instructions must be broken up into *basic blocks*.

> A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [1].

All of the instructions within a basic block are guaranteed to execute if any of the instructions in the block execute. The branch decision that occurs at the end of a basic block cannot be determined at compile time and therefore assumptions cannot be made concerning further instructions.

3.8 Code generation

Code generation is concerned with converting intermediate code into assembly code. The resulting assembly code can be fed into the assembler and from there into the linker. Assembly code consists of hardware instructions in mnemonic form with additional information regarding data types, functions, etc.

Each intermediate code instruction has to be substituted with one or more instructions from the target processors instruction set. For each instruction a textual string is generated. All the strings generated together form an assembly language program. The assembly code is generally expressed using the AT&T syntax as opposed to Intel syntax as AT&T syntax is very precise and therefore is better suited to back-end assembly.

3.9 Automating the compiler creation process

Tools exist to automate parts of the compiler creation process. Typically both the lexical and syntax analysers are created by automated tools from description files provided by the programmer. The use of compiler creation tools simplifies development as well as allowing for the use of more maintainable definitions for the lexical and syntax analysers. An added advantage of automated compiler generation tools is that the code that the generators produce is already highly optimised.

The *Lex* program creates a lexical analyser from a given definition file. The definition file describes the lexical elements to be matched in terms of regular expressions and provides the programmer with the ability to provide some custom code for each expression. The Lex program is standardised (IEEE POSIX P1003.2) with many implementations in existence. The *Flex* implementation of the Lex standard was used in the compiler implementation.

Similarly the *YACC* (Yet Another Compiler Compiler) program (also standardised in IEEE POSIX P1003.2) creates a syntax analyser from a given definition file. The parser generated is designed to work with the lexical analyser generated by Lex. The definition file describes the grammar in terms of rules similar to the ones shown in the section on syntax analysers and as with the Lex program, custom code can be added for each rule or clause. The YACC program is also standardised (in the POSIX standard) with a implementation from the Free Software Foundation under the name *Bison*. This implementation

of the YACC standard was used in the compiler implementation.

The book "Introduction to Compiler Construction with Unix" by Shreiner et al [28] discusses the use of compiler creation tools in detail.

Chapter 4

Instruction scheduling for code optimisation

In Chapter 3 we saw that a program is divided up into basic blocks and that these basic blocks are optimised individually. Given that we are trying to maximise the performance of the program we must therefore minimise the duration of each basic block. In order to achieve this goal we will have to determine the ideal order or schedule of the instructions in the basic block subject to a number of constraints.

UNIVERSITY
OF
JOHANNESBURG

In general scheduling problems are concerned with finding the optimal schedule for n *jobs* which must be processed by m *machines*. A job refers to a single indivisible task that must be performed on a machine. In the case of a computer program a job would be an instruction. The term machine describes a reusable entity that can do the work described by the individual jobs. Execution units perform this task in a computer processor.

The terms *job* and *machine* originate from the manufacturing industry where goods are manufactured on a construction line. Scheduling the jobs on the construction line can have a significant impact on the cost per item therefore the manufacturing industry has been the primary driving force in scheduling research. Additional background material regarding the scheduling problem can be found in French [16].

Different variations of the scheduling problem have different names. Two types of scheduling problems are of interest to us. The *job shop* problem and the *parallel machine* problem. The problem we need to solve is actually a variation of the parallel machine problem but since there is very little suitable material on this type of problem methods developed

for the job shop problem will be used instead. Various types of scheduling problems are described in Baker [4].

The parallel machine problem is concerned with scheduling n jobs where each of the n must only run through one of the m identical machines. Constraints are present to restrict the order in which jobs are processed. This model allows for parallelism on a per job basis. Please note that this type of problem does not exactly describe the problem we need to solve since it assumes all of the m machines are identical. In a processor different types of execution units are present although multiple units may be present for each type. Initially we will assume that all executions units are identical, however, the following chapter will deal with different types of execution units.

The job shop problem is concerned with scheduling n jobs where each job must pass through m machines. Additionally jobs are subject to constraints regarding the order in which they pass through the m machines.

Regardless of the differences, most of the techniques developed for the job shop problem can be adapted to our variation of the parallel machine problem that better suits the problem at hand.

Different aspects of a scheduling problem can be optimised, however we are only interested in optimising the *makespan*. The makespan of a schedule is the amount of time required to complete all of the jobs. Since we are scheduling the instructions in a computer program the makespan refers to the amount of time required to execute all of the instructions in a basic block.

4.1 Formulating the scheduling problem as a mathematical program

In Chapter 3 we saw that the final representation for a basic block was in the form of a data dependency graph. An example of a data dependency graph is given in Figure 4.1 where each node represents an instruction and each arc a data dependency.

In order to represent a schedule each instruction must have a variable associated with it that represents the time at which the instruction executes. For this purpose we could use either the time at which the instruction starts or the time at which it ends. We will

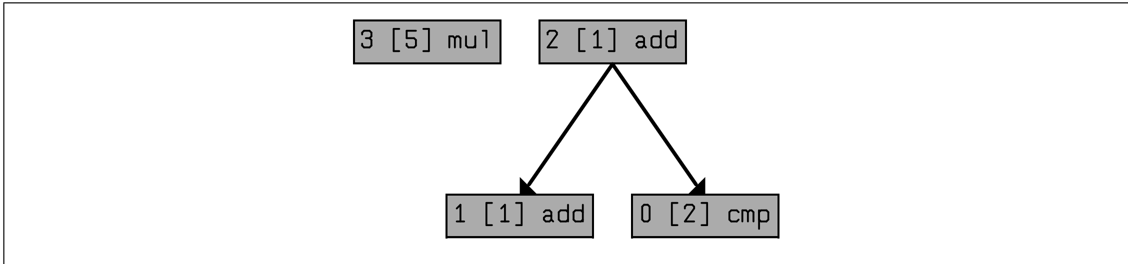


Figure 4.1: *An example of a data dependency graph. The first number in each node uniquely identifies the instruction that the node represents. The second number, in square brackets, is the duration of the instruction.*

use the ending times of the instructions because a dependency restricts an instruction to execute *after* one or more instructions that it is dependent on. In addition to this we will be minimising the time at which the basic block ends.

Let:

n	be the number of instructions in the basic block.
$x_i, i = 0, \ldots, n-1$	be the ending times of the instructions measured in clock cycles. Since x_i represents a point in time it has a non-negativity restriction.
$t_i, i = 0, \ldots, n-1$	be the instruction durations of the n instructions. With the exception of instruction durations associated with dummy instructions (which will be covered shortly) all durations must be positive integers.
$N = \{\ldots, (j, i)\}$	be the set of instruction dependencies where i is the index of the dependent and j is the index of the preceding instruction.

For each dependency in the data dependency graph a constraint must be added as demonstrated by Baker [4]. The general form for the constraints is given below.

$$x_i - t_i \geq x_j \quad \forall (j, i) \in N \tag{4.1}$$

This constraint ensures that the time at which instruction i starts executing (calculated by $x_i - t_i$) can only be after instruction j has ceased executing.

Instructions that are not dependent on any other instructions are a special case that we can refer to as *starting instructions*. Because all instructions are scheduled according to the time they cease execution, constraints are required to ensure that the starting instructions

do not have ending times smaller than their durations. Without these constraints starting instructions would have ending times of zero. Therefore a constraint must be added for each starting instruction in the form given below.

$$x_i \geq t_i \tag{4.2}$$

These constraints are very similar to the dependency constraints because they serve a similar purpose. We can convert our starting instruction constraints to dependency constraints if we say that each starting instruction is dependent on a dummy instruction (shown as *sdummy* in figures). This new dummy instruction has a duration of zero as it is not a real instruction. Its ending time will always be equal to zero since it is not dependent on any other instruction. Figure 4.2 repeats the data dependency graph from Figure 4.1 but with the addition of a dummy instruction that the starting instructions are dependent on. Let us refer to this dummy instruction as the starting dummy instruction.

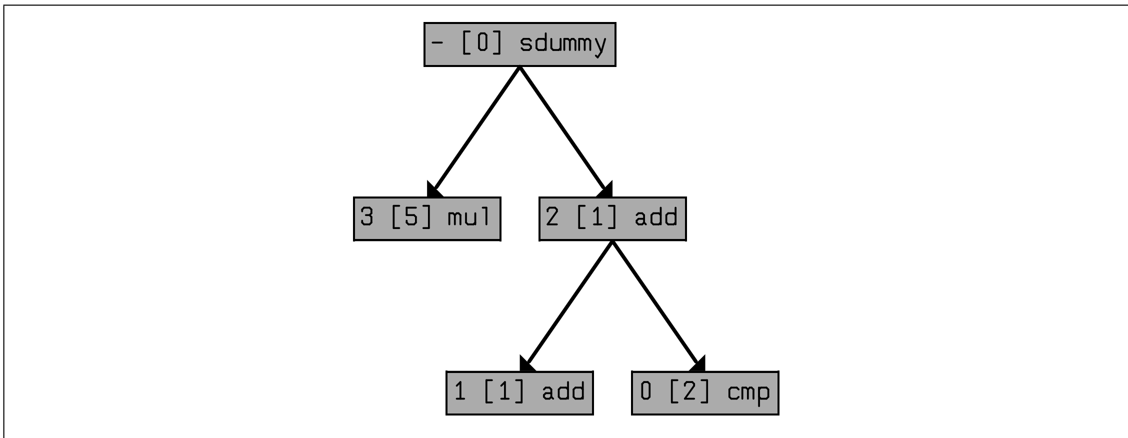


Figure 4.2: *Data dependency graph with a dummy instruction that the starting instructions are dependent on.*

4.2 Minimising the makespan

The total time taken for the entire basic block is determined by the time at which the execution of the last instruction ends. In other words the maximum ending time of all of the instructions is the makespan of the basic block.

In order to formulate the problem as a linear programming problem the objective function can only contain linear terms. The use of the maximum function in the objective function

would mean that the objective function is no longer linear. Therefore the problem could not be solved as a linear program or as a *MILP (mixed integer/linear program)*.

Suppose there is an instruction that is dependent on all of the other instructions in the problem. Formulating an objective function in order to minimise the maximum is simple in this scenario. This is because all that would have to be done would be to minimise the variable representing the ending time of this instruction.

The solution is therefore to introduce a dummy instruction (*edummy*) into the problem that is dependent on all of the other instructions in the problem. This however will introduce a large number of unnecessary dependency constraints. Only a subset of the instructions can be *ending instructions*. Ending instructions are instructions that do not have any dependents. One of these ending instructions will be the last instruction to execute and therefore will determine the ending time of the dummy instruction. This ending dummy instruction is similar to the starting dummy instruction in that it has a duration of zero. A constraint in the form given below must be added for each ending instruction i where x_n represents the ending time of the ending dummy instruction.

$$x_n \geq x_i \tag{4.3}$$

UNIVERSITY
JOHANNESBURG

If the necessary dependency constraints have been added then our objective function is simply

Minimise x_n

A new graph is shown in Figure 4.3 that has the ending dummy instruction added, and as a result only has one ending instruction.

4.3 Justification for the use of continuous variables for instruction times

Scheduling is in general an integer programming problem and therefore the x_i variables should have an integer value restriction. I will show that any optimal solution will always have integer valued makespan even if the problem is solved using continuous variables. Some of the instruction ending times may not be integer valued however they can be

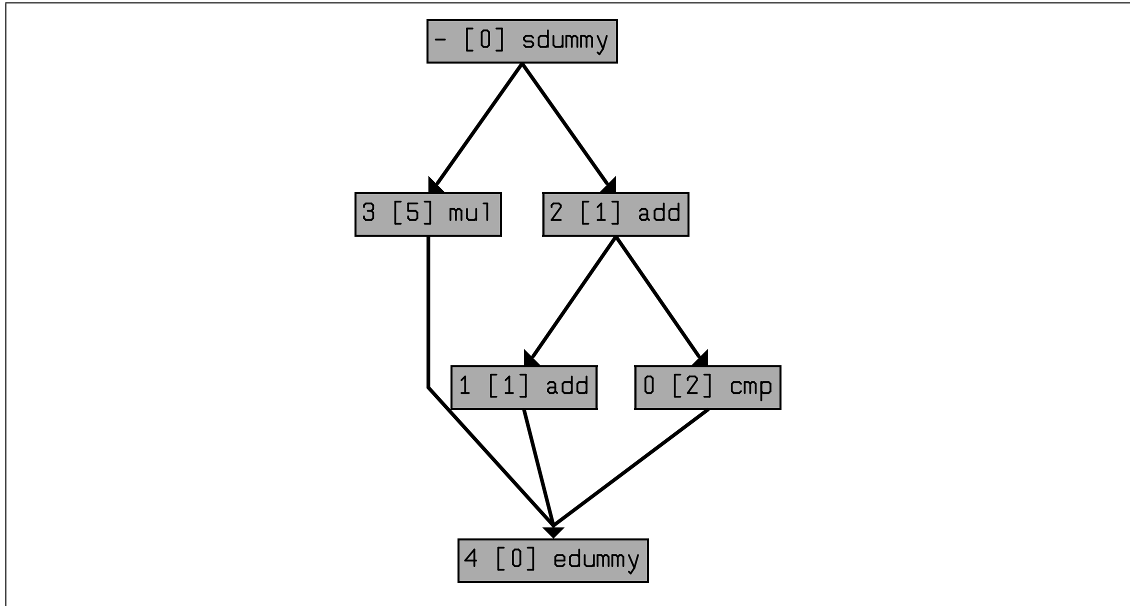


Figure 4.3: *Data dependency graph with both starting and ending dummy instructions.*

truncated should exact instruction ending times be required.

In order to minimise the makespan of the basic block each instruction in the basic block must execute as soon as possible. Starting instructions have a starting time of zero or in other words $x_i = t_i \; \forall \; i$.

However, as we have seen, non-starting instructions have dependencies between them and therefore these dependencies will alter the starting times of the instructions. In order for an instruction to start execution all of the instructions that it depends on must have completed their respective executions. Therefore an instruction can start executing as soon as the instructions that it depends on have ended their execution. The starting time of an instruction will be determined by the durations of the instructions executing before it. To put it more formally the starting time of an instruction is equal to the sum of the durations of a subset of the instructions in the basic block. We stated that durations are always integer values, therefore the sum of a set of durations will also be an integer value. In some circumstances an instruction may have a range of possible ending times that do not affect the makespan of the optimal solution. In such circumstances the ending time of the instruction may drift however all integer values in the range will be valid and therefore instruction ending times can be truncated.

The following section will deal with the fact that there is a limitation on how many instructions may execute in parallel. This limitation will introduce additional constraints however this will not impact the fact that the x_i variables will always have integer values.

The constraints that will be added to limit parallelism behave like dependency constraints so the points made concerning the effects of dependency constraints apply.

4.4 Resource constraints

Without resource constraints the model assumes an unlimited number of execution units. The cost of a path through the graph is the sum of the instruction durations on that path. The makespan of the basic block is equal to the highest cost path through the dependency graph. This type of path is referred to as the *critical path*. Critical paths are well documented as they are of particular interest in project management and as a result the topic is covered in many books dealing with project management, for example the book by Kendall [20]. Calculating the critical path is trivial, however it does not solve our problem since we have a limit imposed on the number of execution units available to us. For example in a computer that has two execution units for executing arithmetic instructions, a maximum of two arithmetic instructions may execute at the same time.

In the following sections I will present two different methods for limiting resource usage. The first is an adaptation of the standard method, shown in Baker [4], for resource usage limitation. The adaptation allows for control of the maximum number of execution units used at any point.

UNIVERSITY OF JOHANNESBURG

The second method employs the concepts discussed in a paper by Greenberg [17]. Again certain changes are made to allow for multiple execution units. This second method of resource limitation is not of much interest on its own, however it can be combined with the first method to reduce the computational time required.

The case where there is a single execution unit will first be discussed followed by a discussion on dealing with multiple execution units. This applies to both the first and second method of limiting resource usage.

4.5 Disjunctive method for eliminating parallelism with a single execution unit

The standard method of restricting resource usage is the method given in several texts (see Baker [4], for example). The standard method of restricting resource usage in MILP models ensures that no two instructions execute at the same time. For example in a Gantt chart like Figure 4.4 the execution of instruction 0 overlaps that of instruction 1. The schedule is valid if there are two or more execution units available to execute the two instructions in parallel.

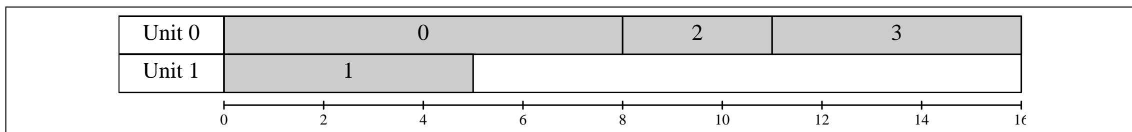


Figure 4.4: *Gantt chart showing four instructions and two execution units.*

In this section we focus on scheduling the instructions where only a single execution unit is present. In other words the resulting schedule ensures that only one instruction executes at a time. Additional constraints must be introduced to accomplish this goal as described by Baker [4].

To ensure that there is no overlap between two instructions i and j we have to ensure that instruction i is executed either before or after instruction j. The x_i variables refer to the ending times of the instructions therefore if we want to use the starting time of an instruction we have to calculate it by subtracting the instruction's duration from its ending time.

Therefore to constrain execution of instruction i to end before the execution of instruction j starts, we can use the constraint given below (Constraint 4.4).

$$x_i \leq x_j - t_j \tag{4.4}$$

Alternatively if we wanted to constrain the execution of instruction i to start after the execution of instruction j ends we could use the following constraint (Constraint 4.5).

$$x_i - t_i \geq x_j \tag{4.5}$$

For no value of x_i and x_j can both constraints (4.4 & 4.5) be satisfied at the same time. Remember that t_i is the duration of instruction i and therefore has to be positive. If $x_i < x_j$ then Constraint 4.5 cannot be satisfied. Similarly if $x_i > x_j$, and t_j is positive, then Constraint 4.4 cannot be satisfied. It is therefore necessary to use a disjunctive constraint pair to enforce either Constraint 4.4 or Constraint 4.5 but not at the same time.

A *disjunctive constraint* pair consists of two constraints where one is satisfied trivially and the other non-trivially. Such constraint pairs create an either-or relationship through the use of a 0-1 variable y and a large positive constant M. If one constraint is satisfied trivially then the other constraint is satisfied non-trivially as shown by Winston [31]. A method for determining a suitable value for M is introduced in Chapter 5.

A *disjunctive term* is an expression responsible for creating the disjunction in a disjunctive constraint pair. A single disjunctive constraint pair will have two different disjunctive terms, My and $M(1-y)$. Remember that y has only two possible values which are 0 and 1. Suppose $y = 1$ then My is overwhelmingly large and $M(1-y) = 0$. Alternatively if $y = 0$ then $My = 0$ and $M(1-y)$ is overwhelmingly large. It is this behaviour that allows one of the two constraints in a disjunctive constraint pair to be satisfied trivially while the other is enforced.

Let:

y_{ij} be a 0-1 variable used to create a disjunctive constraint pair for the instruction pair (i, j).

JOHANNESBURG

Disjunctive terms are added to the Constraints 4.4 & 4.5 to produce a new pair of Constraints 4.6 & 4.7. The new constraint pair is given below.

$$x_i \leq x_j - t_j + My_{ij} \tag{4.6}$$

$$x_i - t_i \geq x_j - M(1 - y_{ij}) \tag{4.7}$$

If Constraints 4.6 & 4.7 are satisfied then either $y_{ij} = 0$ and Constraint 4.4 is satisfied or $y_{ij} = 1$ and Constraint 4.5 is satisfied. Constraints 4.6 & 4.7 must be included for all pairs of instructions that have the possibility of over-using resources. Criteria for determining this are discussed in the following section.

4.6 Locating resource contention

It is important to realise that in realistic circumstances only a subset of the instructions will be able to execute in parallel. Most instructions have a direct dependence or a transitive dependence on most of the other instructions. Therefore we only need to prevent concurrency in a subset of the total number of cases. This property is very important in that it drastically reduces the number of constraints and variables required.

The *resource contention set* R_i of an instruction i is the set of all instructions (including instruction i) that may contribute to the simultaneous use of more execution units than the processor has to offer. Only instructions that can execute in parallel to instruction i are added to R_i. Several criteria are applied in order to determine which instructions are placed in the set.

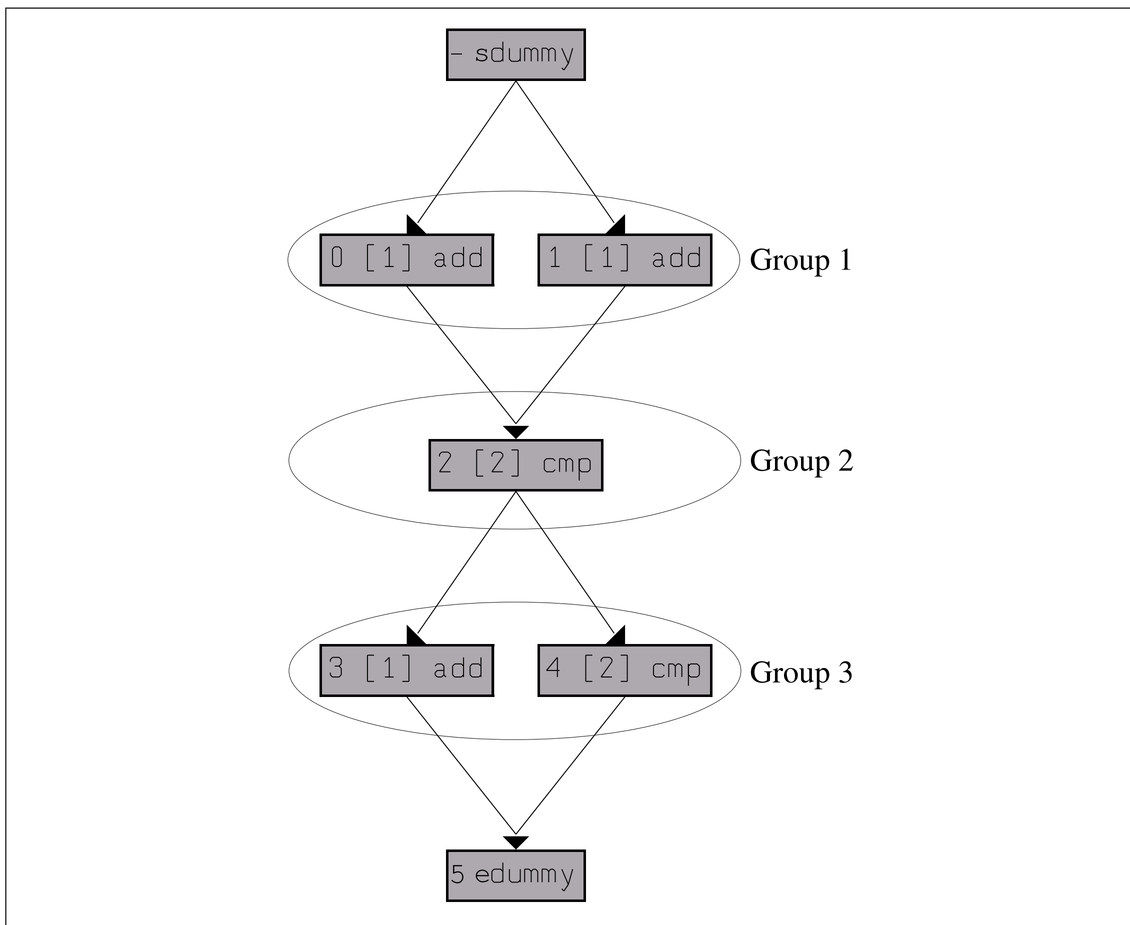


Figure 4.5: *Dependency graph illustrating possible resource contention sets. Excluding the dummy instructions there are only two types of instructions add and cmp. Each node number is a unique identifier for the instruction so that reference can be made to the instruction in the text.*

Assuming instruction j is being evaluated for addition to R_i the following criteria apply:

1. Instruction i must not have a direct or transitive dependency on instruction j as the dependency would prevent instruction i from executing in parallel to instruction j. Similarly when instruction j has a dependency on instruction i.

2. Instruction j must use the same type of execution unit as instruction i. If instruction j uses a different type of execution unit than instruction i then they will not compete for execution units.

A simple example is given in Figure 4.5 and the corresponding resource contention sets in Table 4.1. Dummy instructions are not processed because they do not have concurrency issues. The group references the circled instructions in Figure 4.5. Each group consists of instructions that do not have any dependencies on any other instruction in the group. The contention set lists the instructions in the resource contention set for the instruction specified.

Instruction	Group	Execution unit	Contention set
0	1	ALU	0, 1
1	1	ALU	0, 1
2	2	ALU	2
3	3	ALU	3, 4
4	3	ALU	3, 4

Table 4.1: *Instruction information and resource contention sets for the example given in Figure 4.5.*

UNIVERSITY OF JOHANNESBURG

From Table 4.1 we can conclude that instruction pairs $(0, 1)$ and $(3, 4)$ may execute in parallel on the same type of execution unit. If we have only one *ALU* execution unit then we must prevent these instructions from executing in parallel.

The cardinality of a resource contention set is the maximum number of instructions which require the same type of execution unit that can execute in parallel with instruction i. Instruction i is included in this total.

For any type of instruction there will be one or more execution units supporting it. If the cardinality of the resource contention set is less than or equal to the number of supporting execution units then the problem can be simplified. In such circumstances there are sufficient execution units available and therefore the instructions will never compete for resources. If this is the case then no resource constraints are required for the resource contention set R_i.

4.6.1 Generating constraints from resource contention sets

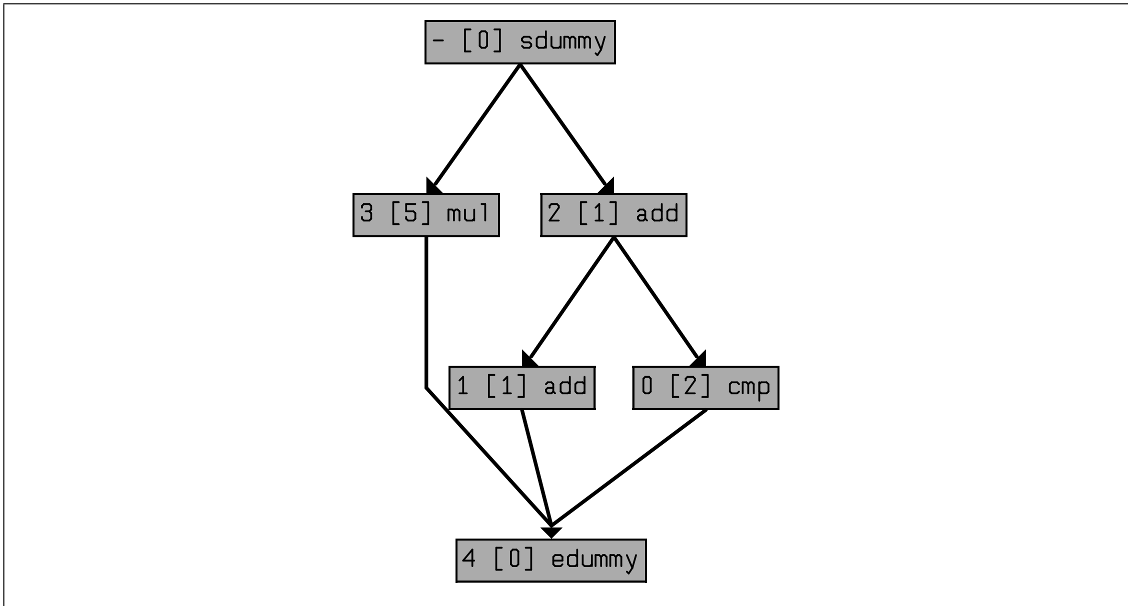


Figure 4.6: *Dependency graph illustrating a more complex program than Figure 4.3.*

Suppose Figure 4.6 is the dependency graph for the program being compiled and the targeted processor has two identical execution units. By applying the process discussed in the previous sections the resource contention sets in Table 4.2 are generated.

Instruction identifier	Resource contention set contents
0	0, 1, 3
1	0, 1, 3
2	2, 3
3	0, 1, 2, 3

Table 4.2: *The resource contention sets for the program given in Figure 4.6.*

At first glance it would appear that instruction 3 can execute in parallel with instructions 0, 1 and 2. However instruction 3 can not execute in parallel with instruction 2 and instruction 1 at the same instant in time because instruction 1 is dependent on instruction 2. The same could be said for instruction 0 as opposed to instruction 1. Instructions 0 and 1 are dependent on instruction 2 and therefore will never execute in parallel with instruction 2. The number of instructions that can execute in parallel may therefore be less than the cardinality of the resource contention set in question. From this it is clear that we need some additional processing of the resource contention sets in order to only generate constraints for instruction pairs that require them.

To determine if there will be contention for the use of the execution units it is necessary to inspect each instruction j in each resource contention set R_i. The intersection of R_i

and R_j is computed to determine which instructions will execute in parallel with both instruction i and instruction j.

If instruction j is in R_i then by definition instruction i must be in R_j. Therefore the instruction pair (i, j) will be considered twice. Clearly this causes unnecessary overhead as well as introducing the need to check the resulting list of instruction pairs for duplicates. This situation can be avoided by only considering the intersection of resource contention sets R_i and R_j if $j > i$. Let $S = \{\ldots, (i, j)\}$ be the set of instruction pairs that require resource limiting constraints.

The algorithm below is used to create a set of instruction pairs that require resource limiting constraints. In the algorithm, R_i represents the resource contention set of instruction i and m represents the number of execution units in the target processor.

Input: Resource contention sets R_i
Output: The set S of instruction pairs (i, j) requiring resource limiting constraints.

```
S = ∅
foreach instruction i
{
    foreach instruction j in R_i
    {
        if j > i
            if |R_i ∩ R_j| > m
                S = S ∪ {(i, j)}
    }
}
```

UNIVERSITY OF JOHANNESBURG

Algorithm 4.1.

Table 4.3 illustrates the steps taken when applying the algorithm to the graph given in Figure 4.6 which was a two execution unit problem. Steps where $j \leq i$ have been omitted for brevity.

In Algorithm 4.1 we computed $R_i \cap R_j$ however we were only interested in the cardinality of $R_i \cap R_j$. We can directly compute $|R_i \cap R_j|$ without first calculating $R_i \cap R_j$ and therefore reduce storage and computation requirements. Algorithm 4.2 calculates $|R_i \cap R_j|$ directly leaving the result in c.

i	j	R_i	R_j	$R_i \cap R_j$	**Add to** S
0	1	{0, 1, 3}	{0, 1, 3}	{0, 1, 3}	Yes
	3	{0, 1, 3}	{0, 1, 2, 3}	{0, 1, 3}	Yes
1	3	{0, 1, 3}	{0, 1, 2, 3}	{0, 1, 3}	Yes
2	3	{2, 3}	{0, 1, 2, 3}	{2, 3}	No

Table 4.3: *Steps for creating the resource contention sets for the program given in Figure 4.6.*

Input: Resource contention set pair R_i and R_j
Output: $c = |R_i \cap R_j|$

sort(R_i)
sort(R_j)

$p = q = c = 0$

while $p < |R_i|$ **and** $q < |R_j|$
{
  a = p-th Instruction in R_i
  b = q-th Instruction in R_j
  if $a == b$
  {
    p++
    q++
    c++
  }
  else if $a < b$
    p++
  else
    q++
}

Algorithm 4.2.

The purpose of this section was to determine which instruction pairs require resource limiting constraints. We have also shown that some instructions will never participate in the over-use of execution units and therefore do not require constraints to prevent over-use from occurring.

4.6.2 Adaptations for multiple identical resources

In order to allow for controlled parallelism of instructions in the formulation given above it is necessary to change the nature of the constraint pairs. In the single resource problem the Constraints 4.6 & 4.7 force either Constraint 4.4 or Constraint 4.5 to be satisfied. It is this behaviour that forces instruction i to either execute before or after instruction j but not at the same time.

In the case of scheduling with multiple execution units some instructions will execute in parallel with other instructions. However the extent of the parallelism must be controlled to ensure that the solution will not use more resources than are available at any given point.

In order for two instructions to execute in parallel the constraint pair associated with the two instructions must be relaxed. A constraint pair is required for each pair of instructions (i, j) in S.

Let:

v_{ij} be a variable used to relax the constraint pair for the instruction pair $(i, j) \in S$.

A Mv_{ij} term is introduced to the constraints given in the single resource formulation (Constraints 4.6 & 4.7) to produce the following.

$$x_i - Mv_{ij} \leq x_j - t_j + My_{ij} \ \forall j \in R_i \text{ where } j \neq i \tag{4.8}$$

$$x_i - t_i + Mv_{ij} \geq x_j - M(1 - y_{ij}) \ \forall j \in R_i \text{ where } j \neq i \tag{4.9}$$

When the variable v_{ij} is positive then the constraint pair will always be satisfied regardless of whether the original Constraints (4.6 & 4.7) would be satisfied. When $v_{ij} > 0$ then the left hand side of Constraint 4.8 is overwhelmingly negative. Since the left hand side of Constraint 4.8 must be less than or equal to the right it will be satisfied trivially. Similarly the left hand side of Constraint 4.9 will be overwhelmingly positive and will therefore always be greater than the right hand side.

However this takes us back to the original situation where the model assumes infinite resources. However for the two instructions to execute in parallel the associated v_{ij} variable

will have to be positive. If v_{ij} is not positive then constraints 4.6 & 4.7 are enforced.

4.6.3 Absolute difference approach to constraint relaxations

In the previous section we saw that we can affect the parallelism of an instruction pair (i, j) through an associated v_{ij} variable. In this section constraints for the v_{ij} variables are introduced.

A new variable e_i is introduced for each instruction i that may contribute to the overuse of execution units. This variable identifies the execution unit used by the instruction. In this chapter it is assumed that all execution units are identical, the following chapter deals with different types of execution units. Assume execution units are numbered from zero increasing by one for each additional execution unit present. Variable e_i is an integer variable because execution units are discrete entities and each execution unit must be uniquely identified. An upper bound is placed on the variable e_i to restrict it to a value less than the number of execution units that support instruction i. Since execution units are numbered from 0 and variable e_i has an integer value restriction, this constraint ensures that e_i has m unique values where m is the number of executions units that support instruction i.

When v_{ij} is positive the constraints will be relaxed. If instruction i executes on the same execution unit as instruction j then v_{ij} must be equal to zero. If instruction i executes on a different execution unit than instruction j then v_{ij} must be positive. Therefore if $e_i \neq e_j$ then v_{ij} must be positive but when $e_i = e_j$ then v_{ij} must be equal to zero.

Representing this behaviour in terms of constraints can be achieved by utilising the absolute value of the difference of two variables. If $e_i \neq e_j$ then $|e_i - e_j| \geq 1$. In terms of v_{ij} the constraint becomes

$$v_{ij} = |e_i - e_j| \tag{4.10}$$

The absolute difference constraint ensures that the variable v_{ij} is equal to the absolute difference between the two variables e_i and e_j. If e_i is equal to e_j then this would indicate that instruction i is executing on the same execution unit as instruction j and therefore v_{ij} is equal to zero to force instructions i and j to not execute in parallel. Alternatively if e_i is not equal to e_j then $|e_i - e_j| \geq 1$, $v_{ij} \geq 1$ and therefore the resource constraints will be satisfied trivially by virtue of v_{ij} being positive. A method for implementing absolute

difference constraints will be introduced in the following chapter.

4.6.4 The mathematical program for the example

At this point we can formulate a mathematical program to solve the problem first introduced in Figure 4.6. This program consists of three parts: the objective, the dependency constraints and the resource constraints. The ending dummy instruction is instruction 4.

The objective is simply:
Minimise x_4

Subject to the following constraints:

Data dependency constraints (Constraint 4.1).

$$\begin{aligned} x_1 - 1 &\geq x_2 \\ x_0 - 2 &\geq x_2 \end{aligned}$$

Starting instruction constraints (Constraint 4.2).

UNIVERSITY OF JOHANNESBURG

$$\begin{aligned} x_3 &\geq 5 \\ x_2 &\geq 1 \end{aligned}$$

Ending instruction constraints (Constraint 4.3).

$$\begin{aligned} x_4 &\geq x_0 \\ x_4 &\geq x_1 \\ x_4 &\geq x_3 \end{aligned}$$

In Section 4.6 parallel instruction pairs that may contribute to resource overuse were identified. The example used was also Figure 4.6 and three instruction pairs were identified, (0,1), (0,3) and (1,3). Each e_i variable has an upper bound of 1 therefore e_i variables have two possible values 0 and 1. The two values represent the two execution units used in the example.

Resource limiting constraints for instruction pair (0, 1).

$$\begin{aligned} x_0 - Mv_{01} &\leq x_1 - 1 + My_{01} \\ x_0 - 2 + Mv_{01} &\geq x_1 - M(1 - y_{01}) \\ v_{01} &= |e_0 - e_1| \end{aligned}$$

Resource limiting constraints for instruction pair (0, 3).

$$\begin{aligned} x_0 - Mv_{03} &\leq x_3 - 5 + My_{03} \\ x_0 - 2 + Mv_{03} &\geq x_3 - M(1 - y_{03}) \\ v_{03} &= |e_0 - e_3| \end{aligned}$$

Resource limiting constraints for instruction pair (1, 3).

$$\begin{aligned} x_1 - Mv_{13} &\leq x_3 - 5 + My_{13} \\ x_1 - 1 + Mv_{13} &\geq x_3 - M(1 - y_{13}) \\ v_{13} &= |e_1 - e_3| \end{aligned}$$

The y_{ij} variables are 0-1 variables and the e_i variables are restricted to integer values.

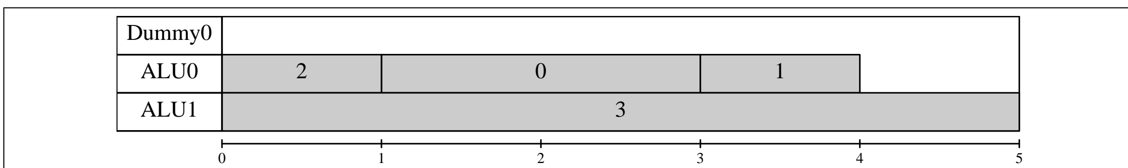


Figure 4.7: *The optimal solution in the form of a Gantt chart. ALU0 and ALU1 are the two real execution units and Dummy0 is the single execution unit supporting dummy instructions. Since dummy instructions have a duration of 0 they do not appear on the Gantt chart.*

In the optimal solution the values for the x_i variables are as follows; $x_0 = 3$, $x_1 = 4$, $x_2 = 1$, $x_3 = 5$ and $x_4 = 5$. The optimal solution is expressed in a Gantt chart in Figure 4.7.

4.7 An alternative method for a special case

This section describes an alternative set of constraints for limiting parallelism, however it is only applicable in a particular circumstance. The advantage of this alternative method is that it does not use the absolute difference of two variables. This is an advantage in terms of performance since the absolute difference is computationally expensive to implement as will be shown in the following chapter. This method is applicable in cases where the maximum number of instructions that can execute, at a given point in the program, is equal to $m + 1$. Both the general case method and the special case method are applied on a per instruction basis, therefore a combination of both methods can be used for a single program.

Suppose the v_{ij} variables are 0-1 variables and that they have no relationship to the e_i variables. If $v_{ij} = 0$ then instructions i and j cannot execute in parallel. The opposite is true if $v_{ij} = 1$ as instructions i and j can execute in parallel.

For example there are three instructions numbered 0, 1 and 2 which will execute on a two execution unit processor ($m = 2$). If there is a v_{ij} variable for every possible combination of instructions then there will be three variables v_{01}, v_{02} and v_{12}. If v_{01}, v_{02} and v_{12} are all equal to 0 then none of the instructions are executing in parallel. However if v_{01} and v_{02} are equal to 1 but v_{12} is equal to 0 then a maximum of two execution units are being used. The Gantt chart in Figure 4.8 illustrates this scenario.

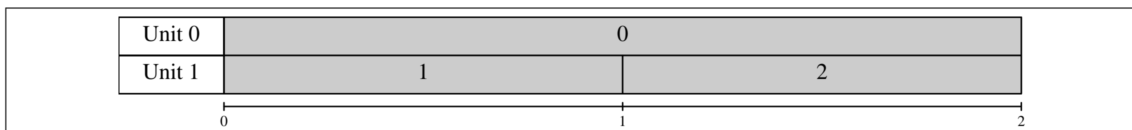


Figure 4.8: *A schedule where v_{01} and v_{02} are equal to 1 and v_{12} is equal to 0.*

Therefore if we wanted to limit the number of instructions executing in parallel to two we could simply add a constraint stating the following:

$$v_{01} + v_{02} + v_{12} \leq 2$$

This would ensure that at least one of the three v_{ij} variables would have to be equal to zero since v_{ij} variables are either equal to 1 or 0. Therefore a maximum of two instructions could execute in parallel. Using v_{ij} variables in this way allows us to prevent resource overuse. However it requires that we add a constraint for every combination of v_{ij} variables where resource overuse is possible. The number of constraints that would be required for an average problem is therefore very large. However in the case where the

number of instructions that can execute in parallel is greater than the number of execution units by one, then only one constraint is required. In the remainder of this section only this special case will be discussed.

The resource contention sets R_i are again used to determine which instructions require resource limiting constraints. The example given in Figure 4.6 along with the associated resource contention sets is used as an example in the remainder of this section.

Algorithm 4.3 is used to generate a set T_i for each instruction i. Initially T_i contains a single instruction i. Each subsequent element in the set T_i is an instruction j from R_i where $|R_i \cap R_j| > m$. Algorithm 4.3 is an adaptation of Algorithm 4.1.

Input: Resource contention set information R_i and the number of execution units in the processor m.

Output: Set T_i containing all of the instructions that may overuse resources when executing in parallel with instruction i

foreach instruction i
{
  $T_i = \{i\}$
  foreach instruction j in R_i
  {
    if $i \neq j$
      if $|R_i \cap R_j| > m$
        $T_i = T_i \cup \{j\}$
  }
}

Algorithm 4.3.

UNIVERSITY OF JOHANNESBURG

When the algorithm is applied to the program described by Figure 4.6 the steps in the table below are taken.

i	j	R_i	R_j	$R_i \cap R_j$	T_i
0	1	$\{0,1,3\}$	$\{0,1,3\}$	$\{0,1,3\}$	$\{0,1\}$
	3	$\{0,1,3\}$	$\{0,1,2,3\}$	$\{0,1,3\}$	$\{0,1,3\}$
1	0	$\{0,1,3\}$	$\{0,1,3\}$	$\{0,1,3\}$	$\{1,0\}$
	3	$\{0,1,3\}$	$\{0,1,2,3\}$	$\{0,1,3\}$	$\{1,0,3\}$
3	0	$\{0,1,2,3\}$	$\{0,1,3\}$	$\{0,1,3\}$	$\{3,0\}$
	1	$\{0,1,2,3\}$	$\{0,1,3\}$	$\{0,1,3\}$	$\{3,0,1\}$

Table 4.4: *The steps taken when applying the algorithm to the problem described by Figure 4.6. Steps where $i = j$ or $|R_i \cap R_j| \leq m$ have been omitted for brevity.*

After the algorithm has been applied a T_i set will exist for each instruction. Duplicate sets are removed as well as sets that have a cardinality less than or equal to m. This special case method cannot be employed in cases where the cardinality of T_i is greater than $m + 1$. In such circumstances the general case method must be used instead. The only set remaining from the example is T_0 containing the following elements $\{0, 1, 3\}$. In practice there may be more than one resulting set.

For each remaining set T_i a single constraint will be included. Algorithm 4.4 is applied to each remaining T_i set in order to determine which v_{ij} variables will need to be present in the constraint.

Input: Resource contention set T_i.
Output: The set S of instruction pairs (i, j) requiring resource limiting constraints.

foreach instruction j in T_i
{
    foreach instruction k in T_i
    {
        if $k > j$
            include term v_{jk} in a constraint for instruction i
    }
}

Algorithm 4.4.

The constraints generated by Algorithm 4.4 have the general form given below in Constraint 4.11.

$$v_{jk} + \ldots \leq \frac{|T_i|(|T_i| - 1)}{2} - 1 \tag{4.11}$$

The right hand side of the constraint must be equal to the number of terms on the left minus 1. The number of terms can be determined by calculating the total number of unique pairs in the set T_i by evaluating $\frac{n(n-1)}{2}$, where $n = |T_i|$. In practice it is possible to just count the number of terms generated.

If we apply Algorithm 4.4 to T_0 we discover that we need a constraint that prevents v_{01}, v_{03} and v_{13} from all being equal to 1. The final form of the parallelism limiting constraint

required for the example problem is given below.

$$v_{01} + v_{03} + v_{13} \leq 2 \tag{4.12}$$

This constraint is quite logical when we consider that in the program given in Figure 4.6 only one opportunity for resource overuse exists. If instructions 0, 1, and 3 all attempt to execute in parallel.

This special case method can be used in conjunction with the general case method as mentioned at the beginning of this section. The special case method would first be attempted and only instructions where $|T_i| > m + 1$ would use the general case method. Algorithm 4.5 is an adaption of Algorithm 4.1 for integration with the special case method.

Input: Resource contention sets R_i and special case sets T_i.
Output: The set S of instruction pairs (i, j) requiring general case resource limiting constraints.

$S = \emptyset$
foreach instruction i
{
    if $|T_i| > m + 1$
        foreach instruction j in R_i
        {
            if $j > i$
                if $|R_i \cap R_j| > m$
                    $S = S \cup \{(i, j)\}$
        }
}

Algorithm 4.5.

An additional advantage of the special case method is that it does not require an e_i variable. In practice the special case method can be supported with very little additional work since the code necessary is almost identical to that of the general case.

4.8 A branch and bound approach to limiting parallelism

When we first introduced resource constraints (for the single resource problem) we showed that they are an adaptation of the dependency constraints. That is to say that they create a dependency when they are non-trivially satisfied. This dependency is necessary to ensure that the instructions in the pair do not execute in parallel and therefore the same execution unit can be used for both instructions.

Greenberg showed [17] that branch and bound enumeration could be used instead of disjunctive constraints when solving the job shop problem. The technique he advocated is that the problem is modelled as a plain linear program without any resource limiting constraints. The solution is checked for the first instance of two jobs (a, b) executing in parallel (the job shop problem is a single resource problem) and two subproblems would be generated. An additional dependency constraint is added to both subproblems. In the first subproblem it states that job a must execute before job b. Similarly job b must execute before job a in the second subproblem. As soon as a solution to a subproblem without any parallelism is found then its makespan can be used as an upper bound on the problem.

Regarding the scalability of this approach Greenberg [17] states the following:

> Experience suggests (though not verified) that as the size of the problem increases, the ratio of L.P. problems solved to total number of solutions diminishes. However, since for larger scheduling problems larger L.P. problems are solved, the ratio of computer time to possible solutions appears to remain fairly constant at about 4.

UNIVERSITY
JOHANNESBURG

The advantage of this method is that it prevents unnecessary constraints from being added to the problem. If we were to apply this method to the parallel machine problem (the one we are trying to solve) then modelling the problem as a linear program is unnecessary. Unlike the job shop problem we can easily minimise the ending times of the instructions in the program. Algorithm 4.6 is a recursive algorithm that starts with the ending instruction (provided by the caller).

Input: Instruction durations t_i and instruction dependencies. All ending times x_i are initialised to -1 to indicate unsolved.
Output: Instruction ending times x_i.

```
Minimise(i) {
    p = 0
    foreach instruction j on which i depends
    {
        if x_j == -1
            Minimise(j)
        if x_j > p
            p = x_j
    }
    x_i = p + t_i
}
```

Algorithm 4.6.

For each instruction i Algorithm 4.6 first evaluates each instruction j on which i depends. If each instruction j has been evaluated then variable x_j will be the correct ending time for instruction j. Instruction i must start execution after all of instructions on which it depends have completed execution. Therefore instruction i must start executing at the maximum of the ending times of the instructions on which instruction i depends. The duration of instruction i, t_i, is added to the starting time of instruction i to produce the ending time x_i. The recursive nature of this algorithm ensures that all of the instructions in the graph are evaluated and the ending times are finalised in a top down fashion.

Determining if resource overuse is taking place is slightly more complicated in our case since we are not just checking for parallelism. The algorithm given below simulates execution of the program and therefore can determine when overuse takes place as well as which instructions are involved. An additional use of this algorithm is for assigning execution units to instructions, however this will be dealt with in the following chapter. In the algorithm the set E contains all of the instructions that are currently executing hence it is initialised as an empty set.

Input: Instruction durations t_i and ending times x_i. The instructions are sorted by their starting times $(x_i - t_i)$.
Output: Instructions involved in the execution unit overuse incident if applicable.

```
E = ∅

foreach instruction i
{
    // Remove all instructions in E that have finished executing
    foreach instruction j ∈ E
    {
        if x_j ≤ x_i − t_i
            E = E − {j}
    }
    E = E ∪ {i}
    if |E| > m
        All of the instructions in E are scheduled to execute in parallel but there are
        insufficient execution units available. Further evaluation cannot take place.
}
```

Algorithm 4.7.

The steps given below describe a branch and bound approach for solving the parallel machine problem. This method is an adaptation of the method developed by Greenberg for solving the job shop problem. The process starts with the basic problem which only contains dependency constraints.

1. The x_i variables are minimised using Algorithm 4.6 which also results in the makespan of the schedule being available in the form of the ending instruction's time. This subproblem is added to the list of unbranched subproblems.

2. A subproblem is selected and removed from the list of unbranched subproblems. A discussion on criteria for selection follows the list of steps.

3. The selected subproblem is checked for resource overuse through the use of Algorithm 4.7. If $|E| > m$ then no further examination of the selected subproblem takes place. Two different actions are taken depending on whether or not resource overuse is present.

    (a) If there is no resource overuse in the subproblem then it is a candidate optimal solution. If its makespan is less than the current upper bound then it may

result in the optimal schedule for the original problem. The makespan of this schedule is used as a new upper bound. All unbranched subproblems that have a makespan greater than or equal to the new upper bound are discarded.

(b) If resource overuse is present then subproblems must be generated for all pairs of instructions involved in a resource overuse incident. As in the original job shop problem each subproblem has an additional dependency. As with the initial problem the x_i variables of each subproblem are minimised using Algorithm 4.6. Each new subproblem is added to the list of unbranched subproblems.

4. The process is repeated from step 2 until there are no more unbranched subproblems. The last subproblem to result in a new upper bound must be the optimal solution to the original problem since it has the lowest makespan.

There are many different methods for selecting sub-problems or in this case schedules for processing. The *best first* approach states that the schedule that has the lowest makespan (the best candidate) should be the schedule that is selected for processing. In practice any method could be used to determine the suitability of the schedule not just the makespan. Branch and bound algorithms are a subset of state space enumeration which is well covered by many books concerned with solving complex problems such as Nilsson [24].

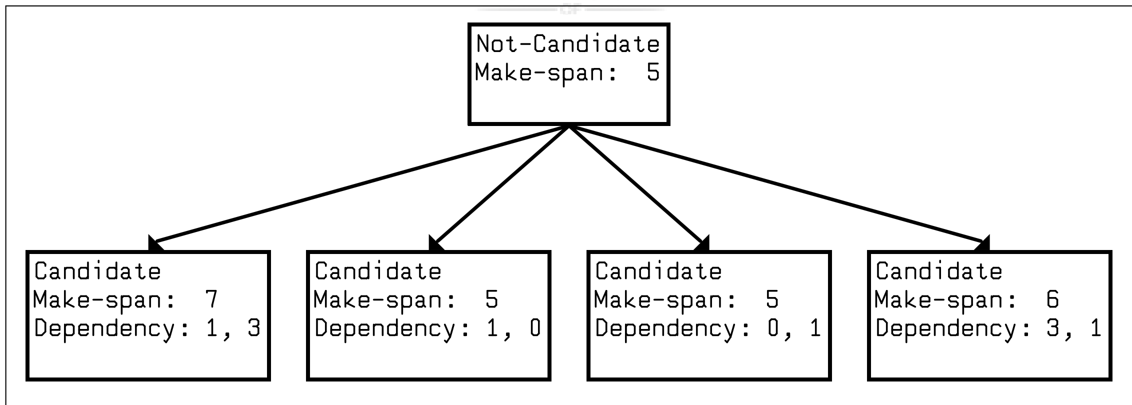


Figure 4.9: *Each node represents a schedule where the title* Not-Candidate *refers to a schedule that has resource overuse. For nodes that do not overuse resources a* Candidate *title is used instead. Each child node is identical to the parent node except that the child node has an extra dependency which is given in the form* i, j*. A dependency of* $1, 3$ *would mean that instruction* 3 *can only start execution after instruction* 1 *has finished executing. Each schedule has a* makespan *field that shows the makespan of the schedule in question.*

Figure 4.9 shows the problem tree or state space for the program given in Figure 4.6 which is the same program used in previous examples. The branch and bound approach

is implemented in the compiler described in later chapters, however it is also the basis for a heuristic algorithm introduced in Chapter 5. This algorithm is used to generate bounds on problem variables to reduce solver computational time.

In order to implement a compiler using the model introduced in this chapter, several aspects need to be developed further. In particular we have to determine a suitable value for M. Additionally we have to find a way to implement absolute difference constraints using existing linear solvers. The following chapter will develop solutions to these problems.

UNIVERSITY
OF
JOHANNESBURG

Chapter 5

Implementing instruction scheduling

This chapter bridges the gap between the model introduced in the previous chapter and an actual implementation. The previous chapter presents a model that is technically correct. However, it employs concepts that must be clarified in an actual implementation, for example a suitable value for the large constant M must be determined. There are two parts to this chapter, the first part is concerned with translating a problem into something a MILP solver can optimise. The second part focuses on reducing the computational time required to solve the problem.

UNIVERSITY
OF
JOHANNESBURG

5.1 Binding a variable to the absolute difference of two variables

In order to efficiently model the problem at hand it is useful to bind a variable d to the absolute difference between two variables. The equation below (Equation 5.1) illustrates the relationship between d and the two variables a and b.

$$d = |a - b| \tag{5.1}$$

The binding of d to the absolute difference $|a - b|$ has to be implemented using techniques available to MILP solvers. Therefore it is implemented by enforcing both an upper and lower bound on d. There are two pairs of constraints, one pair to enforce a lower bound on d and another pair to enforce an upper bound on d. It should be noted that a, b and d

are restricted to non-negative values.

Consider the following constraint pair for enforcing a lower bound on d in the form $d \geq |a - b|$.

$$d \geq a - b \quad (5.2)$$

$$d \geq b - a \quad (5.3)$$

Therefore

$$d \geq \max\{a - b, b - a\} = |a - b| \quad (5.4)$$

The only difference between the two constraints in the pair is the reversal of roles in regards to variables a and b. The behaviour of the constraint pair can be investigated by evaluating the three possible scenarios that exist.

Scenario 1	$a < b$	$d \geq \max\{a - b, b - a\}$ therefore $d \geq b - a$.
Scenario 2	$a > b$	$d \geq \max\{a - b, b - a\}$ therefore $d \geq a - b$.
Scenario 3	$a = b$	$d \geq \max\{a - b, b - a\}$ therefore $d \geq 0$.

Although placing a lower bound of the absolute difference on a variable is relatively simple the same is not true for an upper bound.

5.1.1 Why the upper bound technique will not work

Based on the formulation for enforcing the lower bound on d to enforce an upper bound on d of $|a - b|$ we would apply the following constraints.

$$d \leq a - b \quad (5.5)$$

$$d \leq b - a \quad (5.6)$$

Therefore

$$d \leq \min\{a - b, b - a\} = -|a - b| \quad (5.7)$$

If $a > b$ then $a - b > 0$ and therefore $a - b = |a - b|$. However this also means that $b - a < 0$ and therefore $b - a = -|a - b|$. Similarly if $a < b$ then $b - a = |a - b|$ and $a - b = -|a - b|$.

Whether $a > b$ or $a < b$ one of the two terms $a - b$ or $b - a$ will be equal to $|a - b|$. The other term will be equal to $-|a - b|$. Clearly this is a problem because both constraints enforce an upper bound on d and both constraints must be satisfied.

Clearly enforcing an upper bound on d cannot be implemented using these constraints. It turns out that enforcing the upper bound on d is far more difficult than enforcing the lower bound.

5.1.2 The difficulty in enforcing the absolute difference as an upper bound

Through out this discussion some assumptions are made concerning the types of variables and constraints that may be used. All variables, unless otherwise specified, have continuous values. All constraints are linear and hence the mathematical program can be solved as a linear program or as a MILP.

UNIVERSITY OF JOHANNESBURG

Suppose the non-linear constraint $|a - b| \geq 3$ could be replaced with one or more linear constraints that have the same effect on the solution. If that were the case then the following problem could be solved as a linear program.

$$\text{Minimise } z = a + b$$

$$\begin{aligned} |a - b| &\geq 3 \\ a &\geq 0 \\ b &\geq 0 \end{aligned}$$

In the above problem two optimal solutions can be identified by means of inspection. In the first solution $a = 3$ and $b = 0$. In the second solution $a = 0$ and $b = 3$.

If there is more than one optimal solution then there are an infinite number of optimal solutions (a basic principle of linear programming discussed by Winston [31] among others). This holds true for any problem containing only continuous variables and linear constraints.

The presence of two and only two optimal solutions in the above problem contradicts the point that there is either one optimal solution or an infinite number of optimal solutions. This implies that the supposition that the absolute difference constraint could be replaced with one or more linear constraints was false. In order to employ constraints that utilise the absolute difference of two variables it is necessary to formulate the problem as a MILP (mixed integer/linear program). It may also be possible to implement absolute difference with non-linear programming (NLP) however this approach has not been investigated.

5.1.3 Using 0-1 variables to implement the upper bound

In the analysis (Section 5.1.1) of the failed upper bound constraints we observed that of the two constraints in the upper bound pair, one constraint enforces the correct upper bound. However the other constraint in the pair enforces $-|a-b|$ as an upper bound on d and therefore the problem becomes infeasible except when $a = b$. The constraints can be modified to accomplish the original goal, which was to enforce an upper bound on d, provided the program is solved as a MILP instead of as a pure linear program. In order to achieve this goal we need to relax the constraint stating that $d \leq -|a-b|$ (from Constraint pair 5.5 & 5.6). A disjunctive constraint formulation can be used to achieve this goal.

A disjunctive term is added in order to allow one of the two constraints in the pair to be satisfied trivially. In order to create the disjunction a 0-1 variable u is introduced. The large constant M is used in the same capacity as before.

The following constraint pair enforces the relationship $d \leq |a-b|$ by virtue of the fact that one of the two constraints will have to be satisfied trivially. The other will enforce the correct upper bound on d.

$$d \leq a - b + M(1-u) \tag{5.8}$$

$$d \leq b - a + Mu \tag{5.9}$$

If $a > b$ then $a - b > 0$ and $b - a < 0$. If $u = 0$ then constraint 5.9 requires that $d \leq b - a$ however this is impossible since $b - a < 0$ so this constraint can only be satisfied if $u = 1$. If $u = 1$ then Constraint 5.8 requires that $d \leq a - b$ where $a - b = |a-b|$ since $a > b$.

Similarly if $a < b$ then $a - b < 0$ and $b - a > 0$. If $u = 1$ then constraint 5.8 requires that $d \leq a - b$ however this is impossible since $a - b < 0$ so this constraint can only be satisfied if $u = 0$. If $u = 0$ then Constraint 5.9 requires that $d \leq b - a$ where $b - a = |b - a|$ since $a < b$.

If $a = b$ then $a - b = 0$ and $b - a = 0$. Constraints 5.8 & 5.9 can both be satisfied non-trivially. However in this scenario $d = 0$ as this is the only possible value for d.

5.2 Exceptions to the requirement for both upper and lower bounds

In some circumstances it may not be necessary to enforce both upper and lower bounds. The advantage of this is that fewer constraints and/or variables will be required to formulate the problem.

The problem below illustrates a typical scenario where the absolute difference of two variables is used in a constraint. In this problem only the upper bound constraints are necessary.

UNIVERSITY OF JOHANNESBURG

$$\text{Minimise } z = a + b$$

$$\begin{aligned} |a - b| &= d \\ d &\geq 3 \\ a &\geq 0 \\ b &\geq 0 \end{aligned}$$

Without the constraints $|a - b| = d$ and $d \geq 3$ both a and b would be equal to zero in the optimal solution. This is because it is a minimisation problem and $d \geq 3$ creates a lower bound on $|a - b|$. There is no constraint enforcing an upper bound on $|a - b|$.

In order to illustrate the point the constraint containing the absolute difference has been converted as discussed in the section on using 0-1 variables to formulate the problem. Constraints for the lower bound on d have not been included because the aim of this section is to show that they are not needed in this circumstance.

$$\text{Minimise } z = a + b$$

$$\begin{aligned} d &\leq a - b + M(1-u) \\ d &\leq b - a + Mu \\ d &\geq 3 \\ a &\geq 0 \\ b &\geq 0 \end{aligned}$$

A lower bound on $|a-b|$ makes little sense as it would not alter the optimal solution. The objective function states that a and b are to be reduced as much as possible so a constraint stating that d must be less than or equal to 3 has no effect.

5.3 Determining a suitable value for the large constant value M

Previously M was defined as a very large constant value. However, in order to implement constraints where M is present it is necessary to find a suitable value for M. If the value chosen for M is too small then the constraints where it is used will not have the desired effect. It may be tempting to make M extremely large except that it could cause numerical difficulties for the solver. To choose a suitable value for M we must evaluate both the problem being solved as well as the constraints where M is used.

The large constant M is present in three pairs of constraints, 4.6 & 4.7, 4.8 & 4.9 and 5.8 & 5.9. Each constraint must be evaluated to determine how large M needs to be. In each of the six constraints M can be multiplied with either 0 or 1. In cases where M is multiplied with 0 the value of M is unimportant leaving only the cases when M is multiplied with 1. In each of the constraints it will be assumed that M is multiplied with 1 regardless of the actual expression that M is multiplied with.

5.3.1 Constraints 4.6 & 4.7

Constraints 4.6 & 4.7 are repeated below.

$$x_i \leq x_j - t_j + My_{ij}$$

$$x_i - t_i \geq x_j - M(1 - y_{ij})$$

We known several facts about the variables present in these constraints. We know that x_i and t_i all have non-negativity restrictions, $x_i \geq t_i$ and that $x_j \geq t_j$. Let p represent the largest possible value for x_i which implies $p \geq x_i$.

In Constraint 4.6 the largest possible difference between the left and right sides would be in the case where $x_i = p$ and $x_j - t_j = 0$. If that were the case then Constraint 4.6 would reduce to

$$M \geq p, \tag{5.10}$$

if $y_{ij} = 1$.

Similarly in Constraint 4.7 the largest possible difference between the left and right sides would be the case where $x_i - t_i = 0$ and $x_j = p$. If that were the case then Constraint 4.7 would reduce to

$$M \geq p, \tag{5.11}$$

if $y_{ij} = 0$.

5.3.2 Constraints 4.8 & 4.9

Constraint pair 4.8 & 4.9 (repeated below) is identical to Constraint pair 4.6 & 4.7 except that Constraints 4.8 & 4.9 also have Mv_{ij} terms.

$$x_i - Mv_{ij} \leq x_j - t_j + My_{ij} \quad \forall j \in R_i \qquad \text{where} \qquad j \neq i$$

$$x_i - t_i + Mv_{ij} \geq x_j - M(1 - y_{ij}) \quad \forall j \in R_i \qquad \text{where} \qquad j \neq i$$

If v_{ij} is equal to 0 then Constraints 4.8 & 4.9 are identical to Constraints 4.6 & 4.7. The requirements for the value of M in respect to Constraints 4.6 & 4.7 will also hold for Constraints 4.8 & 4.9.

In Constraint 4.8 the largest possible difference between the left and right sides is where

$x_i = p$ and $x_j - t_j = 0$. If v_{ij} is equal to 1 then Constraint 4.8 would reduce to

$$\begin{aligned} p - M &\leq M \\ M &\geq \frac{p}{2}, \end{aligned} \tag{5.12}$$

if $y_{ij} = 1$ and $v_{ij} = 1$.

Similarly in Constraint 4.9 the largest possible difference between the left and right sides is where $x_j = p$ and $x_i - t_i = 0$. If v_{ij} is equal to 1 then Constraint 4.9 would reduce to

$$\begin{aligned} M &\geq p - M \\ M &\geq \frac{p}{2}, \end{aligned} \tag{5.13}$$

if $y_{ij} = 0$ and $v_{ij} = 1$.

Both Constraint 4.8 and Constraint 4.9 have the same requirements in terms of M which is $M \geq \frac{p}{2}$ and $M \geq p$. Clearly this pair of inequalities can be reduced to $M \geq p$.

5.3.3 Constraints 5.8 & 5.9

UNIVERSITY
JOHANNESBURG

Constraints 5.8 & 5.9 (repeated below) are used to enforce an upper bound on a variable d. The upper bound is equal to the absolute difference of two variables a and b.

$$d \leq a - b + M(1 - u)$$

$$d \leq b - a + Mu$$

Let q represent the largest possible value for a and b. When d was first introduced it was defined as being equal to $|a - b|$.

The only scenario where $d \leq a - b$ will not be satisfied is when $b > a$ so suppose $b = q$ and $a = 0$. Since d was defined as being equal to $|a - b|$ we can substitute d with q because the $|a - b|$ will always be less than or equal to q. Constraint 5.8 would therefore reduce to

$$M \geq 2q \tag{5.14}$$

if $u = 0$.

Similarly the only scenario where $d \leq b - a$ will not be satisfied is when $a > b$ so suppose $a = q$ and $b = 0$. Since d was defined as being equal to $|a - b|$ we can substitute d with q because the $|a - b|$ will always be less than or equal to q. Constraint 5.9 would therefore reduce to

$$M \geq 2q \tag{5.15}$$

if $u = 1$.

Constraints 5.8 & 5.9 are used to implement Constraint 4.10 which states that $v_{ij} = |e_i - e_j|$. Therefore $a = e_i$ and $b = e_j$ and q is the number of execution units. If there are different types of execution units then q must be greater than or equal to all of the instances of the different types of execution units.

Constraint reference	Lower bound on M
4.6	$M \geq p$
4.7	$M \geq p$
4.8	$M \geq \frac{p}{2}$
4.9	$M \geq \frac{p}{2}$
5.8	$M \geq 2q$
5.9	$M \geq 2q$

Table 5.1: *The required lower bound for M for each constraint.*

UNIVERSITY OF JOHANNESBURG

From Table 5.1, which summarises the lower bounds for M, we can conclude that

$$M \geq \max\{p, 2q\} \tag{5.16}$$

We have established how to determine q however we must still determine some method for finding p. When introduced p was defined as being greater than or equal to x_i however if $p \geq x_n$ then $p \geq x_i$ since x_n is the ending time of the ending dummy instruction. The makespan of a schedule will always be equal to x_n because it is the last instruction to start execution. Therefore the value of p will be the makespan of the worst case schedule.

The worst case schedule for any problem is where there is no parallelism at all. In such a schedule the makespan will be equal to the sum of all the instruction durations since each

instruction will execute individually. The equation below (5.17) introduces a value for p.

$$p = \sum_{i=0}^{n-1} t_i \tag{5.17}$$

In practice any value for M that is greater than or equal to $\max\{p, 2q\}$ will be suitable, however from here on it is assumed that $M = \max\{p, 2q\}$.

5.4 Dealing with differentiated execution units

In Section 4.6 it was assumed that all of the execution units are identical. Each execution unit was assumed to support the same instructions and the duration of an instruction is the same on each execution unit. This view of execution units does not accurately represent real processors which have different types of execution units. For example a typical processor may have one execution unit implementing arithmetic instructions and another execution unit implementing memory access instructions. The number of identical execution units may also differ, for example a processor might have two arithmetic execution units and a single memory access execution unit. Table 5.2 illustrates the execution unit information for a typical processor that will be used in the following discussion.

UNIVERSITY
JOHANNESBURG

Group	**Supported instructions**	**Execution units**
Arithmetic	addition (*add*) comparison (*cmp*) multiplication (*mul*)	ALU 0 ALU 1
Memory access	load	MMU

Table 5.2: *An example of a processor with three execution units and two different types of execution unit.*

The constraints described in the previous chapter cannot deal with identical instructions of differing durations. Processors that fall into that category cannot be optimally targeted with the techniques described. In some cases, like Intel's Itanium, the purpose of instructions with varying durations is that the longer running versions of the instruction consume less power. Yang et al. [32] discuss this feature and present a quantitative analysis of it.

A possible adaptation for our model is to first optimise with only the shorter duration instructions. Then convert the shorter duration instructions into a longer duration instructions provided the conversion does not alter the makespan of the schedule. This

would only be possible if there is sufficient time between the instruction ending and its dependents starting. This adaptation has not been implemented since it is not related to performance.

Two aspects of the model given in the previous chapter changes with the introduction of different types of execution units. Each e_i variable represents the execution unit that instruction i will execute on where $e_i < m$ and m is the number of execution units. When multiple types of execution units are present then e_i should be defined as being less than or equal to the number of execution units that instruction i can execute on. The other aspect that is affected is the composition of the resource contention sets.

Suppose instruction i executes on arithmetic execution units and instruction j executes on memory access execution units. Instruction i and instruction j will never compete for execution units because they use different types of execution units. Since they will never compete with one another for execution units we can conclude that instruction i should not be in instruction j's resource contention set R_j. This is the basis for the requirement (in Section 4.6) that both instruction i and instruction j must use the same type of execution unit in order for instruction i to be present in resource contention set R_j.

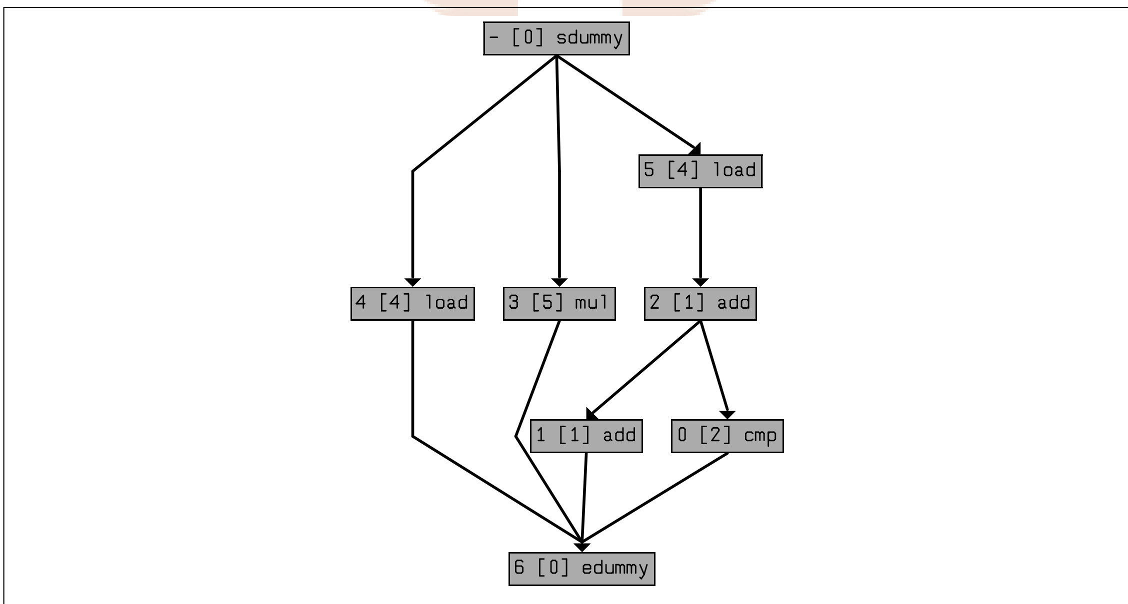


Figure 5.1: *Data dependency graph from Figure 4.3 with the addition of two load instructions, instruction 4 and instruction 5.*

Figure 5.1 is almost identical to Figure 4.6 except that Figure 5.1 has two additional instructions, instruction 4 and instruction 5. Both of the new instructions use a different type of execution unit to the other instructions in the graph. The instruction/execution unit information in Table 5.2 is applied to Figure 5.1 to produce the resource contention

sets for the problem shown in Table 5.3.

Instruction identifier	Contention set contents
0	0, 1, 3
1	0, 1, 3
2	2, 3
3	0, 1, 2, 3
4	4, 5
5	4, 5

Table 5.3: *The resource contention sets for the program given in Figure 5.1.*

The resource contention sets can be processed using the techniques described in Section 4.6.1 to determine what resource limiting constraints are required. The constraints themselves do not change with the introduction of multiple types of execution units.

5.5 Adding bounds to reduce computational time

UNIVERSITY OF JOHANNESBURG

Adding upper and lower bounds to instruction ending times x_i helps to reduce the amount of time required to solve a MILP. The addition of bounds can reduce the number of solutions that the solver has to evaluate and therefore yields a performance benefit. The solver can discard any feasible solution where a bounded variable is outside its bounds. Because MILP's are computationally difficult to solve, adding bounds helps to alleviate this problem.

A lower bound l_i for each instruction ending time x_i is computed based on the ending times of the instructions on which instruction i depends. Algorithm 5.1 is a recursive algorithm that computes the lower bound l_i for instruction i's ending time x_i. The algorithm starts with the ending instruction which is provided by the caller.

Input: Instruction durations t_i and instruction dependencies. All instruction ending time lower bounds l_i are initialised to -1 to indicate that they are unresolved.
Output: The lower bound instruction ending times l_i.

Lowerbound(i) {
  $p = 0$
  foreach instruction j on which i depends
  {
    if $l_j == -1$
      Lowerbound(j)
    if $l_j > p$
      $p = l_j$
  }
  $l_i = p + t_i$
}

Algorithm 5.1.

To compute the upper bound u_i for each instruction ending time x_i the approach taken is similar to computing the lower bound. The upper bound u_n of the ending instruction's ending time x_n is simply the makespan of the best known solution. This is possible because x_n will always be equal to the makespan of the schedule described by the x_i variables. For each instruction i where $i \neq n$ the upper bound u_i is computed based on the upper bound and duration of each instruction dependent on instruction i.

Before the upper bounds can be computed it is necessary to find the makespan of a feasible solution. In Section 5.3 an equation (Equation 5.17 is repeated below) for finding a feasible solution makespan was introduced.

$$p = \sum_{i=0}^{n-1} t_i \tag{5.18}$$

Algorithm 5.2 is a recursive algorithm that computes the upper bound u_i for instruction i's ending time x_i. The algorithm traverses the graph from the ending instruction which is provided by the caller as well as the makespan of a known feasible solution.

Input: Instruction durations t_i and instruction dependencies. All instruction ending time upper bounds u_i are initialised to -1 to indicate that they are unresolved. Let c be the makespan of a known solution.
Output: The instruction ending time upper bounds u_i.

```
Upperbound(i, c) {
  if u_i > c || u_i == -1
     u_i = c
  foreach instruction j which is dependent on i
  {
     if u_j > c || u_j == -1
        Upperbound(j, c - t_i)
  }
  u_i = p + t_i
}
```

Algorithm 5.2.

Although upper bounds u_i generated using p are valid it is possible to generate better upper bounds. When p was first introduced it was defined as the makespan of a schedule where there is no parallelism. Often heuristic solutions will have a makespan less than p and will therefore produce better upper bounds.

UNIVERSITY OF JOHANNESBURG

5.6 Generating heuristic solutions

In the previous section we saw that the effectiveness of an upper bound on x_i was determined by the makespan of the known solution. The closer the known solution is to being optimal the better the upper bound on x_i will be. Therefore we should use the best possible solution that can be computed in a reasonable amount of time. The known solution used in the previous section was a solution where there is no parallelism. However, even a relatively simple heuristic algorithm will generally produce a better solution that has some parallelism.

Two basic principles are employed in the heuristic algorithm. The ending times of the instructions are minimised subject to instruction dependencies. For each iteration of the algorithm minimisation takes place, the schedule is then checked for resource overuse. If resources are over-used then a dependency is added between two instructions involved in

the overuse incident. The addition of a dependency reduces the available parallelism and therefore reduces the number of execution units necessary to execute the instructions in the schedule.

The heuristic algorithm, minimum slack dependency addition (MSDA), is similar to the branch and bound technique introduced in Chapter 4 assuming a depth first approach was taken. However, the branch and bound technique guaranteed optimality while the heuristic algorithm does not. This is because the search terminates after a feasible (not optimal) solution is found. Unlike the branch and bound approach, the selection of two instructions for dependency addition is very important. The added dependency must be adhered to in the final solution and will therefore affect the makespan of the solution.

The steps have been enumerated below along with references to the necessary algorithms for each step.

1. Minimise instruction ending times using Algorithm 4.6.
2. Check for an execution unit overuse incident using Algorithm 4.7. If there is no execution unit overuse then the solution is a feasible solution and processing stops.
3. Algorithm 4.7 returns a set containing the instructions involved in the execution overuse incident. A pair of instructions are extracted from this set and a dependency is introduced between the two instructions. A discussion on which pair of instructions to extract follows, however the algorithm will produce a feasible solution regardless of which instructions are chosen.

A score is determined for each pair of instructions (i,j) in the overuse set. The pair is evaluated as if instruction j were dependent on instruction i. The score is the maximum number of clock cycles of delay incurred by the instruction i instruction j dependency on instructions dependent on instruction j. We define the delay for an instruction f, that is dependent on j, as the difference between (i) the starting time of instruction f before the dependency is introduced and (ii) the ending time of instruction j after the dependency is introduced. We have assumed that j is dependent on i. The case where i is dependent on j is similar.

This value can be positive or non-positive. A positive value indicates that the instructions dependent on instruction j will be delayed. A non-positive value indicates that the instructions dependent on instruction j will not be delayed. A positive score would mean

that the created path is a critical path. Similarly a non-positive score would mean that it is a non-critical path.

Instruction delay can therefore be minimised by finding the instruction pair (i,j) with the lowest score. Remember that any instruction pair where the score is less than or equal to zero will not result in a delay. Instead of finding the instruction pair with the lowest score, a more complex evaluation criteria is used. If there are no instruction pairs with a score of zero or less then the pair with the lowest score is selected. If there is at least one instruction pair with a score of zero or less then only instruction pairs with a score of zero or less are evaluated. When evaluating instruction pairs with a score of zero or less, the pair with the greatest score is selected, the score will still be less than or equal to zero.

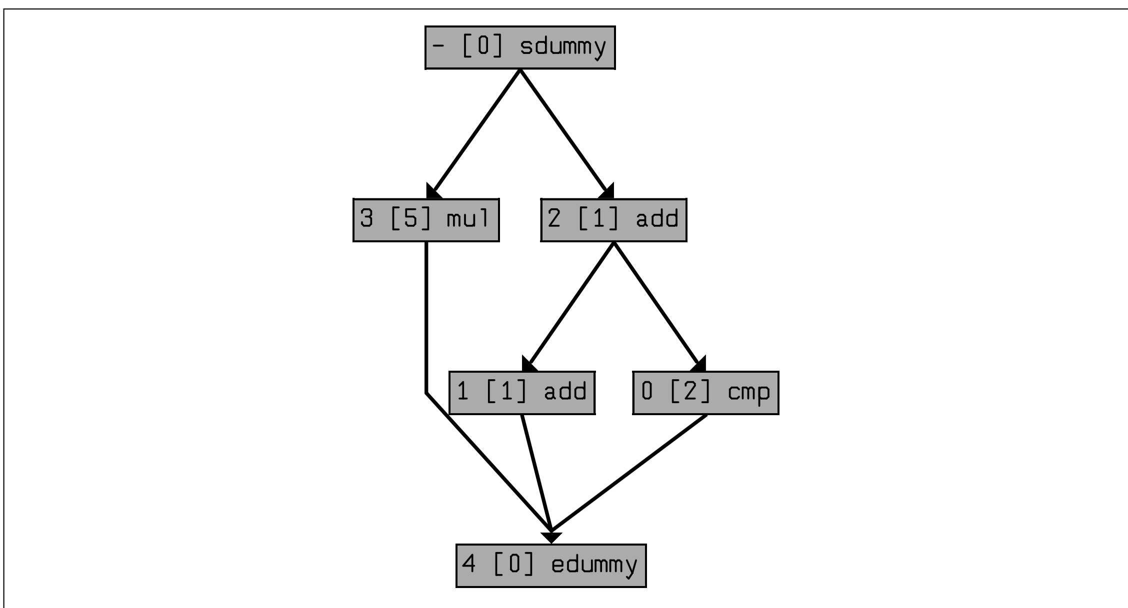


Figure 5.2: *Data dependency graph from Figure 4.3.*

The selection of instruction pairs with the least negative score results in better schedules in most cases. The improved schedules result from reduced fragmentation in terms of resource availability and instruction dependency.

	0	**1**	**2**	**3**	**4**
Ending time	3	2	1	5	5

Table 5.4: *Instruction ending times for Figure 5.2. The first row contains instruction labels that correspond to the graph.*

The heuristic algorithm is applied to the problem originally introduced in Figure 4.3 (repeated as Figure 5.2). Table 5.4 shows the instruction ending times after they have been minimised in the first step of the heuristic algorithm. We know from previous analysis

of this problem that instruction 0, instruction 1 and instruction 3 contribute to resource overuse (see Chapter 4 for details) since there are only two execution units. The resource overuse set therefore contains these three instructions.

instruction j	**0**	**1**	**2**	**3**	**4**
0		-1		2	
1	-1			1	
2					
3	3	2			
4					

Table 5.5: *Table of instruction pair scores where the j component is enumerated vertically and the i component is enumerated horizontally.*

Table 5.5 shows the scores for all combinations of the three instructions. All of the numbered instructions are presented but values are only supplied for combinations of overuse instructions. From the table we can conclude that we should either choose instruction pair $0, 1$ or instruction pair $1, 0$. By inspecting the problem shown in Figure 5.2 we can see that this is indeed the correct pair of instructions. The optimal makespan is equal to the duration of instruction 3 therefore introducing a dependency involving instruction 3 would increase the makespan.

	0	**1**	**2**	**3**	**4**
Ending time	3	4	1	5	5

Table 5.6: *Instruction ending times for Figure 5.2 where instruction 1 is dependent on instruction 0. The first row contains instruction labels that correspond to the graph.*

A dependency stating that instruction 1 is dependent on instruction 0 is included based on the discussion in the previous paragraph. Then the instruction ending times were minimised again yielding the results shown in Figure 5.6. Resource overuse is not present in this schedule therefore it is a solution. In this example the heuristic solution is an optimal solution, however this is not always the case.

UNIVERSITY OF JOHANNESBURG

Chapter 6

Parallelism in array boundary assertions

Programs written using programming languages are sometimes susceptible to a group of security exploits collectively known as buffer overflow attacks. These exploits take advantage of the fact that the program generated by the compiler does not check the validity of load and store addresses. Memory access beyond array bounds can be exploited to either change parts of the application being executed, or alternatively, read secret information from other sections of the application's memory. Buffer overflow attacks take advantage of functions that do not specifically check the lengths of data blocks read or written. The behaviour of such functions (commonly string related) is clearly incorrect, however, due to the complexity of modern programs they sometimes go unnoticed. Pfleeger discusses buffer overflow attacks in [26].

6.1 Array boundaries and exploitation

Figure 6.1 shows three arrays in the memory of the computer. When an application writes data to array A it uses the memory address of array A and an offset depending on which element of the array is to be accessed. If the program writes to the third element of array A and each element is 1 byte then the actual address would be 2. The actual address for the second element of array B is 9 since the address of the array is 8 and the second element is offset by 1 from the beginning of the array.

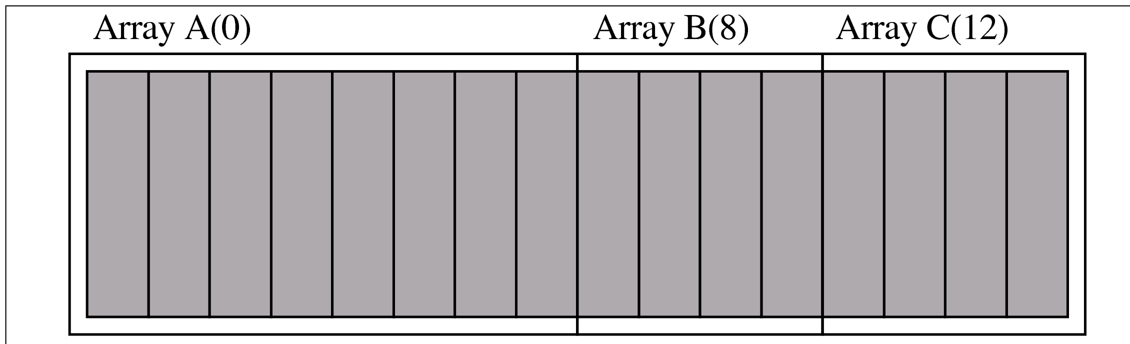


Figure 6.1: *The values in parenthesis are the memory addresses of the respective arrays.*

Using this scheme to access memory has one inherent problem. What if the index into the array is greater than or equal to the size of the array? If this were the case then instead of accessing an element in the intended array, data from the memory of the following array would be accessed instead. Suppose, for example, the intention is to read an element from array A with an index of 9. Array A only has 8 elements so the element accessed will not be in array A. The address of array A is 0 but the index is 9 therefore the final address is 9. This is actually an element of array B since the area of memory for array B starts at 8. Instead of accessing an element at index 9 (which is impossible) of array A the program would instead access the second element of array B.

The division of memory into particular arrays is a high level concept of which the processor has no concept. The compiler (for static memory allocations) or the operating system (for dynamic memory allocations) partitions the memory into the logical array structures. Static allocation is used when the size of the array is known at compile time. The opposite of static allocation is dynamic allocation. It is used when the size of the array is only known at runtime.

The problem is that nothing on the processor level differentiates a load or store to the different logical structures. The logical structures are high level constructs so the processor is unaware of how the compiler or operating system has partitioned the memory available to it.

Checks that a program is operating according to plan are called assertions, therefore a check that ensures that loads and stores are only to valid locations would be a special kind of assertion. Insertion of memory address assertions is a job for the compiler given that the assertions are program code themselves. Memory address assertions are not a new concept, however there are two principal barriers to their widespread deployment.

Generating code at compile time to check the validity of access to memory that was al-

located statically is fairly simple, however the same is not true for dynamic allocations. In the case of dynamic allocations the compiler must generate code that wraps around the allocation request to the operating system. This is necessary because the chunk of code that tests memory accesses must be aware of the allocated memory as well as how large it is. The wrapping code records the address and size of dynamic memory allocations as well as keeping track of references to it.

In some languages (especially C/C++) it can be very difficult for the compiler to associate a reference to memory with the original memory allocation. This problem is called *aliasing* and is the first barrier to widespread implementation of boundary assertions. Steps to alleviate this problem are being proposed in the form of language extensions that would require the programmer to be more specific when creating aliases of arrays.

The second barrier to deployment is that the inclusion of boundary assertions has a significant negative impact on performance. The aim of this chapter is to reduce the overhead incurred by boundary assertions. This is achieved by introducing parallelism into boundary assertions.

When examining the dangers of extra-boundary access an important aspect concerning application structure needs to be taken into account. In modern computers, application code is separated from application data. Usually two separate segments are maintained for data and code. The details of this concept were discussed in Chapter 2.

UNIVERSITY
JOHANNESBURG

If an application attempts to access data outside of its data segment a page fault occurs and the application's execution is terminated. This behaviour is justified because the program has clearly malfunctioned and therefore the state of the program is no-longer reliable.

In certain circumstances ranges of the data segment may also have some restrictions. The most notable of these cases is that of shared memory. This is where one application can share parts of its data segment with another application even though the two applications have completely separate memory spaces. The application that allocated the shared memory can impose restrictions on what type of memory accesses the other applications are allowed to perform on its memory. Access restrictions on shared memory is implemented with page rights as are the restrictions on other segments like the code segment.

6.2 Dangers of unauthorised loads and stores

The primary danger in loading from unauthorised areas of memory is that the user attempting to exploit the weakness will be accessing restricted data that they would not normally have access to. Depending on the exact details of the weakness exploited they may have access to the entire data segment of the application. The data segment maps all of the memory for data that the application has available to it.

An example where this kind of access would be a problem is in the case of a web server where the attacker may download a copy of the file-cache that it maintains. In this way the attacker may gain access to restricted files that legitimate users have downloaded, perhaps from some sort of secure zone.

Clear text passwords are especially vulnerable in this respect because they will generally be available in an easy to read format in the application's memory. The user may gain higher levels of access by gleaning information in this way.

Illicit stores pose more of a danger than illicit loads in that the attacker can potentially compromise the entire application. In the case of an illicit load the attacker can syphon off information, by abusing stores the attacker can take control of the application.

If a computer allows stores to the memory area containing the application code then the attacker can insert their own code overwriting the original application. It is for this reason that the code segment is restricted to read-only status. By eliminating the ability to change the program's code the only other benefit an attacker can gain from illicit writes is the ability to change the applications data.

The application data includes runtime variables, data structures, temporary storage and the application's stack. The application's stack is the place where *activation records* are stored [1]. Each time a function is called, information regarding the caller is stored on the stack. When a function returns, the information in the most recent activation record is retrieved. The information retrieved is necessary to continue execution of the caller. Figure 6.2 shows a section of the stack containing the activation records where a function, A, calls another function B. Function B calls C which has two local variables a and b. Function A also has two local variables i and j.

The most important piece of information in an activation record is the address of the call instruction that initiated the transfer of control to the called function. The address of the

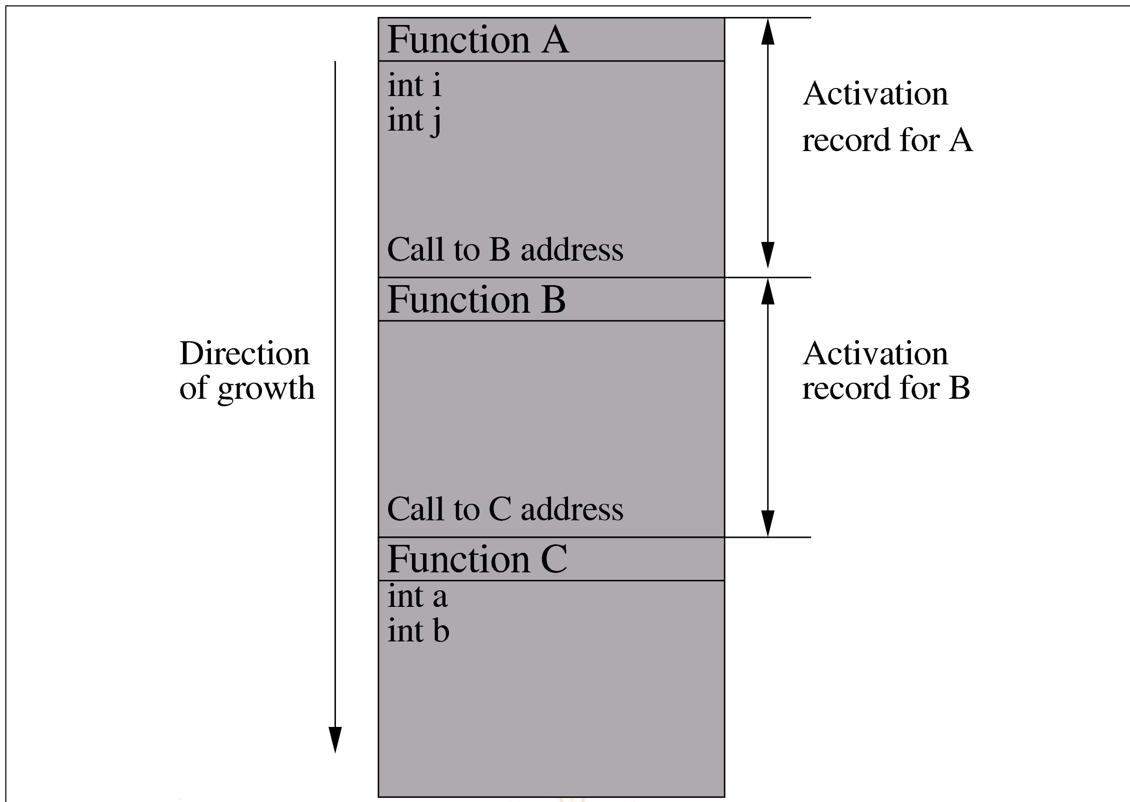


Figure 6.2: *The function currently executing is function C. The address of the instruction in function B that initiated the call to function C has been recorded in the activation record for instruction C. A similar address exists in the activation record of function B for the call to function B. Each activation record also contains local variables and function parameters.*

UNIVERSITY OF JOHANNESBURG

call instruction is an address in the code segment which is restricted to read or execute and could not be altered by an illicit store. By altering the address specified in the activation record any instructions in the code segment can be executed since the application can both load and store from the stack. Once the return address has been altered the function will not return to the caller as was intended but instead control will transfer to the program segment specified by the user.

6.3 Evaluating load/store addresses

Two tasks must be performed when accessing data in an array. First the final address of the data must be computed. Once the address is available the actual load or store can be performed. On most processors the parts of the calculation of the final address can be combined with the load or store instruction. For example most processors load

instructions allow the address of the load to be specified in two parts, a base and an offset. The load instruction will add the two together to produce the final address. A processor may support load and store instructions with a variety of different *addressing modes*. An addressing mode is a method for describing a memory address.

Checking if the relative component of an address (let's call it a) is within bounds is fairly simple if the array was statically allocated. In principal assertions must be inserted that ensure that a is both greater than or equal to 0 as well as being less than the size of the array. In practice only instructions to test if the upper bound has been exceeded are necessary since memory addresses are always unsigned and therefore a negative number would be interpreted as a very large number instead. Because the size of the array is known at compile time the instructions to check if the load or store is within bounds can be hard-coded with the size of the array. Checking if access to a dynamically allocated array is within bounds is a fair bit harder than for a statically allocated array.

When checking bounds on a static array the size of the array is hard-coded into the code that performs the check. Since the size of a dynamic array is only determined at runtime, information regarding the array must be kept in a separate global table within the data segment. The current size of the array must be fetched from this table each time a boundary check is performed. The boundary check itself is identical to that of the statically allocated array.

UNIVERSITY OF JOHANNESBURG

An additional problem related to dynamic array allocation is the use of pointers or array aliases. An array alias is a secondary reference to an array. The array in question would be defined like any other array except that an additional reference is made to the physical array. The compiler cannot tell which array an alias points to at compile time introducing difficulties when retrieving the array information.

6.4 Boundary assertions on the instruction level

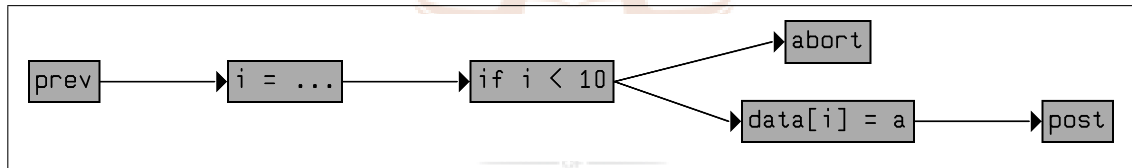
For each memory access to an array the computer has to check the validity of the access before the access itself takes place. The compiler introduces bounds checks where applicable. Table 6.1 shows both the original code as well as an example of what a compiler might do. The compiler has inserted a boundary check for the write operation to the array data. Only once the index has been verified does the actual store take place. If the index verification fails then the program would terminate.

Programmers view	**computational view**

int data[10]
...
 	
// i is a variable that has
// been initialised by the client
i = ...
 	
data[i] = a;
...
	int data[10]
	...
	
	// i is a variable that has
	// been initialised by the client
	i = ...
	
	if (i < 10) {
	   data[i] = a;
	else
	   segmentation_fault();
	...

Table 6.1: *Shows the original program code on the left and the same program on the right with the addition of bounds checking code.*

In order to calculate the time requirements as well as the parallelism the program needs to be converted into a data dependency graph (the process to do this is described in Chapter 3). The data dependency graphs introduced in this section are a simplification of the actual instructions that may need to be performed. However, the graphs do illustrate the parallelism and structural implications of boundary checking.



JOHANNESBURG

Figure 6.3: *The directed graph of the operations required for a boundary check using traditional methods. The nodes represent operations and the arcs represent dependencies. "prev" refers to the operations that are performed before i is used as an index into the $data$ array. Similarly "post" represents the operations following the code sample.*

In Figure 6.3 the computational time is determined by the time taken for the longest path to complete. This path is commonly referred to as the critical path and defines the absolute minimum makespan of the basic block. Critical paths were discussed in Chapter 5.

The "abort" node represents the result of a boundary overrun and would result in the application terminating. In such circumstances additional information could be produced.

6.5 Parallelism in boundary evaluations

It is important to realise that the result of a boundary overrun is the termination of the program in question. This is an absolute requirement due to the fact that the program is in an inconsistent state. The behaviour of a program after a boundary failure is undefined and therefore the program in question cannot continue execution.

It is possible to shorten the dependency chain illustrated in Figure 6.3 by executing the boundary check during or after the array access. Figure 6.4 illustrates the same program segment however now the boundary check is performed in parallel with the array access. The end result is that the critical path through the program segment has been shortened and will, in principle, require less time to execute.

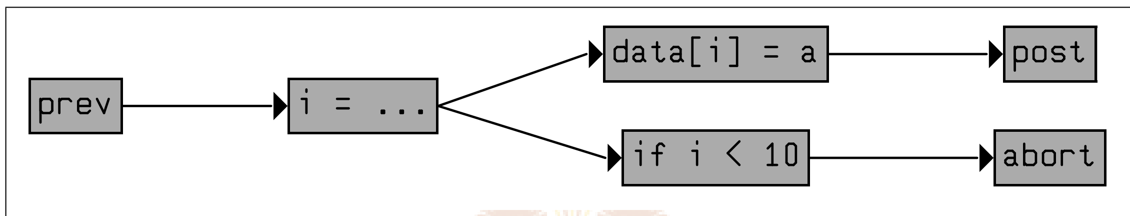


Figure 6.4: *This directed graph is solving the same problem as 6.3 but performs the boundary check in parallel with the access of the array "data".*

Any instructions performed after the array access and before the boundary check will be 'tainted'. That is to say the instructions or data may have been affected by the array access. The exact status of these instructions will only be clarified after the boundary check has been performed.

Limits have to be set on when the boundary checks are performed in order to prevent the tainted instructions from affecting the operating environment. Because the application is flushed from the system in the event of a boundary overrun the tainted operations and their data will also be flushed. It is therefore important to perform the boundary checks before any instructions that may affect the wider environment are executed.

In the section on the effects of invalid loads, the example of a plain text password being read illicitly was mentioned. Such an action involved two parts, firstly reading the password, and secondly returning that information to the user. It is therefore important to schedule the boundary check before data is returned to the user.

In general it is advisable to perform the boundary check before the termination of the basic block. All operations in the basic block are guaranteed to execute and the end of the

basic block marks a point of uncertainty. In practice this requirement can be modelled as an instruction dependency when scheduling the instructions.

This approach works well with static arrays, however, with dynamic arrays several problems crop up. The information regarding the array is stored in a table in the data segment. An illicit store could, in theory, write to this table altering the contained data. Since the boundary check is dependent on the information contained in the table, having tainted information in the table would affect the legitimacy of the boundary check. The solution to this problem is to ensure, through the use of dependencies, that the necessary information is fetched from the table before the store occurs. The actual check could still be performed after or during the store as long as the data from the table has been loaded into a register. The contents of a register cannot be altered by the store instruction since it can only write to the data segment, therefore the integrity of the information is guaranteed.

6.6 Remaining drawbacks

In practice several issues may still result in boundary checked code being slower than non-checked code. A parallel solution is still dependent on sufficient resources being available in order to prevent delays in the critical path. If insufficient resources are available then the critical path will be lengthened to compensate.

UNIVERSITY
JOHANNESBURG

Several other aspects may also affect computational overhead:

Instruction cache	The instructions required for checking boundary overruns still consume cache space. In cache constrained applications this may affect performance.
Instruction issuing capacity	Computer processors can issue a maximum number of instructions simultaneously. In most modern processors the ability to issue instructions is greater than the processors ability to execute them and is therefore rarely a problem.

Boundary information maintenance — As discussed previously, arrays that have been dynamically allocated have special requirements and introduce additional overhead. Information has to be maintained for each dynamic array and possibly each alias as well. The information also has to be loaded from the table before a store can take place and therefore reduces the parallelism of the boundary evaluation.

A major factor in the way of adoption of the introduced technique for current processors is that the order of instructions cannot be guaranteed on out-of-order processors. The processor could potentially allow tainted loads or stores because the dependencies introduced in this chapter do not involve data.

6.7 Boundary check performance

Two different programs were evaluated using three different configurations in terms of boundary checking. The three configurations are as follows: The first configuration applies no boundary checks. The second configuration applies boundary checks that are performed in the traditional (non-parallel) way and the third configuration applies parallel boundary checks using the technique introduced in this chapter. In each case the array was statically allocated due to limitations of the compiler implementation. The compiler implementation did not have the necessary infrastructure to perform checks on dynamically allocated arrays.

The first program copies the contents of one static array into another static array. The copying process continues until a null value is encountered in the source array, in this circumstance a null value is equal to 0. This program mimics a string copy function like *strcpy* in C/C++. The second program also copies the content of one static array into another static array, however the amount of data copied is fixed because it is specified as a parameter. This function is somewhat similar to the *memcpy* function in C/C++. In both programs the makespan of the schedule for the body of the for-loop was recorded. An optimal schedule was determined using the techniques introduced in Chapter 4 and Chapter 5. The results are shown in Table 6.2. The source code for both test programs is available in Appendix B.

The parallel technique improved the optimal solution to the makespan of the basic blocks where they were employed however the gains achieved vary. In both cases the number

UNIVERSITY JOHANNESBURG

Code sample	boundary assertion	number of instructions	makespan
Null terminated copy	No assertions	17	10
	Traditional assertions	22	12
	Parallel assertions	22	10
Fixed size copy	No assertions	18	7
	Traditional assertions	24	10
	Parallel assertions	24	9

Table 6.2: *Two sets of results are presented in this table. The number of instructions per program given in column 3 include the dummy instructions. Traditional assertions are assertions that do not employ any parallelism. Parallel assertions are boundary assertions that employ the techniques introduced in this chapter.*

of instructions were the same for both programs because the instructions required were identical. The only difference was which dependencies were required. In practice the number of instructions required to implement the parallel technique may be different to that of the traditional technique.

In order to exploit the parallelism introduced through the parallel boundary check a good scheduling algorithm is required since this technique is only applicable to in-order processors. The compiler used to perform the tests did not attempt to increase the parallelism of the test programs. It is unknown if such techniques will result in lesser or greater gains.

UNIVERSITY
OF
JOHANNESBURG

Chapter 7

Implementing an optimising compiler

The aim of the implemented compiler was to demonstrate instruction scheduling in a realistic environment. The compiler also had to support boundary assertion generation that is capable of generating the parallel boundary assertions introduced in Chapter 6.

The compiler was implemented in the C programming language with the aid of compiler construction tools. The compiler implements a subset of the C (ANSI-C standard) programming language.

UNIVERSITY OF JOHANNESBURG

The implemented compiler consists of multiple modules shown in Figure 7.1. The modules are divided up into two primary groups. The first group of modules is for the compiler core while the second group is for the compiler backend.

The compiler core consists of the parser, the instruction generator and the program simplifier. The instruction generator is the interface to the CPU targeter and instruction scheduler. The third part is the simplifier which modifies the program in order to simplify it or reduce the computational requirements through the use of traditional compilation techniques. The backend modules include register assignment, instruction scheduling and assembly language program generation.

7.1 Data structures and representation

Separate data structures are used to represent the syntax tree and the DDG because of their differing requirements. The use of two different data structures has an additional

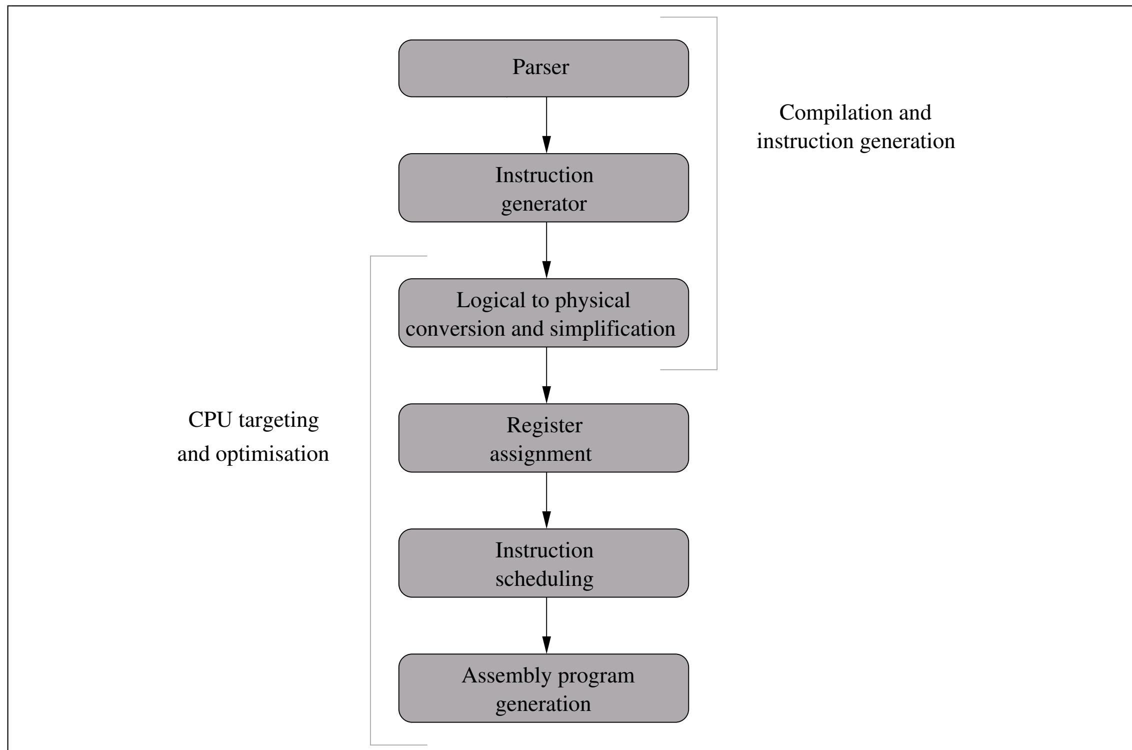


Figure 7.1: *A high-level data flow diagram of the components of the compiler (includes program optimiser).*

benefit in that the compiler core and optimiser are separate modules. Sharing code for data structures would introduce additional complexity due to intermodule dependencies.

Unlike the DDG, the syntax tree only needs to be linked in a single direction since it is only traversed using a top down approach. Therefore the data structure used is a very simple tree which uses an array for the actual data storage. The array can grow dynamically using the *realloc* memory allocation call, which, through the use of the memory paging system, allows for fast enlargement. Each node is generated using a node type identifier as well as a list of node references that will be child nodes of the new node. Therefore the tree is constructed using a bottom up approach.

The program representation structure of the optimiser is that of a general graph. Each node has a list of parent-to-child links as well as child-to-parent links. The lists of links must be able to be altered dynamically and is therefore implemented using linked-lists. The cost of using linked-lists for this purpose is negligible given the small number of elements linked in this way. The nodes and the links are stored in separate arrays which are identical in implementation to the arrays used in the compiler core. Problem specific information is encoded into the node as well as the parent-to-child links. Each child-to-parent link references the equivalent parent-to-child link.

Each node in the DDG is an instruction and therefore requires information regarding the instruction. Each instruction has an identifier which identifies the type of instruction. Identical instructions which operate on different types of data have different identifiers therefore no separate data type information is required.

Each instruction also has a reference to the syntax tree node that it was generated from. This information is used when reporting error messages in order to report the correct line and source file where the erroneous code can be found.

Each instruction also has a reference to an execution unit as well as an ending time. After scheduling, both fields will be filled with the relevant information. Before scheduling the contents of the fields are undefined. The reference to the execution unit is an index into the execution unit array which is created when the CPU description is processed. Each instruction also has a variable to store a reference to the mathematical program variable that represents the ending time of the instruction.

Some types of instructions need additional information. For example, a call instruction needs a reference to the symbolic name of the function that will be called. The requirements of the different instructions in terms of additional information vary widely, therefore the mechanism must be fairly flexible. Each instruction is allowed to have multiple attributes which are stored separately from the instruction's data in a linked-list. The instruction has a reference to the chain of attributes for the instruction in question. In practice very few instructions have attributes, however some instructions may have multiple attributes. The attribute system is used for both the DDG and the syntax tree however the attributes of the two data structures are not comparable.

The resource contention sets (the abbreviation *RCS* is used in the implementation) are implemented as simple arrays which are internally allocated from a single block of memory. Internal allocation is performed to prevent fragmentation, besides being very simple to implement. The underlying data structure used for allocation is the enlargeable array structure used elsewhere in the implementation. Resource contention sets cannot be enlarged once created which is not a problem since there is no reason to do so.

7.2 Describing the CPU

Converting an intermediate form program to a program for the target processor requires information on the targeted CPU. The information about the CPU must include general information as well as specific information on individual instructions. General information includes things like the number of bits required for memory addresses and alignment requirements of the CPU. A single format cannot describe the multitude of different CPUs. However, the format decided upon does a fairly good job of the simple instruction sets of VLIW processors. An example CPU description file is given in Appendix C.

The CPU description file is parsed using similar techniques to those used to parse the source code except that it is far simpler. The information contained is converted to an in-memory representation to simplify further use.

The CPU description format contains five different sections which are named *description*, *registers*, *logical*, *physical* and *function*. Each section is defined using its name followed by a pair of braces which enclose the information.

Comments begin with a double backslash and continue to the end of the line. Comments are the only language elements which are sensitive to end-of-lines. String literals are enclosed in quotes which may be escaped (to allow the string to contain a quote character) using the backslash character. Numeric values may be placed in the description without any additional characters. All statements not ending in a closing brace must be followed by a semi-colon. All keywords and identifiers are case sensitive. Individual terminal characters are placed in single inverted commas. In BNF(Backus Naur Form), terminals are terms that must be matched unlike non-terminals which are rule references within the grammar. Terminals are printed in bold font and non-terminals are printed in regular font.

7.2.1 Description section

Each element within the description section consists of a property followed by a value for the specified property. Each statement is terminated with a semi-colon.

The BNF grammar for the description section is as follows.

description_section	→	**description** '{'	aspect_definitions '}'
aspect_definitions	→	**name**	string_literal ';'
		model	string_literal ';'
		addresses	number ';'
		strict_alignment	number ';'
		alignbytes	number ';'
		baseptr_top	number ';'
		baseptr_bottom	number ';'

Table 7.1: *Grammar for the description section.*

Information regarding the various fields in the description section of the CPU description file follows.

name	Identification of the CPU family.
model	Identification of the exact model within the CPU family.
addresses	The number of bits needed to represent a memory address.
strict_alignment	A zero value means that the CPU does not require memory addresses(for data) to be aligned. For a non-zero value the inverse is true.
alignbytes	The advisable alignment of memory addresses, in bytes, whether or not strict alignment is required. Even if a CPU does not have a strict alignment requirement, it may still be beneficial to align data.
baseptr_top	This is the amount of space, in bytes, that separates the parameters of a caller from the address pointed to by the base pointer.
baseptr_bottom	This is the amount of space, in bytes, that separates the local variables of a function from the address pointed to by the base pointer.

UNIVERSITY
JOHANNESBURG

7.2.2 Registers section

Ranges of registers are defined in this section. Additional restrictions can then be placed on register combinations. For example some instruction sets will share storage for both integer and floating point registers and therefore they cannot both be used at the same time.

register_section	→	**registers** '{' register_statements '}'
register_statements	→	new_statement | mutex_statement | alias_statement
new_range	→	**identifier**
existing_range	→	**identifier**
new_statement	→	**new** new_range **number** **';'**
mutex_statement	→	**mutex** existing_range existing_range **';'**
alias_statement	→	**alias** new_range register_expressions **';'**
register_expressions	→	register_expressions **','** register_expression | register_expression
register_expression	→	existing_range | existing_range **'['** **number** **']'** **';'**

Table 7.2: *Grammar for the register section.*

Three different types of statements can be constructed regarding registers.

new — Creates a new range of registers which can be referenced using the specified identifier. The number of registers in the range is specified after the identifier.

alias — Associates a new identifier with an existing register range. A subset of a range can also be specified instead of an entire range.

mutex — Prevents the registers from two register ranges (or aliases) from being used at the same time. Suppose a register is allocated from one register range which was mutexed with another register range. The register in the second register range with the same index as the allocated register will not be available for the lifetime of the allocated register.

7.2.3 Logical section

The logical section defines logical (or intermediate) instructions. Each logical instruction references one or more physical instructions which are the physical implementation of the logical instruction. The compiler expects a specific instruction set in terms of logical instructions. Support for some logical instructions is optional because the compiler can generate alternative code should the logical instruction not be present.

Each definition contains a single logical instruction which may have numbered parameters as well as one or more physical instructions. The physical instructions also have numbered parameters. The numbering of the parameters is used to connect physical instructions to one another as well as connecting the physical instructions to the logical instruction.

logical_section	→	**logical** '{' logical_instructions '}'
logical_definitions	→	logical_definitions logical_definition
reference	→	**'$' number**
inputs	→	inputs **','** reference | reference | ϵ
outputs	→	outputs **','** reference | reference | ϵ
logical_instruction	→	**identifier** '(' inputs ')' '(' outputs ')'
physical_instruction	→	**identifier** '(' inputs ')' '(' outputs ')'
logical_definition	→	logical_instruction physical_instructions **';'**
physical_instructions	→	physical_instructions physical_instruction

Table 7.3: *Grammar for the logical instruction section.*

For example some digital signal processors support a multiply-add instruction which multiplies two numbers and adds the product to a third number. Suppose a multiply-add instruction is supported in the intermediate format (logical instructions) but is not supported by the target processor (physical instructions). The definition for the logical instruction muladd (multiply-add) would be as follows.

muladd($1, $2, $3)($4)   mul($1, $2)($5)   add($3, $5)($4);

Which is the functional equivalent of $\$4 = \$1 * \$2 + \$3$. The reference for the output of the add instruction is identical to the reference of the output of the muladd instruction. Therefore the final result is the output of the addition.

7.2.4 Physical section

The physical section defines physical or CPU specific instructions. Physical instructions are referenced by logical instructions to allow for conversion from intermediate code to physical code.

Physical instructions can only be defined within an execution unit definition. An execution unit is defined using the *new* keyword. Multiple identical execution units are supported through the use of the *clone* keyword which duplicates an existing execution unit. The *in*, *out*, *latency* and *emit* fields are described below.

physical_section	→	**physical** ‘{‘ execution_units ‘}‘
existing_unit	→	**identifier**
new_unit	→	**identifier**
execution_units	→	execution_units execution_unit | execution_unit
execution_unit	→	**new new_unit** ‘{‘ physical_instructions ‘}‘ |
		clone new_unit existing_unit ‘;‘
instruction_name	→	**identifier**
physical_instructions	→	physical_instructions physical_instruction |
		physical_instruction
physical_instruction	→	instruction_name ‘{‘ description ‘}‘
description	→	in_field out_field latency_field emit_field
in_field	→	**in:** register_list ‘;‘ | ϵ
out_field	→	**out:** register_list ‘;‘ | ϵ
latency_field	→	**latency: number** ‘;‘
emit_field	→	**emit: string_literal** ‘;‘ | ϵ
register_list	→	register_list ‘,‘ register | register
register	→	**identifier** | ‘@‘**number**

Table 7.4: *Grammar for the physical instruction and execution unit section.*

in	The list of inputs that the instruction accepts.
out	The list of outputs that the instruction produces.
latency	The duration of the instruction in clock cycles.
emit	Assembly string to emit when the instruction is generated. Details regarding the formating of the string is given at the end of this chapter in the section on assembly program generation.

7.3 The compiler core

The parser component and the instruction generator components in Figure 7.1 are expanded in Figure 7.2. The first three components (lexical pass, syntax pass and semantic pass) were discussed in Chapter 3.

The lexical analyser was implemented using the Flex tool which generates a lexical analyser from a description file. The syntax analyser calls the lexical analyser each time it wants another token to process. The description file contains a set of regular expressions where each regular expressions has an associated action. Actions can do almost anything because they are written in C. However most of the actions just issue a token for the lexical element and sometimes include some additional information, like a value for a constant.

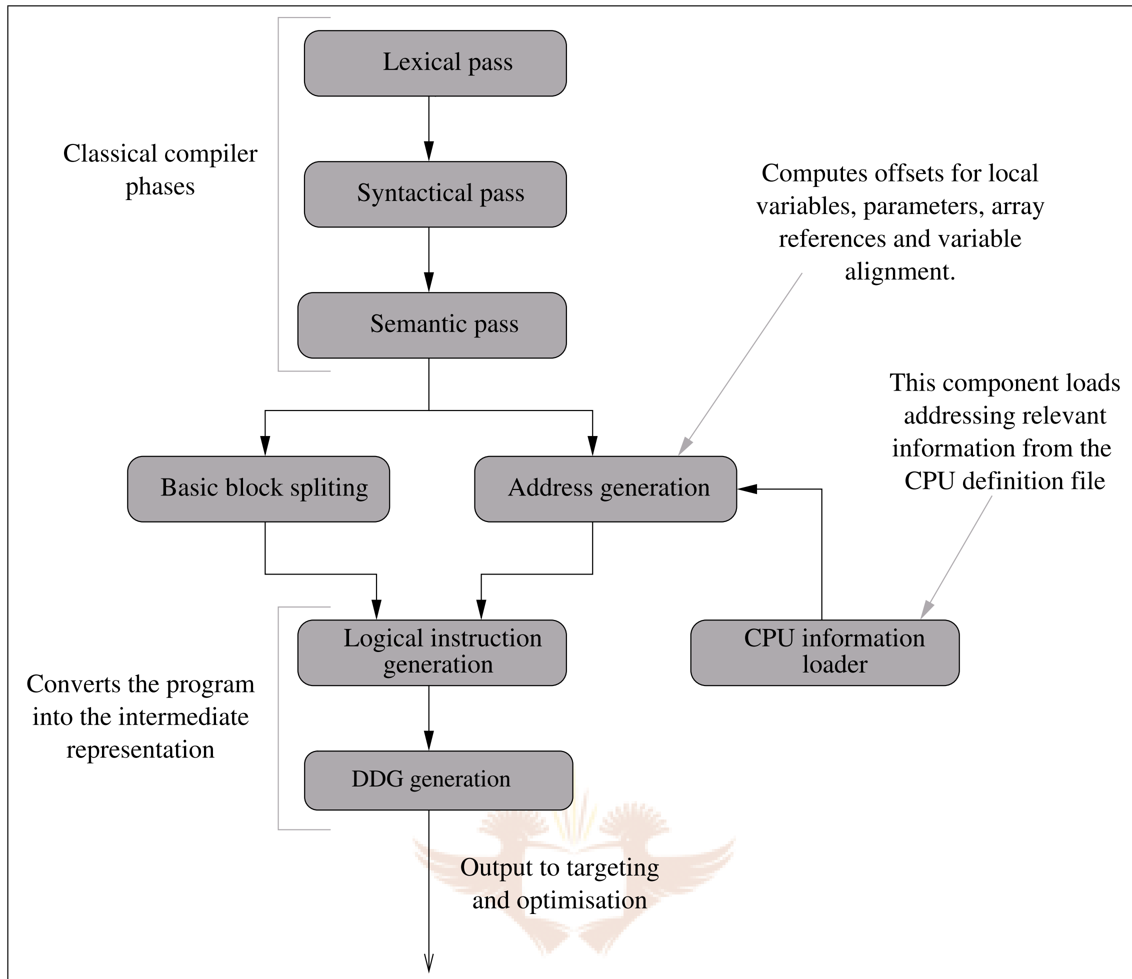


Figure 7.2: *A more detailed data flow diagram (than Figure 7.1), showing the modules of the compiler core.*

This token is then passed to the syntax analyser in the form of a return value. The lexical elements described in the description file were for the C programming language.

The syntax analyser was implemented using Bison [14] which, similar to Flex [15], generates a syntax analyser from a description file. The grammar of the language is described by the description file in BNF. BNF statements can also have associated actions and, like the actions for the Lexical analyser, they are written in C. Unlike the lexical analyser the actions of the syntax analyser do not return anything. Instead they call the tree building routines passing information regarding the type and children of the new node.

Symbols are resolved and complex data types processed during the syntax analysis phase. The resolution of symbols is a normal task of this phase, however the processing of complex data types is not. Representation and processing of data types is fairly complex in C therefore type definitions are processed as they are encountered. Processing of data types

entails validity checking and replacement with a virtual type if it is more complex than a primitive type. A primitive type is a data type which cannot be decomposed into two or more components. A virtual type is created to allow a complex type to behave more like a primitive type . In practice the replacement process is implemented the same way custom data types are implemented in C.

A single grammar can have multiple BNF representations since there are multiple ways to structure a BNF definition. The structure of the definition affects the structure of the resulting tree therefore the simplest BNF representation of a grammar may not be the best to perform further processing. The BNF grammar implemented by the syntax analyser implements the C programming language however it was rewritten from the BNF definition given in the C specification.

Once the syntax analysis is complete, semantic analysis can take place. Semantic analysis is used to propagate data type information throughout the syntax tree. Before semantic analysis only some fundamental nodes have data type information. Identifiers (variables) and constants are the most common examples of nodes which acquire data type information during the syntax analysis phase.

During the semantic phase additional error checking also takes place. Errors detected include inappropriate use of data types, incompatible data type use as well as assertions regarding the structure and content of the syntax/semantic tree.

UNIVERSITY
JOHANNESBURG

Once the semantic pass has been completed the intermediate representation instructions must be generated. A logical instruction exists for each basic operation that is supported by the compiler even if the CPU does not have an equivalent instruction. If this is the case then several CPU instructions (physical instructions) might be required to implement the logical instruction. For example there is a logical instruction to perform a logic AND operation (not bitwise). IA-32 architecture CPUs do not have a suitable instruction therefore a logic AND instruction must be implemented using other instructions.

7.3.1 DDG generation

The division of the program into basic blocks is performed as the instructions are generated. Each different type of node in the compilation tree has a separate handler. The handlers for nodes that alter the flow of the program (like *if* statements) create the basic blocks.

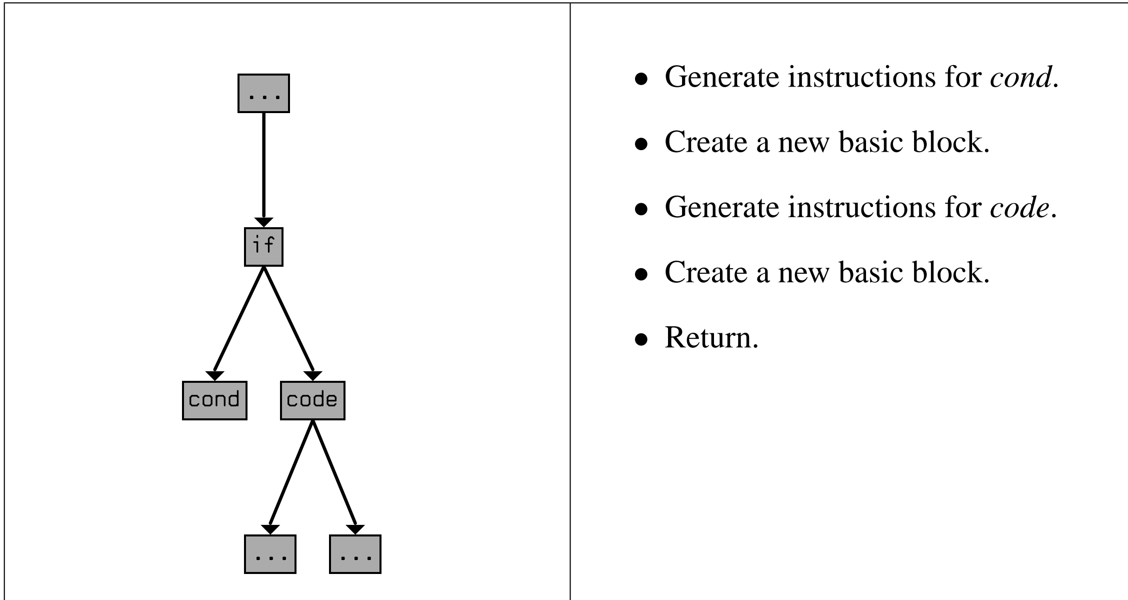


Figure 7.3: *An example instruction generator for an* `if` *statement. The steps taken are for demonstration and do not necessary reflect the steps taken in the implementation.* `cond` *refers to the* `if` *condition and* `code` *refers to the code executed if the condition is true. Both* `cond` *and* `code` *contain extra instructions that perform the relevant tasks.*

Figure 7.3 shows a part of the syntax graph for an if statement as well as the steps taken by the handler. The code segment executed if the condition evaluates to true must exist within a separate basic block from the code surrounding the if statement. The body of the if statement may be divided up into additional basic blocks, however the if statement only requires one basic block. Naturally the code following the true case of the if statement must exist within another basic block therefore the if statement handler must create two basic blocks.

7.4 Targeting and simplifier stages

The targeting and simplifier stages convert the intermediate code output of the compiler into machine specific code. Figure 7.4 shows the steps taken when targeting and simplifying the program.

The simplifier is responsible for performing simplifications of the program in order to improve performance. Typical situations that can be simplified include addition of two constants or multiplication by one. Simplification rules alter the program in its DDG form.

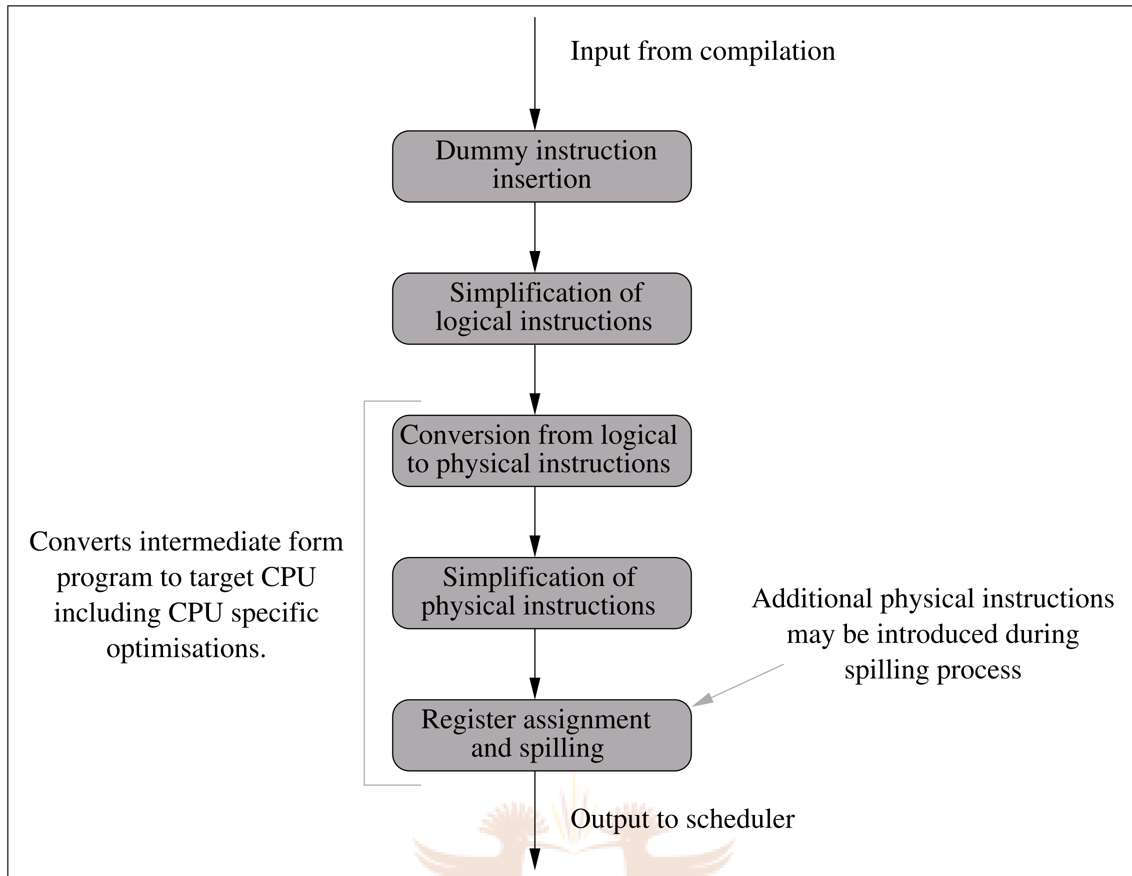


Figure 7.4: *A data flow diagram showing the modules that target and simply the program being compiled.*

UNIVERSITY OF JOHANNESBURG

The dummy instructions must be inserted before simplification can take place. The insertion of dummy instructions is shown in Figure 7.4 as occurring before simplification. The insertion of dummy instructions is performed by first inserting both dummy instructions and then traversing the list of instructions. Each instruction that is not dependent on any instructions is made a child of the starting dummy instruction. Each instruction that does not have at least one child instruction is made into a parent of the ending dummy instruction.

Following the addition of the dummy instructions a simplification pass takes place. This simplification pass is performed on a graph containing logical instructions and therefore is platform independent. The importance of logical simplification is that through the generation of the logical instructions certain optimisations may become possible. The logical simplification pass is shown in Figure 7.4 following the addition of the dummy instructions.

Once simplification of the logical program has taken place the instructions in the program

must be converted to physical instructions. Physical instructions are instructions that are supported by the CPU. Each logical instruction will be substituted by one or more physical instructions. The mapping information provided in the CPU description file is used for this purpose. When converting the logical program graph to a physical program graph, a new graph is created. Each physical instruction created for a single logical instruction inherits the properties of the logical instruction. Properties include the syntax tree reference and attributes. Although logical instructions have fields for an ending time and an execution unit, the status of these fields is undefined.

For each instruction in the logical graph the equivalent physical instructions are inserted into the physical graph. The dependencies between the physical instructions can only be inserted once all of the physical instructions have been inserted. Additional dependencies may be introduced due to the dependencies between the physical instructions used by the logical instruction. Regarding dependencies to and from the original logical instruction, there are two different behaviours depending on whether the data transfer is explicit or implicit (or no data at all). If the dependency involves explicit data transfer then only physical instructions that require the data as input inherit the dependency. If the dependency is not explicit then all of the physical instructions created to implement the logical instruction inherit the dependency.

Once the conversion from a graph containing logical instructions to a graph containing physical instructions is complete, the original logical graph is destroyed. Information regarding whether the instructions are logical instructions or physical instructions is stored with the graph.

Simplification is again performed after the conversion from logical instructions to physical instructions. This second simplification pass employs CPU specific optimisations and also allows for the use of specialised instructions. Apart from the difference in terms of type of instructions, it is identical to the simplification pass on the logical program.

Registers must be assigned to instructions before scheduling can take place because register assignment may introduce additional instructions. The additional instructions will be needed if there is an insufficient number of registers and spilling is necessary. Assigning registers to instructions in data dependency graph is a graph colouring problem [1]. Graph colouring problems are difficult to solve, however heuristic algorithms work well in practice. An evaluation of several heuristic algorithms for register assignment was presented by Chaitin et al. [7] and, more recently, Liberatore et al. [22]. The compiler implementation uses the heuristic algorithm introduced by Chaitin [6]. Although this algorithm is

fairly simple, it is a well known algorithm and performs predictably.

7.5 Scheduling and assembly program generation

The final steps involved with compilation is to schedule the instructions and generate the assembly language program. The methods introduced in Chapter 4 and Chapter 5 are implemented in order to schedule the instructions. Figure 7.5 shows the modules involved with scheduling and assembly language program generation.

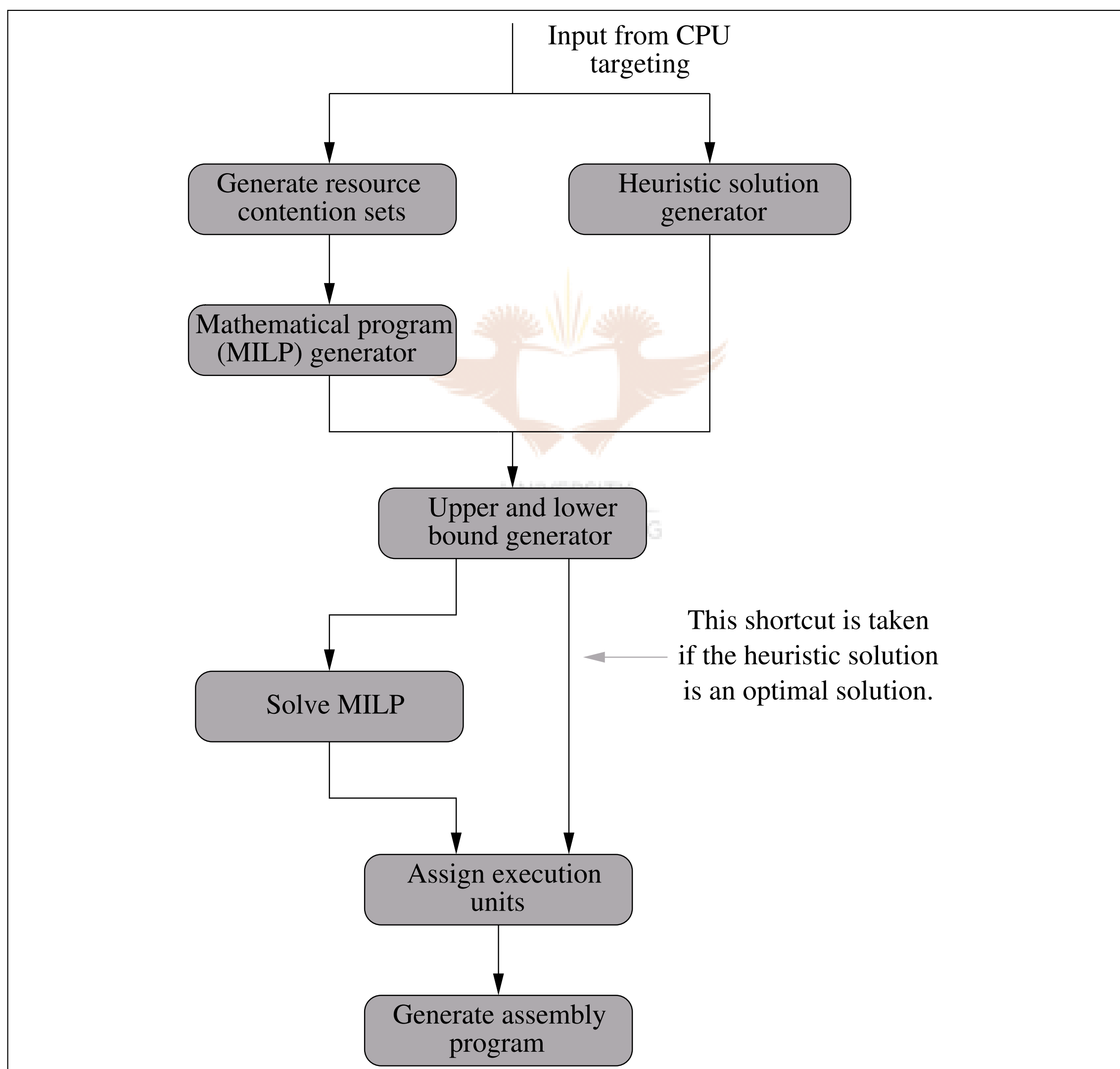


Figure 7.5: *A data flow diagram showing the scheduling and assembly language program generation modules. The final stage of compilation performed by the implementation is assembly language program generation. It produces a file containing the assembly code for the program.*

Before the mathematical program can be generated the resource contention set information must be extracted. As stated earlier, the representation for a resource contention set is fairly simple since it does not need to be altered once generated. The algorithm introduced in Chapter 4 was used in the compiler implementation to generate resource contention sets.

Once generated a resource contention set exists for each instruction in the graph. Only resource contention sets where the cardinality is greater than the number of supporting execution units are processed further. Pairs of instructions are generated from the execution units and are placed in an array for later use. The pairs of instructions will be used when generating the mathematical program in the manner described in Chapter 4.

Resource contention set information and the exceeding pairs list are stored separately from the program information. Once the mathematical program has been generated the resource contention set information will be released as it is no longer required.

7.5.1 MILP generator

The implementation of the optimiser includes an internal representation for mathematical programs. The accessor functions for the mathematical program structures allow programs to be expressed as mixed integer linear programs (MILP). The accessors functions also take care of generating suitable MILP variables and constraints to implement absolute difference constraints. Absolute difference constraints are specified as separate upper and lower bounds, allowing the caller to only generate one pair of bounds if that is all that is required.

Internally the compiler backend uses a *sparse* representation for the sake of efficiency. A sparse representation for a matrix is a matrix that is not stored as a two dimensional array, instead only non-zero values are stored. This delivers substantial benefits in this application since most constraints only reference two or three variables.

Once the mathematical program has been generated then upper and lower bounds are generated. Each variable defined in the mathematical program has an optional upper and lower bound. Some variables already have bounds as part of the generation processes, most notably 0-1 variables. Since the mathematical program exists before additional bounds are generated no temporary storage is required. The data can be placed directly into the mathematical program data structure which is passed as a parameter to the bound

generation functions.

The lower bounds are generated using Algorithm 5.1. The lower bound values are placed directly into the mathematical program's data structures as they are generated. Generating upper bounds are performed in a very similar manner to lower bounds except that Algorithm 5.2 requires a known feasible solution. Since the effectiveness of the known feasible solution has a significant impact on solution time, the makespan of the known solution should be a good solution.

The MSDA algorithm, which was introduced in Chapter 5, was implemented to generate good feasible solutions. In practice the heuristic solutions are quite good and sometimes they are optimal. A heuristic solution is an optimal solution when the upper bound of the ending dummy instruction is equal to the lower bound of the ending dummy instruction. The upper bound of the ending instruction is the makespan of the heuristic solution therefore if the makespan of the heuristic solution is equal to the lower bound then it is optimal. This short cut is shown in Figure 7.5 leaving the upper and lower bound generator.

Three different solvers were used during the development of the implementation. They were LP_solve [5], GLPK [23] and COIN (CLP and SBB components) [13]. The implementation can use any of the specified solvers. However only the GLPK solver was used to generate the published results since it was the only open source solver that was found to be both sophisticated enough and stable enough. LP_solve was too simple and has significant problems solving even fairly simple optimisation problems. The COIN solver was simply too immature for use as it would either crash or perform poorly depending on the various options chosen.

From the scheduler's perspective all solvers have an identical interface and can therefore be interchanged at will. The interface code for each solver is loaded dynamically or, in other words, once the program has already started execution. Only the chosen interface module is loaded as opposed to all of the supported modules, this would have been the case if the modules had been statically linked into the main program. Although the interface code to the solver is fairly small the code for the individual solvers is often quite large. By loading solvers on demand, load times and memory consumption are kept to a minimum.

The solver interface code must perform several functions. These functions include converting MILP data to the internal formats used by the solver, setting solver parameters and retrieving solution results and statistics. Some statistics were not available with

GLPK, most notably, the number of subproblems inspected before the optimal solution was found. Since this information is fairly important, the feature was added to the GLPK solver.

Once the optimal solution has been found (or the time limit has expired), the solution found by the solver must be retrieved and stored in the mathematical program's data structure. Once the optimal solution has been stored in the mathematical program data structure, the ending times for the instructions in the program can be updated. If the time limit expired then the solution may not be optimal or it may not be a solution at all. If the solution retrieved is inferior to the heuristic solution or no solution was available then the heuristic solution is used to update the instruction ending times.

7.5.2 Solution processing

Once a schedule has been obtained, the assignment of execution units must take place as shown in Figure 7.5. Execution unit assignment is performed by an implementation (and to some degree adaptation) of Algorithm 4.7. The original algorithm checked if the schedule was valid by assigning instructions to execution units.

A problem with using Algorithm 4.7 for this purpose is that it assumes that all execution units are identical. However, supporting different execution units requires a simple modification to Algorithm 4.7. Instead of having only one set for all of the executing instructions each type of execution unit must have its own set. The position in the set (assuming the sets are ordered) is used to uniquely identify the execution unit that the instruction is executing on.

The assignment of execution units serves an additional purpose. It can verify that a schedule is valid. This is to say the scheduled instructions do not attempt to use more executions units than are available. Invalid schedules should not occur, however during development this was not always the case, therefore validity checking was introduced.

The final step in the implemented compiler is the generation of assembly code from the scheduled instructions in the data dependency graphs. The function template provided in the CPU description file is used by the compiler to provide the bare bones of a function.

Before assembly code can be generated for the instructions an index of instructions must be created. Each instruction has an ending time and a duration. The starting time can be

computed from this information and is necessary for generating assembly code. The index of instructions contains instruction references in the order in which they will execute. The textual assembly representation for each instruction is generated in the order specified by the index.

Control sequence	Example	Description
%unit	%unit	Emit a reference to the execution unit that the instruction will execute on.
%in{number}	%in1	Emit register information for specific input of instruction specified by number. The example will output register information for the second parameter of the instruction.
%out{number}	%out0	Emit register information for specific output of instruction specified by number. The example will output register information for the first output of the instruction.
%addr	%addr	Emit the destination address associated with the instruction in the form of an attribute.
%const	%const	Emit textual representation a constant value associated with the instruction in the form of an attribute.
%string	%string	Emit the contents of a string literal associated with the instruction in the form of an attribute.
%symbol	%symbol	Emit a textual reference to a symbol associated with the instruction in the form of an attribute.

Table 7.5: *Table of control sequences and their meanings.*

UNIVERSITY OF JOHANNESBURG

Each instruction consists of three distinct parts, an instruction name, parameters and an execution unit reference. The actual textual representation is produced from the emit string given when the physical instruction was defined in the CPU description file. The emit string is sent to the output stream as literal text except when a control sequence is encountered. If a control sequence is encountered then it is substituted with the relevant information. Control sequences all begin with a % sign and are listed in Table 7.5.

As the assembly code is generated it is saved to a file in textual form. There is an option for creating a Gantt chart for each basic block. The implemented compiler leaves the program in this form unlike a production compiler. A production compiler would perform the necessary calls to assemble and link the assembly code. However, this is unnecessary for the purposes of the compiler implementation.

Chapter 8

Experimental results

The CPU definition file used to produce the results in this section is provided in Appendix C. The default CPU definition was altered for several tests, however the changes are described in each circumstance. As stated in Chapter 7, the mathematical programs were solved using version 4.7 of the GLPK solver.

Eight different test programs were compiled, each yielding one or more basic blocks. Each basic block was scheduled using the compiler implementation introduced in Chapter 7. Basic blocks with one or two instructions (excluding dummy instructions) are not reported in the following sets of results.

Each program contains a single function, the functions implement simple algorithms purely for the purpose of evaluation. A list of the programs is given below. Each program has a unique name and a description of what is does.

`bubble`	An implementation of the bubble sort algorithm.
`dotproduct`	Computes the dot product of two 1x3 vectors.
`fibonacci`	Computes the nth element in a fibonacci sequence where n is a hard-coded constant.

UNIVERSITY JOHANNESBURG

`matrix1` Multiplies a 1x3 vector with a 3x3 matrix in 3 iterations. The loop iterates for each row in the matrix.

`matrix2` Multiplies a 1x3 vector with a 3x3 matrix. The loops have been unrolled resulting in a static block of code with a large degree of parallelism. Unrolling a loop involves converting a loop body that is executed a fixed number of times into a single block of code that is executed once.

`primes` Counts the number of prime numbers between 0 and n where n is a hard-coded constant.

`search` Searches an array for a specified value returning the index of the value if found or -1 if it was not found.

`strcpy` Copies the content of a null terminated array of integers into another array. A null terminated array is an array where the end of the array is indicated by a 0 value element.

The source code for each test program is provided in Appendix D. The test programs only use integer variables so that the complexity of the CPU definition can be kept to a minimum. The implemented compiler also has better support for integer data types than some of the other data types.

UNIVERSITY OF JOHANNESBURG

8.1 Test results

In this section each test program was optimised for three different variations of a basic CPU. Therefore there are three sets of results where each set of results has an entry for each test program. Each set of results is presented in a separate table. The results within each table were conducted using the same solver configuration.

Two different solver options were specifically configured for the following tests. The first option is the method for deciding which variable should be selected when branching. The other option is which method should be used to choose an unfathomed branch, in other words the back-tracking method. The results in this section used the Driebeek Tomlin heuristic for variable selection and depth first back-tracking. The Driebeek Tomlin heuristic is described in the GLPK documentation [23]. A comparison of the different options is provided in the following section.

The following abbreviations and headings are used in Table 8.1, Table 8.2 and Table 8.3.

Test	Provides the name of the test the results refer to.
Block	An identifier for the basic block in the test program that the test results refer to.
LB	The lower bound on the makespan of the basic block.
UB	The upper bound on the makespan of the basic block. The upper bound is also equal to the makespan of the heuristic solution. If the upper bound is equal to the lower bound then the heuristic solution is an optimal solution.
Branches	The number of branches performed by the solver when optimising the mathematical program. This number is defined even if the solver did not find the optimal solution before the time limit expired.
Pivots	The number of pivots performed by the solver when optimising the mathematical program. This number is defined even if the solver did not find the optimal solution before the time limit expired.
Sol	The makespan of the basic block obtained from solving the mathematical program. If a value is not present in this field then the time limit was reached before the solver found an candidate solution. If an asterisk appears after the value then the time limit was reached before optimality could be guaranteed.
CV	The number of continuous variables in the mathematical program.
IV	The number of integer or 0-1 variables in the mathematical program.
Cntr	The number of constraints present in the mathematical program.

Table 8.1 shows the statistics for the various tests when optimised for the basic CPU description introduced at the beginning of this chapter.

In Figure 8.1 three basic blocks have identical results. Basic block 1 of the *fibonacci* test, basic block 0 of the *primes* test and basic block 0 of the *search* test have identical results. Inspection of the graphs in each of the three cases showed that they were identical with the exception of immediate values. The basic blocks in question implement part of a for loop which was identical in each of the three programs. This similarity implies that it may be possible to have a database of pre-scheduled basic blocks that are adapted to specific instances as required.

Table 8.2 shows the statistics for the various tests when optimised, however unlike the results in Table 8.1 changes were made to the CPU description file. In the default CPU

Test	Block	LB	UB	Branches	Pivots	Sol	CV	IV	Cntr
bubble	0	4	4						
	1	16	17	3	8	17	6	1	8
	2	20	22	8	27	21	12	3	20
	3	10	13	14	46	12	17	10	44
dotproduct	0	18	27	2901505	5514954	27*	40	70	172
fibonacci	0	3	4	13	30	4	7	4	16
	1	16	17	3	8	17	6	1	8
	2	8	11	14	43	9	12	9	39
matrix1	0	15	15						
	1	24	37			37*	137	266	567
matrix2	0	24	36			36*	74	167	368
primes	0	16	17	3	8	17	6	1	8
	1	36	36						
search	0	16	17	3	8	17	6	1	8
	1	18	20	6	17	19	9	2	14
strcpy	0	19	21	5	18	20	11	2	17
	1	6	7	8	32	7	12	4	24

Table 8.1: *The results of the test programs where load instructions have a duration of 2 clock cycles and only one execution unit supports load instructions.*

Test	Block	LB	UB	Branches	Pivots	Sol	CV	IV	Cntr
bubble	0	3	3						
	1	14	15	3	8	15	6	1	8
	2	17	18	7	21	18	12	3	20
	3	7	7						
dotproduct	0	11	17			17*	40	70	172
fibonacci	0	3	4	13	30	4	7	4	16
	1	14	15	3	8	15	6	1	8
	2	5	7	15	45	6	12	9	39
matrix1	0	14	14						
	1	22	28			28*	137	266	567
matrix2	0	16	42			42*	74	167	368
primes	0	14	15	3	8	15	6	1	8
	1	36	36						
search	0	14	15	3	8	15	6	1	8
	1	16	16						
strcpy	0	17	18	6	20	17	11	2	17
	1	4	5	7	16	4	12	4	24

UNIVERSITY OF JOHANNESBURG

Table 8.2: *The results of the test programs where load instructions have a duration of 1 clock cycle and only one execution unit supports load instructions.*

definition, load instructions have a duration of 2 clock cycles. The results in Table 8.2 assume that load instructions have a duration of 1 clock cycle instead.

Table 8.3 shows the results from the same tests as Table 8.2. The CPU description stated that each load instruction required 2 clock cycles. Unlike the two previous sets of results, it also specified that there are 2 execution units that support load instructions.

Test	Block	LB	UB	Branches	Pivots	Sol	CV	IV	Cntr
bubble	0	4	4						
	1	15	15						
	2	19	19						
	3	9	9						
dotproduct	0	13	19			19*	70	109	241
fibonacci	0	3	4	13	30	4	7	4	16
	1	15	15						
	2	7	9	16	52	7	15	14	46
matrix1	0	15	15						
	1	24	28			28*	186	326	676
matrix2	0	16	35			35*	140	245	512
primes	0	15	15						
	1	36	36						
search	0	15	15						
	1	18	18						
strcpy	0	19	19						
	1	5	5						

Table 8.3: *The results of the test programs where load instructions have a duration of 2 clock cycles and two execution units support load instructions.*

Notice that the heuristic solution was optimal in far more of the test cases than in previous tests indicating that load instructions are the bottle neck in most of these programs. The importance of load instructions and the occurrence of various types of instructions is discussed by Patterson and Hennessy in [25].

8.2 Impact of solver options

The solver used supports three different methods for selecting a variable to branch on. The three options are; branch on first variable, branch on last variable and the Driebeek Tomlin heuristic. The solver also supports three different methods for back-tracking, these are depth first, breadth first and best first.

Each test program was solved using every combination of branching and back-tracking methods. The results for each test program are presented in a separate table where the back-tracking methods are enumerated across the table and the branching methods down

the table. Each result consists of the total number of pivots followed by a comma and the number of nodes evaluated when branching. An asterisk replaces a field in the table when no result was obtained before the time limit was reached.

Basic block	Variable	Depth	Breadth	Best
0	**First**	8, 3	8, 3	8, 3
	Last	8, 3	8, 3	8, 3
	DT	8, 3	8, 3	8, 3
1	**First**	28, 9	28, 9	28, 9
	Last	28, 7	31, 8	31, 8
	DT	27, 8	27, 8	27, 8
2	**First**	55, 18	51, 15	51, 15
	Last	68, 17	75, 18	75, 18
	DT	46, 14	47, 15	47, 15

Table 8.4: *Test program* `bubble`. *Basic block 3 is not reported because it contained only one instruction.*

Table 8.4 contains results for effects of the various solver options on the `bubble` test program.

Basic block	Variable	Depth	Breadth	Best
0	**First**	1293764, 550793	611253, 243869	523810, 227915
	Last	1203985, 560448	811341, 272243	685532, 247796
	DT	*	477555, 221275	427849, 192940

Table 8.5: *Test program* `dotproduct`.

Table 8.5 contains results for effects of the various solver options on the `dotproduct` test program.

Basic block	Variable	Depth	Breadth	Best
0	**First**	29, 11	29, 11	29, 11
	Last	27, 12	27, 12	27, 12
	DT	30, 13	30, 13	30, 13
1	**First**	8, 3	8, 3	8, 3
	Last	8, 3	8, 3	8, 3
	DT	8, 3	8, 3	8, 3
2	**First**	39, 10	68, 24	68, 24
	Last	209, 47	156, 36	146, 32
	DT	43, 14	42, 14	42, 14

Table 8.6: *Test program* `fibonacci`.

Table 8.6 contains results for effects of the various solver options on the `fibonacci` test program.

Basic block	Variable	Depth	Breadth	Best
0	**First**	971587, 136990	*	*
	Last	494635, 170854	*	*
	DT	*	129699, 55300	136230, 55174

Table 8.7: *Test program* `matrix1`. *Basic block 1 is not reported because it contained only one instruction.*

Table 8.7 contains results for effects of the various solver options on the `matrix1` test program.

Basic block	Variable	Depth	Breadth	Best
0	**First**	*	*	*
	Last	644955, 260377	*	*
	DT	*	*	*

Table 8.8: *Test program* `matrix2`.

Table 8.8 contains results for effects of the various solver options on the `matrix2` test program.

Basic block	Variable	Depth	Breadth	Best
0	**First**	8, 3	8, 3	8, 3
	Last	8, 3	8, 3	8, 3
	DT	8, 3	8, 3	8, 3

Table 8.9: *Test program* `primes`. *Basic block 1 is not reported because it contained only one instruction.*

Table 8.9 contains results for effects of the various solver options on the `primes` test program.

Basic block	Variable	Depth	Breadth	Best
0	**First**	8, 3	8, 3	8, 3
	Last	8, 3	8, 3	8, 3
	DT	8, 3	8, 3	8, 3
1	**First**	16, 5	16, 5	16, 5
	Last	17, 5	17, 5	17, 5
	DT	17, 6	17, 6	17, 6

Table 8.10: *Test program* `search`.

Table 8.10 contains results for effects of the various solver options on the `search` test program.

Table 8.11 contains results for effects of the various solver options on the `strcpy` test program. Table 8.12 contains the sum of all of the previous results for a side by side

UNIVERSITY OF JOHANNESBURG

Basic block	Variable	Depth	Breadth	Best
0	**First**	15, 4	15, 4	15, 4
	Last	29, 7	29, 7	29, 7
	DT	18, 5	18, 5	18, 5
1	**First**	34, 9	37, 10	37, 10
	Last	38, 13	41, 16	41, 16
	DT	32, 8	32, 8	32, 8

Table 8.11: *Test program* `strcpy`*.*

comparison of the different options. Only tests results where no time limits were reached were considered therefore the `dotproduct`, `matrix1` and `matrix2` tests were not included.

Variable selection	Depth	Breadth	Best
First	248, 78	276, 90	276, 90
Last	448, 120	408, 114	398, 110
DT	245, 80	245, 81	245, 81

Table 8.12: *Totals of the number of pivots and number of nodes processed for each combination of solver options.*

According to Table 8.12 the best combinations of options are depth first / first variable and depth first / Driebeek Tomlin. However the results from the individual tests show that the best combination varies from test program to test program.

UNIVERSITY OF JOHANNESBURG

8.3 Further work

Improvements to the upper and lower bounds would decrease the computational time required by the solver on larger problems. The upper bound and lower bound generation techniques introduced in Chapter 5 could be improved in order to generate tighter bounds. Since computational time has proved to be a big obstacle improved bounds would be useful.

A bigger issue in terms of performance is the cardinality of the resource contention sets. An increase in cardinality of a resource contention set causes a quadratic increase in the number of variables and constraints. If resource contention sets can be reduced then the number of variables and constraints will be greatly reduced. Additional dependencies that do not affect the optimal solution can be introduced into the program undergoing optimisation. These additional dependencies reduce the cardinality of the resource contention

sets because elements within the sets become dependent (or transitively dependent) on some of the other elements in the set. Some experimentation in this regard took place during the development of the compiler implementation and looked fairly promising.

A significant number of integer variables were needed to implement the absolute difference constraints introduced in Chapter 5. It may be possible to implement absolute difference constraints in some other way, possibly by modifying the solver.

8.4 Conclusions

The heuristic scheduling algorithm introduced in Chapter 5 was unexpectedly effective. In most of the cases the optimal solution had very little, if any, advantage over the heuristic solution. However, there was a case where the heuristic algorithm performed poorly. In Table 8.2 the heuristic solution for the `matrix2` test has a significantly greater makespan than the same test in Table 8.1. This result is very poor when you consider that the tests in Table 8.2 had single cycle load instructions. The tests in Table 8.1 used two clock cycle load instructions so logic states that the heuristic makespan in Table 8.2 should be shorter or at least the same.

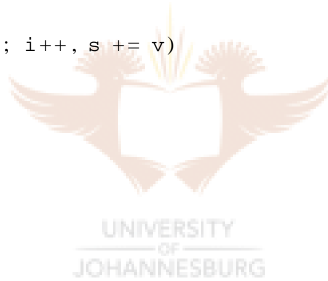
Programs with a large degree of parallelism cannot be optimised in a reasonable amount of time with the techniques developed. The tests performed did not implement any techniques for increasing parallelism like loop unrolling yet some basic blocks were still too large. In order for the introduced techniques to be useful, techniques for reducing the size of the resource contention sets will have to be developed.

Even though optimising large basic blocks is very time consuming, it may still be useful. Some basic blocks are very similar and the same solution can be used for each block. This was demonstrated in the results section where three different basic blocks were identical in terms of instructions and dependencies. The observation regarding the similarity of some basic blocks was made by Ramanan et al. [11] when analysing the basic blocks of the Livermore loops. The Livermore loops are a collection of loop bodies from Fortran implementations of many well known algorithms. The author concluded that a database of basic blocks could be maintained by the compiler to prevent optimisation of similar basic blocks.

Appendix A

Demonstrating implicit parallelism

```c
#include <stdio.h>
#include <time.h>

/* parallel multiplication test */
unsigned int test_mul_p(unsigned int reps)
{
    unsigned int v, s, i;
    for (i = 1, s = 0; i < reps; i++, s += v)
    {
        v = s ^ i;
        s ^= 110 * v;
        s ^= 211 * v;
        s ^= 312 * v;
        s ^= 413 * v;
        s ^= 514 * v;
        s ^= 615 * v;
        s ^= 716 * v;
        s ^= 817 * v;
    }
    return s;
}

/* non-parallel multiplication test */
unsigned int test_mul_np(unsigned int reps)
{
        unsigned int v, s, i;
    for (i = 1, s = 0; i < reps; i++, s += v)
    {
        v = s ^ i;
        v ^= 110 * v;
        v ^= 211 * v;
        v ^= 312 * v;
        v ^= 413 * v;
        v ^= 514 * v;
        v ^= 615 * v;
        v ^= 716 * v;
        v ^= 817 * v;
    }
    return s;
}

/* parallel division test */
unsigned int test_div_p(unsigned int reps)
{
    unsigned int v, s, i;
    for (i = 1, s = 0; i < reps; i++, s += v)
    {
        v = s ^ i;
        s ^= 110 / v;
        s ^= 211 / v;
```



```c
        s ^= 312 / v;
        s ^= 413 / v;
        s ^= 514 / v;
        s ^= 615 / v;
        s ^= 716 / v;
        s ^= 817 / v;
    }
    return s;
}

/* non-parallel division test */
unsigned int test_div_np(unsigned int reps)
{
        unsigned int v, i;
    for (i = 1; i < reps; i++)
    {
        v = v ^ i;
        v ^= 110 / v;
        v ^= 211 / v;
        v ^= 312 / v;
        v ^= 413 / v;
        v ^= 514 / v;
        v ^= 615 / v;
        v ^= 716 / v;
        v ^= 817 / v;
    }
    return v;
}

/* returns starting time which is aligned to the start of a second */
time_t synctime()
{
    time_t st, t;

    st = time(NULL);
    do { t = time(NULL); } while(st == t);

    return t;
}

void print_time(char *str, int reps, time_t s1, time_t s2)
{
    double rate;

    s2 -= s1;
    rate = (s2 > 0) ? rate = (double)reps / (double)s2 : rate = reps;

    /* Each rep performs 32  */
    rate *= 32;

    printf("%s:␣%d␣seconds\n", str, (int)s2);
    printf("%.4g␣Million␣per␣second\n\n", rate / 1000000.0);
}

int main(int argc, char *argv[])
{
    unsigned int d, reps;
    time_t s1, s2;

    reps = atoi(argv[1]);

    s1 = synctime();
    d ^= test_mul_n(reps); d ^= test_mul_n(reps);
    d ^= test_mul_n(reps); d ^= test_mul_n(reps);
    s2 = time(NULL);
    print_time("Non-parallel␣multiplications", reps, s1, s2);

    s1 = synctime();
    d ^= test_mul_p(reps); d ^= test_mul_p(reps);
    d ^= test_mul_p(reps); d ^= test_mul_p(reps);
    s2 = time(NULL);
    print_time("Parallel␣multiplications", reps, s1, s2);

    s1 = synctime();
    d ^= test_div_n(reps); d ^= test_div_n(reps);
    d ^= test_div_n(reps); d ^= test_div_n(reps);
```

UNIVERSITY OF JOHANNESBURG

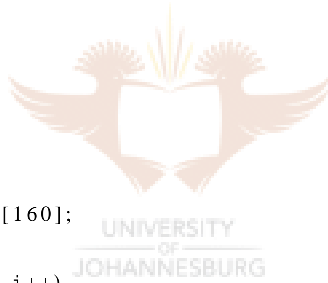
```
    s2 = time(NULL);
    print_time("Non-parallel divisions", reps, s1, s2);

    s1 = synctime();
    d ^= test_div_p(reps); d ^= test_div_p(reps);
    d ^= test_div_p(reps); d ^= test_div_p(reps);
    s2 = time(NULL);
    print_time("Parallel divisions", reps, s1, s2);

    /* returns the data so that the optimiser cannot remove the workload */
    return d;
}
```

Appendix B

Boundary evaluation test cases

```
void fixed_length_copy(int n)
{
        int intstr[200], othstr[160];
    int i;

    for (i = 0; i < n; i++)
    {
        othstr[i] = intstr[i];
    }
}
```

```
void null_term_copy(void)
{
        int intstr[200], othstr[160];
    int i;

    for (i = 0; intstr[i] != 0; i++)
    {
        othstr[i] = intstr[i];
    }
}
```



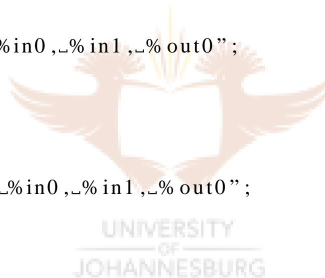
Appendix C

CPU definition file

```
description
{
 name              "Fictional VLIW CPU";
 model             "";
 addresses         32;
 strict alignment  1;
 alignbytes        4;
 baseptr_top       0;
 baseptr_bottom    8;
}

registers
{
   new regs 128;
}

logical
{
     npdeps sdummy()()  sdummy()();
     npdeps edummy()()  edummy()();

     // General arithmetic
     addls($0, $1)($2)  addl($0, $1)($2);
     addl($0, $1)($2)   addl($0, $1)($2);
     subls($0, $1)($2)  subl($0, $1)($2);
     subl($0, $1)($2)   subl($0, $1)($2);
     shrls($0, $1)($2)  shrls($0, $1)($2);
     shrl($0, $1)($2)   shrl($0, $1)($2);
     shlls($0, $1)($2)  shlls($0, $1)($2);
     shll($0, $1)($2)   shll($0, $1)($2);
     incls($0)($1)      incl($0)($1);
     incl($0)($1)       incl($0)($1);
     decls($0)($1)      decl($0)($1);
     decl($0)($1)       decl($0)($1);

     // Multipliers
     mulls($0, $1)($2)  mulls($0, $1)($2);
     mull($0, $1)($2)   mull($0, $1)($2);

     // Dividers
     divls($0, $1)($2)  divls($0, $1)($2);
     divl($0, $1)($2)   divl($0, $1)($2);
     modls($0, $1)($2)  modl($0, $1)($2);
     modl($0, $1)($2)   modl($0, $1)($2);

     // bitwise
     andl($0, $1)($2) andl($0, $1)($2);
     orl($0, $1)($2)   orl($0, $1)($2);
     xorl($0, $1)($2) xorl($0, $1)($2);

     // Logic
```

```
    landl($0, $1)($2)  landl($0, $1)($2);
    lorl($0, $1)($2)   lorl($0, $1)($2);

    // Relational operators
    equls($0, $1)($2)   equl($0, $1)($2);
    equl($0, $1)($2)    equl($0, $1)($2);
    nequls($0, $1)($2)  nequl($0, $1)($2);
    nequl($0, $1)($2)   nequl($0, $1)($2);

    ltls($0, $1)($2)   ltls($0, $1)($2);
    ltl($0, $1)($2)    ltl($0, $1)($2);
    ltels($0, $1)($2)  ltels($0, $1)($2);
    ltel($0, $1)($2)   ltel($0, $1)($2);
    gtls($0, $1)($2)   gtls($0, $1)($2);
    gtl($0, $1)($2)    gtl($0, $1)($2);
    gtels($0, $1)($2)  gtels($0, $1)($2);
    gtel($0, $1)($2)   gtel($0, $1)($2);

    // General loads and stores
    loadls($0)($1)       loadl($0)($1);
    loadl($0)($1)        loadl($0)($1);
    storels($0, $1)()    storel($0,$1)();
    storel($0, $1)()     storel($0,$1)();
    bploadl()($0)    bpload()($0);

    // BP+V and BP-V loads
    ldaddls()($0)        ldaddl()($0);
    ldaddl()($0)         ldaddl()($0);
    ldsubls()($0)        ldsubl()($0);
    ldsubl()($0)         ldsubl()($0);

    // BP+V and BP-V stores
    staddls($0)()        staddl($0)();
    staddl($0)()         staddl($0)();
    stsubls($0)()        stsubl($0)();
    stsubl($0)()         stsubl($0)();

    // immediate loads and data movement
    immls()($0)          loadimml()($0);
    imml()($0)           loadimml()($0);
    movls($0)($1)        movl($0)($1);
    movl($0)($1)         movl($0)($1);

    // Branch to hardcoded-relative if true
    tbranchls($0)()       tbranchl($0)();
    tbranchl($0)()        tbranchl($0)();
    fbranchls($0)()       fbranchl($0)();
    fbranchl($0)()        fbranchl($0)();
    jumpl()()         jump()();

    // special branch instructions
    branchgtels($0, $1)()   equl($0, $1)($2) tbranchl($2)();
    branchgtel($0, $1)()    equl($0, $1)($2) tbranchl($2)();

    // Return
    returnls()()       return()();
    returnl()()        return()();

    // Call
    callls()()            call()();
    calll()()             call()();
}

physical
{

  // //////////////////////////////
  // Mandatory dummy unit
  // //////////////////////////////
  new Dummy
  {
    // dummy instructions
    sdummy { }
    edummy { }
  }
```

```
// /////////////////////////////
// The arithmetic logic units
// /////////////////////////////
new ALU
{
  addl  {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   addl %in0, %in1, %out0";
            }
  subl  {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   subl %in0, %in1, %out0";
            }
  shll  {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   shll %in0, %in1, %out0";
            }
  shlls {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   shlls %in0, %in1, %out0";
            }
  shrl  {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   shrl %in0, %in1, %out0";
            }
  shrls {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   shrls %in0, %in1, %out0";
            }
  incl  {
      in:       regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   incl %in0, %out0";
            }
      decl  {
      in:       regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   decl %in0, %out0";
            }

  // 32Bit  Bitwise  instructions
  andl  {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   andl %in0, %in1, %out0";
            }
  orl   {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   orl %in0, %in1, %out0";
            }
  xorl  {
      in:       regs, regs;
      out:      regs;
      latency: 1;
      emit:     "%unit   xorl %in0, %in1, %out0";
            }

  // 32Bit  Logic  instructions
```



```
    landl  {
        in:       regs, regs;
        out:      regs;
        latency: 1;
        emit:     "%unit␣␣␣landl␣%in0,␣%in1,␣%out0";
              }
    lorl   {
        in:       regs, regs;
        out:      regs;
        latency: 1;
        emit:     "%unit␣␣␣lorl␣%in0,␣%in1,␣%out0";
              }

     }

  new MUL  {
    mull  {
        in:       regs, regs;
        out:      regs;
        latency: 7;
        emit:     "%unit␣␣␣mull␣%in0,␣%in1,␣%out0";
              }
    mulls {
        in:       regs, regs;
        out:      regs;
        latency: 8;
        emit:     "%unit␣␣␣mulls␣%in0,␣%in1,␣%out0";
              }
    }

    new DIV {
    divl  {
        in:       regs, regs;
        out:      regs;
        latency: 23;
        emit:     "%unit␣␣␣divls␣%in0,␣%in1,␣%out0";
              }
    divls {
        in:       regs, regs;
        out:      regs;
        latency: 24;
        emit:     "%unit␣␣␣divls␣%in0,␣%in1,␣%out0";
              }
    modl  {
        in:       regs, regs;
        out:      regs;
        latency: 23;
        emit:     "%unit␣␣␣modl␣%in0,␣%in1,␣%out0";
              }
     }

  new CMP {
    equl  {
        in:       regs, regs;
        out:      regs;
        latency: 1;
        emit:     "%unit␣␣␣equl␣%in0,␣%in1,␣%out0";
              }
    nequl  {
        in:       regs, regs;
        out:      regs;
        latency: 1;
        emit:     "%unit␣␣␣equl␣%in0,␣%in1,␣%out0";
              }
    ltls  {
        in:       regs, regs;
        out:      regs;
        latency: 1;
        emit:     "%unit␣␣␣ltls␣%in0,␣%in1,␣%out0";
              }
    ltl  {
        in:       regs, regs;
        out:      regs;
        latency: 1;
        emit:     "%unit␣␣␣ltl␣%in0,␣%in1,␣%out0";
```

UNIVERSITY OF JOHANNESBURG

```
            }
 ltels  {
     in:       regs, regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   ltels %in0, %in1, %out0";
            }
 ltel  {
     in:       regs, regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   ltel %in0, %in1, %out0";
            }
 gtls  {
     in:       regs, regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   gtls %in0, %in1, %out0";
            }
 gtl  {
     in:       regs, regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   gtl %in0, %in1, %out0";
            }
 gtels  {
     in:       regs, regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   gtels %in0, %in1, %out0";
            }
 gtel  {
     in:       regs, regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   gtel %in0, %in1, %out0";
            }
}
```

```
new MISC
{
 movl  {
     in:       regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   movl %in0, %out0";
            }
 pushl {
     in:       regs;
     latency: 1;
     emit:     "%unit   pushl %in0";
            }
 popl  {
     out:      regs;
     latency: 1;
     emit:     "%unit   popl %out0";
            }

 loadimml {
     out:      regs;
     latency: 1;
         emit:     "%unit   movl %const, %out0";
        }

 bpload {
     out:      regs;
     latency: 1;
         emit:     "%unit   movl bp, %out0";
          }
 //
 // Branch and jump instructions
 //
 tbranchl {
        in:       regs;
        latency: 12;
```

```
        emit:     "%unit   tbranch %addr";
    }
  fbranchl {
        in:       regs;
        latency: 12;
        emit:     "%unit   fbranch %addr";
    }
  jump {
        latency: 4;
        emit:     "%unit   jmp %addr";
    }


  return {
        latency: 4;
        emit:     "%unit   return";
    }

  call {
        latency: 4;
        emit:     "%unit   call %addr";
    }

  //
  // Integer/Integer conversion instructions
  //
  cv_to_bsw  {
     in:       regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   b2w %in0, %out0";
           }
  cv_to_wsl  {
     in:       regs;
     out:      regs;
     latency: 1;
     emit:     "%unit   l2w %in0, %out0";
           }
  }

  new LD
  {
  loadl {
     in:       regs;
     out:      regs;
     latency: 2;
         emit:     "%unit   movl %in0(0), %out0";
       }
  ldaddl {
     out:      regs;
     latency: 2;
         emit:     "%unit   movl bp(%const), %out0";
        }
  ldsubl {
     out:      regs;
     latency: 2;
         emit:     "%unit   movl bp(-%const), %out0";
        }
  }

 new ST
 {
  storel {
     in:       regs, regs;
     latency: 1;
         emit:     "%unit   movl %in0, %in1(0)";
       }
  staddl {
     in:       regs;
     latency: 1;
         emit:     "%unit   movl %in0, bp(%const)";
        }
  stsubl {
     in:       regs;
     latency: 1;
```

```
            emit:    "%unit   movl %in0, bp(-%const)";
           }
   }

   // ////////////////////////////////////
   //        Additional execution units
   // ALU, CMP, MUL, DIV, MISC, LD, ST
   // ////////////////////////////////////
   // 4x arithmetic logic units
   clone ALU_1 ALU;
   clone ALU_2 ALU;
   clone ALU_3 ALU;

   // 4x multiplication units
   clone MUL_1 MUL;
   clone MUL_2 MUL;
   clone MUL_3 MUL;

   // 2x division units
   clone DIV_1 DIV;
}
```

Appendix D

Source code for the experimental results

```
void bubble(int *data, int size)
{
    int i, swap, temp;

    size--;
    swap = 1;
    while(swap)
    {
        swap = 0;
        for (i = 0; i < size; i++)
            if (data[i + 1] < data[i])
            {
                swap = 1;

                /* swap values */
                temp = data[i];
                data[i] = data[i + 1];
                data[i + 1] = temp;
            }
    }
}

void dotproduct(int *v1, int *v2, int *r)
{
    r[0] = v1[0] * v2[0];
    r[1] = v1[1] * v2[1];
    r[2] = v1[2] * v2[2];
}

int fibonacci(int n)
{
    int i, a, b, c;

    a = 1;
    b = 1;

    for (i = 0; i < n; i++)
    {
        c = a + b;
        a = b;
        b = c;


    }

    return b;
}
```

```
void matrix1()
{
    int r[3], v[3];
    int mtx[9];
    int i;

    for (i = 0; i < 3; i++)
        r[i] = v[0] * mtx[i * 3 + 0] +
               v[1] * mtx[i * 3 + 1] +
               v[2] * mtx[i * 3 + 2] +
                      mtx[i * 3 + 3];
}

void matrix2(int v1, int v2, int v3)
{
    int r1, r2, r3;
    int m11, m12, m13, m21, m22, m23, m31, m32, m33;

    r1 = v1 * m11 + v2 * m21 + v3 * m31;
    r2 = v1 * m12 + v2 * m22 + v3 * m32;
    r3 = v1 * m13 + v2 * m23 + v3 * m33;
}

void primes()
{
    int i, j;

    for (i = 3; ; i += 2)
        for (j = 2; j < i; j++)
            if (i % j == 0)
                break;
}

int search(int *data, int needle, int size)
{
    int i;

    for (i = 0; i < size; i++)
        if (data[i] == needle)
            return i;
    return -1;
}

void strcpy(int *dest, int *src)
{
    int i;

    for (i = 0; src[i] != 0; i++)
        dest[i] = src[i];
}
```

Bibliography

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, 1988.

[2] Asian Technology Information Program (ATIP). Sony playstation 2 and hpc. Technical Report ATIP99.091r, 1999.

[3] D.I. August, D.A. Connors, J.C. Gyllenhaal, and W.W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *The 3rd International Symposium on High-Performance Computer Architecture*, pages 84–93, 1997.

[4] K.R. Baker. *Introduction to Sequencing and Scheduling*. John Wiley & Sons, New York, 1974.

[5] M. Berkelaar. lp_solve Version 4.0 - LP solving software ftp://ftp.ics.ele.tue.nl/pub/lp_solve/.

[6] G.J. Chaitin. Register allocation & spilling via graph coloring. In *SIGPLAN Symposium on Compiler Construction*, pages 98–105, 1982.

[7] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Comput. Lang.*, 6(1):47–57, 1981.

[8] Intel Corporation. Desktop performance and optimization for Intel Pentium 4 processor, Feb 2001.

[9] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures*. International Thomson Computer Press, 1995.

[10] H.G. Cragon. *Computer architecture and implementation*. Cambridge University Press, 2000.

UNIVERSITY
JOHANNESBURG

[11] A. Dani, V. Ramanan, and R. Govindarajan. Register-sensitive software pipelining. In *Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 194–198, Los Alamitos, March 30–April 3 1998.

[12] D. Dougherty and A. Robbins. *Sed & Awk*. O'Reilly & Associates, Inc, second edition, 1997.

[13] COIN-OR Foundation. COIN-OR's CLP and SBB/CBC development version http://www.coin-or.org.

[14] Free Software Foundation. Bison - Yacc compatible parser generator http://www.gnu.org/software/bison/bison.html.

[15] Free Software Foundation. Flex - Lex compatible lexical analyser generator http://www.gnu.org/software/flex/flex.html.

[16] S. French. *Sequencing and scheduling*. Ellis Horwood Ltd, 1982.

[17] H.H. Greenberg. A branch-bound solution to the general scheduling problem. *Operations Research*, 16:353–361, 1968.

[18] K.R. Irvine. *Assembly Language for Intel-Based Computers*. Prentice-Hall, Inc, third edition, 1999.

UNIVERSITY OF JOHANNESBURG

[19] D. Kästner and M. Langenbach. Code optimization by integer linear programming. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction*, pages 122–136. Springer-Verlag, 1999.

[20] K.E. Kendall and J.E. Kendall. *Systems analysis and design*. Prentice-Hall, Inc, fourth edition, 1998.

[21] C. Kozyrakis and D. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 283–293, 2002.

[22] V. Liberatore, M. Farach-Colton, and U. Kremer. Evaluation of algorithms for local register allocation. In *Lecture Notes in Computer Science*, volume 1575, pages 137–153. Springer-Verlag, 1999.

[23] A.O. Makhorin. GLPK Version 4.7 - LP solving software http://www.gnu.org/software/glpk/glpk.html.

[24] N.J. Nilsson. *Artificial Intelligence : a new synthesis*. Morgan Kaufmann Publishers, Inc, 1998.

[25] D.A. Patterson and J.L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, Inc, third edition, 2003.

[26] C. Pfleeger. *Security in Computing*. Prentice-Hall International, Inc, Upper Saddle River, second edition, 1997.

[27] K. Sankaralingam, S. Keckler, W. Mark, and D. Burger. Universal mechanisms for data-parallel architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[28] A.T. Shreiner and H.G. Friedman. *Introduction to Compiler Construction with Unix*. Prentice-Hall, Inc, 1985.

[29] G.S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[30] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc, 1999.

[31] W.L. Winston. *Operations research : applications and algorithms*. International Thomson Publishing, Belmont, California, third edition, 1993.

[32] H. Yang, R. Govindarajan, G. Gao, and T. Theobald. Power-performance trade-offs for energy-efficient architectures: a quantitative study. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 174–179. IEEE, September 2002.

[33] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

Index

alias, 106
aliasing, 92
ALU, 9
arithmetic logic unit, 9
assembly, 117
attributes, 103

basic block, 43, 110
benchmarks, 119
BNF, 109, 110
bound, 115
    lower, 115
    upper, 115
boundary assertions, 90
branch, 9
    conditional, 9
    predication, 29
    prediction, 22
    unconditional, 9
branch and bound, 68

cache, 14
clone, 107
compilation
    intermediate, 38
    intermediate code, 40
    lexical analysis, 33
    parsing, 32
    phases, 32
    semantic analysis, 36
    syntax analysis, 34
    token, 33
    tool chain, 31
constraints
    dependencies, 48
    starting instructions, 48
contention, 55
    reducing, 56
conversion, 112

data dependency graph, 40
data type, 109
DDG, 40, 102, 110
dependencies, 48
    constraints, 48
dot product, 119
dummy, 111

ending instructions, 49
EPIC, 25
error reporting, 103
execution unit, 13, 52, 55–57, 81, 107, 117
    different, 82
    multiple, 60
    single, 53
    types, 82
explicitly parallel, 25

fibonacci, 119
function, 117

generate, 57
grammar, 110

UNIVERSITY OF JOHANNESBURG

heuristic, 116
    MSDA, 116

I/O, 6
IA-32, 16
ILP, 16
implicit, 16
in-order, 25
instruction
    order, 20
    searching, 17, 21
instruction level parallelism, 16
interleave, 29
intermediate
    generation, 38, 40
    quadruples, 38
    triples, 38
intersection, 57
intersection cardinality, 57

job, 46
job shop problem, 46, 47
jump, 9

large constant, 78
latency, 10, 107
lexical analyser, 108
lexical analysis, 33
load, 8, 93
    strided, 26
logical, 110, 112

M, 78
machine, 46
makespan, 47, 49, 81
matrix, 120
memory
    latency, 10
    logical, 13
    management unit, 14
    physical, 13
microcode, 20
MILP, 115
minimise schedule, 68
minimising the max, 49
MMU, 14
mutex, 106

new, 106, 107

objective function, 50
opt-out, 20
out-of-order, 20

page, 13, 92
    fault, 92
    map, 13
    rights, 13
paging, 13
parallel machine problem, 46, 47
parallelism, 16, 24, 55
parsing, 32
physical, 112, 113
primes, 120

queueing, 17

RCS, 103, 114
register, 26
    assignment, 113
    spilling, 12, 113
resource, 52
    contention, 55
    contention set, 55–57, 114
    disjunctive method, 53
    multiple, 60
runtime, 17

schedule, 117

UNIVERSITY OF JOHANNESBURG

scheduling, 17, 46
search, 120
security, 90
segments, 13
semantic, 110
semantic analysis, 36
shared memory, 92
SIMD, 26
simplifier, 111
simplify, 112, 113
solver, 116
    CLP, 116
    COIN, 116
    GLPK, 116
    LPSolve, 116
    SBB, 116
sort
    bubble, 119
sparse, 115
special applications
    bounds, 90
speculative execution, 22
spilling, 12, 113
starting instructions, 48
storage
    accumulator, 11
    general purpose registers, 10
    memory, 10
    stack, 12
    work, 10–12
store, 8, 93
strcpy, 120
superscalar, 16
superscalar execution, 16
syntax analyser, 109
syntax analysis, 34

tests, 119
token, 33
tool
    chain, 31
type, 110
typedef, 109

variables, 47
    integer, 50
    real, 50
vector, 26
verification, 117
virtual, 109
VLIW, 27
Von Neumann, 6

x86, 16