MASTER

A VLIW DSP data path with multiple controllers

Vrijnsen, J.H.G.M.

Award date:
2003

TU/e technische universiteit eindhoven

Section of Information and Communication Systems (ICS/ES)
Faculty of Electrical Engineering
ICS/ES 826

Master's Thesis

A VLIW DSP Data Path with Multiple Controllers

J.H.G.M. Vrijnsen

Coach: dr.ir. B. Mesman
Supervisor: prof.dr.ir. J.L. van Meerbergen
Date: 17 March 2003 – 28 November 2003

The Faculty of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses

Authors' address	Vrijnsen, J.H.G.M.	J.H.G.M.Vrijnsen@student.tue.nl JeroenVrijnsen@yahoo.com

Preface

This report is my Master's Thesis, resulting from my graduation project for my studies in Electrical Engineering at the Eindhoven University of Technology. The graduation project was performed within Philips Research, Eindhoven, in the department Embedded Systems Architectures on Silicon (ESAS), which belongs to the Information and Software Technology (IST) sector. Since many people have helped me in completing this thesis, I would like to take this opportunity to thank those who helped me.

First, I would like to thank my supervisors, Bart Mesman and Jef van Meerbergen, for providing me with the option to perform my graduation project within Philips Research. Both have given me a lot of freedom to do things "my way". Furthermore, I would like to thank Albert van der Werf for allowing me to perform my graduation project within the ESAS department.

Nur Engin has helped me a lot with understanding and implementing the Viterbi algorithm. I would like to thank her for the discussions we had on this subject.

From Silicon Hive I would like to thank Lex Augusteijn, Jeroen Leijten, Jos Huisken, Wim Yedema, Antoine van Wel and Marc Quax. They have helped me a lot in understanding and (ab-) using the Silicon Hive tools.

In addition, I would like to thank Marco Bekooij for the discussions we had on scheduling and on the differences between the A|RT Designer and Silicon Hive tools.

My colleagues from the ESAS department have made my stay a very pleasant one. All offered me useful insight in both my work and theirs, and have helped me in completing this project. Thank you all.

I am grateful to my parents, for always being there when I needed them. Their education gave me the attitude and sense of responsibility that was needed to earn my master's degree. In addition, I am grateful to my brother, who was kindly enough to support me in writing my thesis, and pointing out the flaws in it –or at the least the parts he did not understand.

Last but certainly not least, I would to thank my girl friend for supporting me, although she did not understand much of my project. Marjan, thank you for being there and for brighten me up when I needed it.

Of course are any flaws in this thesis the result of my own inexperience and the misinterpretation of good advice.

Abstract

In order to exploit the large amount of instruction-level parallelism offered by a VLIW data path, designers consider the inner loops in streaming digital signal processing algorithms. In general, about eighty percent of the run-time in DSP algorithms is spent in twenty percent of the application. Furthermore, many applications allow a certain degree of loop overlapping. Exploiting this overlapping can reduce the total execution time and the amount of intermediate data that has to be stored.

Several different implementations that allow the overlapping execution of loops are possible. In this report, we study the effect of mapping DSP algorithms on a VLIW data path with multiple controllers, each of them controlling a different loop. We will show how to generate static schedules for the controllers, using currently available design tools, since no commercial design tools are available yet with an instruction scheduling scope spanning multiple controllers. Furthermore, we compare the results of mapping a DSP algorithm (the Viterbi algorithm) on a VLIW data path with multiple controllers to mapping the same algorithm on a regular, single threaded VLIW.

Keywords: VLIW, DSP, scheduling, mapping, multi-threading

Table of Contents

List of Figures

1. Introduction

In recent years, *very long instruction word* (VLIW) approaches have become increasingly prominent in the *digital signal-processing* (DSP) world. Digital signal-processing refers to various techniques for improving the accuracy and the reliability of digital communication applications, for example filter algorithms (e.g., FFT or DCT[1]) or video processing algorithms. The reasons to use VLIW architectures for such kind of applications are straightforward: VLIW architectures provide parallel execution of operations to increase the performance significantly.

A global VLIW architecture template is shown in Figure 1-1. Such an architecture has the following characteristics:

1. The most important characteristic of a VLIW architecture is the presence of several *functional units* (FUs) that can operate in parallel, under control of a single controller. Often the units are all different, but also multiple instances of the same type may occur (e.g., multiple arithmetic-logic units). Each of those (or a subset of the) functional units is controlled by a slot in the instruction word, which results in the very long instruction words, as shown in the figure;
2. Each of the functional units works with operands that are available from registers fields in register files. Several register files are possible, which are all controlled by a particular slot in the instruction word. All input- and output data that is active during the same clock cycle is read from or written to a register field in a register file. Every input of a functional unit can be supplied with a register file of its own, or register files can be merged into one register file (and of course any solution in between is also possible);
3. In principle, every functional unit can receive a new instruction on each clock cycle. When the instruction word would become too wide, functional units can be grouped into *issue slots*. Each of the issue slots is controlled by a single partition of the instruction word. Therefore, multiple functional units are controlled by the same partition of the instruction word, resulting in a smaller instruction word. The width of the instruction word and the corresponding program code size form the principal disadvantages of VLIW architectures, especially for embedded applications. The reason for this is that for such applications only a limited amount of memory is available, due to power consumption requirements and the costs of on-chip memory.

The VLIW architecture in a sense has the same capabilities as a superscalar processor: issuing and completing multiple operations each clock cycle (superscalar means the ability to fetch, issue to functional units and complete more than one instruction at a time). However, a major difference with superscalar processors is that the VLIW hardware is not responsible for discovering opportunities to execute multiple operations concurrently. Instead of that, for a VLIW architecture, it is the task of the compiler to

[1] FFT: Fast Fourier Transformation   DCT: Discrete Cosine Transformation

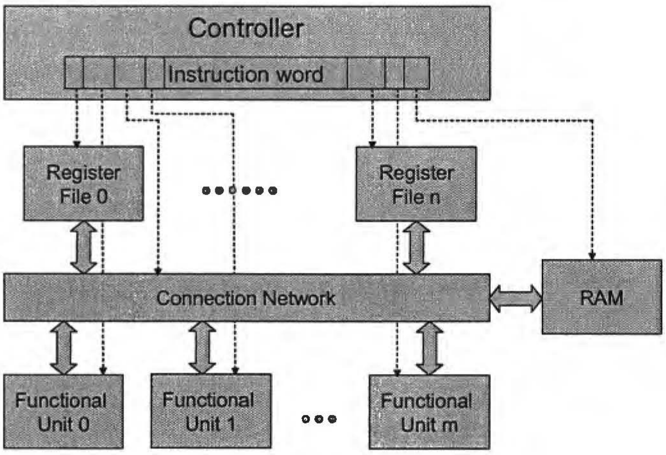


Figure 1-1: Global overview of VLIW architecture

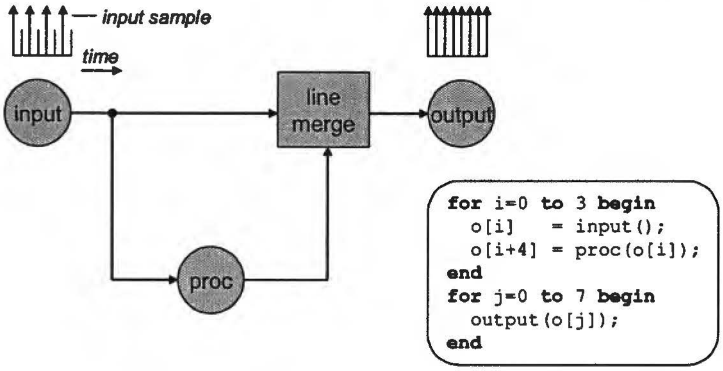
detect the *instruction-level parallelism* (ILP) -independent instructions from one block of code that can be executed in parallel- available in the application and to generate a schedule that is as short as possible. Therefore, in contrary to superscalar processors where scheduling hardware is used to detect the parallelism at run-time, this is done at compile-time in a VLIW architecture, which leads to a reduced hardware complexity compared to a superscalar architecture.

Due to the presence of multiple functional units that can be active simultaneously and due to the reduced hardware complexity with respect to other architectural alternatives, VLIW architectures are well suited to exploit the instruction-level parallelism that is present in many applications.

1.1. Problem statement

So, in order to exploit the large amount of instruction-level parallelism offered by a VLIW data path, designers consider the inner loops in streaming digital signal-processing algorithms. In general, about eighty percent of the run-time in DSP algorithms is spent on twenty percent of the application.

Consider the algorithm presented in Figure 1-2, which shows a line-doubling algorithm. An input line is shown that has only four samples, which arrive at the input with a period of two clock cycles per sample. This input line is subject to a certain processing in



```
for i=0 to 3 begin
  o[i]   = input();
  o[i+4] = proc(o[i]);
end
for j=0 to 7 begin
  output(o[j]);
end
```

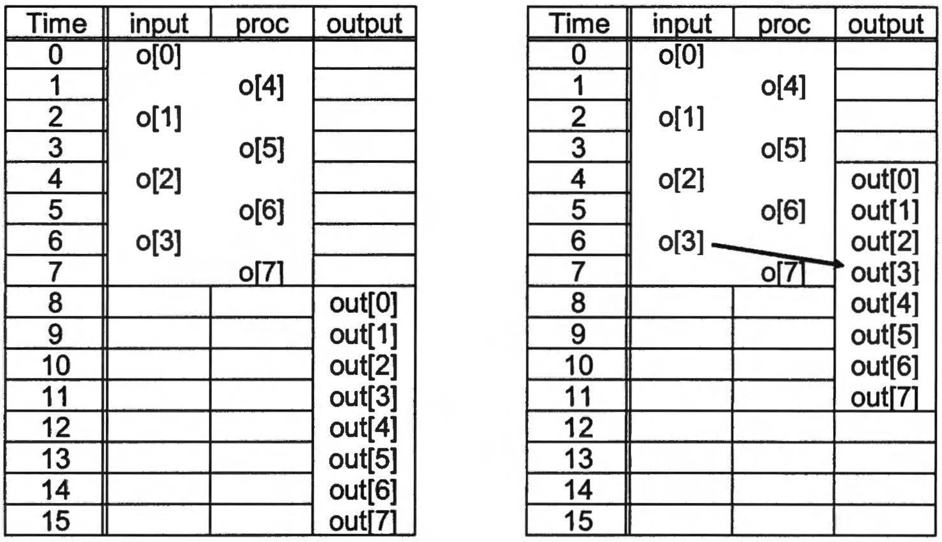
Figure 1-2: An algorithm with two loops

the proc node of which the internal details are of no importance for this discussion. After that, the processed samples are sent to the output, where they follow a copy of the original input samples. Hence, the output period is half of the input period (or, in other words, the rate has doubled). The same figure also shows a possible specification in textual format. This is only one of the possible alternatives. However, all alternatives have one thing in common, namely, the use of loops.

With this specification, a number of implementations are possible. Two of them are shown in Figure 1-3. A processor with a single controller can only implement the schedule as depicted in Figure 1-3a: a strictly sequential execution of the loops. However, this has two main disadvantages. First, the total execution time is the sum of the two loop executions, which implies a large program code. Next to that, the entire array o[i] produced by the first loop has to be stored temporarily before being consumed in the second loop.

An alternative schedule is given in Figure 1-3b. The loop executions partly overlap, thereby reducing the total execution time. This total execution time is now determined essentially by the individual loop executions and the data dependencies between the loops. In the example of Figure 1-2, the data dependence between o[3] and output(o[3]) restricts the earliest start time for the output loop, as the arrow in Figure 1-3b illustrates. Furthermore, in this case, only half of the array o[i] has to be stored. This schedule, however, can only be implemented with at least two threads of control. That is, at least two controllers: one for controlling the input and proc resources and one for controlling the output resource. In general, many applications allow a certain degree of loop overlapping, making it worthwhile to find a solution to allow this.

The two implementations presented in Figure 1-3 differ in the way that loops are dealt with. In Figure 1-3a, the loop hierarchy is preserved, i.e., the loops in the specification are interpreted procedurally. There is no overlap between the various executions of the same instance, or between executions of different instances. However, these loop



Time	input	proc	output
0	o[0]		
1		o[4]	
2	o[1]		
3		o[5]	
4	o[2]		
5		o[6]	
6	o[3]		
7		o[7]	
8			out[0]
9			out[1]
10			out[2]
11			out[3]
12			out[4]
13			out[5]
14			out[6]
15			out[7]

(a)

Time	input	proc	output
0	o[0]		
1		o[4]	
2	o[1]		
3		o[5]	
4	o[2]		out[0]
5		o[6]	out[1]
6	o[3]		out[2]
7		o[7]	out[3]
8			out[4]
9			out[5]
10			out[6]
11			out[7]
12			
13			
14			
15			

(b)

Figure 1-3: Possible implementations: (a) regular, single threaded schedule; (b) two-threaded schedule with overlapping loop executions

boundaries are artificial constraints. Figure 1-3b, therefore, does not consider them: only the real constraints, which are imposed by the data dependencies, have been taken into account. Although the difference may seem small, it is still important to note that the latency is smaller in Figure 1-3b. For high-throughput algorithms, the latency is often related to the amount of internal data that has to be stored. In other words, if we would repeat the same example for a complete video line consisting of, for example, 864 samples, we would have to store a complete line in the situation presented in Figure 1-3a, but only half a line (432 samples) for the situation in Figure 1-3b. Obviously, a procedural interpretation preserving the loop hierarchy is insufficient for efficient implementations of high-throughput algorithms.

There are multiple approaches to map the two-threaded schedule as presented in Figure 1-3b onto hardware. The most straightforward approach is using a multi-processor architecture, where each of the processors executes a single loop. In that way, the loop executions can be overlapped in time (for exploiting the available ILP and reducing the data buffering) and scalability in performance is enhanced. However, implementation on a multi-processor architecture introduces the following problems:

- How to impose a static schedule for the processors;
- How to communicate efficiently between processors, with fixed timing delays;
- How to synchronize between processors.

These problems are now discussed in more detail.

How to impose a static schedule for the processors.
Static scheduling is chosen since it allows simulation and verification of the derived schedule at design-time. Furthermore, on an architectural level, it removes the need for dynamic scheduling hardware with its corresponding overhead. In case of a multiprocessor architecture, the individual processors are designed and scheduled individually. Program code is generated for each processor separately, since there are no commercial tools available (yet) with an instruction-scheduling scope spanning multiple controllers. The practical way to verify whether all timing constraints are met is to simulate the collection of processors and their interactions. It would be much more convenient to derive a combined schedule for the collection of processors in one tool. In that way, the timing constraints can be imposed and guaranteed without extensive simulation efforts.

How to communicate efficiently between processors.
One of the reasons why simulation is necessary for verifying real-time constraints in a multi-processor system is the uncertainty caused by communicating over a shared bus. Since other processors can also access this shared bus, communication between processor A and B may be delayed because another processor occupies the bus. In other words, the communication delay is not fixed. Furthermore, an explicit communication protocol with requests and acknowledgements for accesses on the bus is required. This is necessary to cope with the uncertainty in the system, but takes time and energy. Moreover, communicated data often has to be buffered when waiting for the bus. Therefore, inter-process communication is not efficient in terms of latency and buffer space.

How to synchronize.
Some processors may share certain data from a memory. This data may be updated periodically. In order for a processor to fetch the most recent data, it has to synchronize with the processor that wrote the data in the memory. For this, explicit synchronization protocols are used to enforce a certain order on tasks executed on different processors. Explicit synchronization takes time and implies that processors have to wait, resulting in a loss of performance.

The synchronization problem could be solved by using an (stripped down) operating system. However, in case of using an operating system, there will still be a communication problem, since the operating system will not be able to guarantee fixed timing delays for communication over shared busses.

An alternative to the multi-processor architecture is a VLIW architecture, where multiple controllers control the single data path that is available, as presented in Figure 1-4. A potential problem in this architecture is the sharing of data path elements among different controllers, since the situation may arise where the two controllers control a single resource simultaneously, resulting in a conflict. However, this can be avoided by generating a static combined schedule (and program code), without resources conflicts, for both controllers. In the hardware, the control signals from both controllers for a certain data path resource are combined using an "OR" gate, in order to have a single control signal for that particular resource. Since the compile-time scheduler prevents the simultaneous activity of both control signals for a certain resource, the "OR" gates always generate a valid control signal.

The presented architecture solves the previously mentioned problems for a multiprocessor architecture in the following way:

- *How to impose a combined static schedule*: the schedule is implemented as a composition of multiple schedules;
- *How to communicate efficiently between processors*: communication takes place via the register files that are shared among the controllers;
- *How to synchronize between processors*: the corresponding program counters run in sync on the same hardware clock.

Furthermore, it is a solution that is scalable in performance: there are no central resources (like for example a central instruction memory), so the resources controlled by a certain controller can be kept close to this controller. In addition, the controller itself can be kept small. Finally, executing different loops on different controllers guarantees the overlapping execution of loops.

The goal of this master's project is to find a method for the generation of a static schedule for a VLIW DSP data path with multiple controllers. Since no design tools with an instruction scheduling scope spanning multiple controllers are available yet, this method has to use currently available design tools. The generated static schedule will be a composition of multiple schedules, one for each controller, which allow multiple loops to be executed in parallel on the single VLIW data path. To allow this, the schedules should be generated in such a way, that all available hardware resources are shared among the controllers without any resource conflicts. Having found a method for generating such schedules, the (dis-) advantages of having multiple micro-coded hardware controllers in a VLIW data path will be shown, by mapping an application on different architectural approaches (e.g., single processor with or without loop transformation).

1.2. Thesis overview

The remainder of this report is organized as follows. Since the architecture proposed in the previous section belongs to the class of multi-threaded processor architectures, we present in Chapter 2 an overview of the research that has been performed in the past concerning multi-threaded processor architectures. In addition, we show how these architectural approaches differ from the VLIW data path with multiple controllers that we propose.

In Chapter 3, we present an overview of the main concepts of two design tools that are used within Philips Research for the design of VLIW architectures, as well as a user perspective on the similarities and differences between them.

Chapter 4 consists of the main part of this report: it presents a method for the use of currently available design tools for single-controller VLIW architectures to generate schedules for multiple controllers, which allow the overlapping execution of loops. Furthermore, this chapter contains the results of a simple case study, which has been performed to show the feasibility of the presented method. Based on this case study we present some (dis-) advantages of having multiple controllers in a single VLIW data path.

Chapter 5 presents the results of applying the method presented in Chapter 4 to a more realistic case study: the mapping of the Viterbi decoding algorithm on a VLIW data path with multiple controllers. Again, the (dis-) advantages of having multiple controllers for this case study are shown.

Finally, Chapter 6 presents the main conclusions for the present method, as well as some recommendations for future work with respect to the generation of schedules for a VLIW data path with multiple controllers.

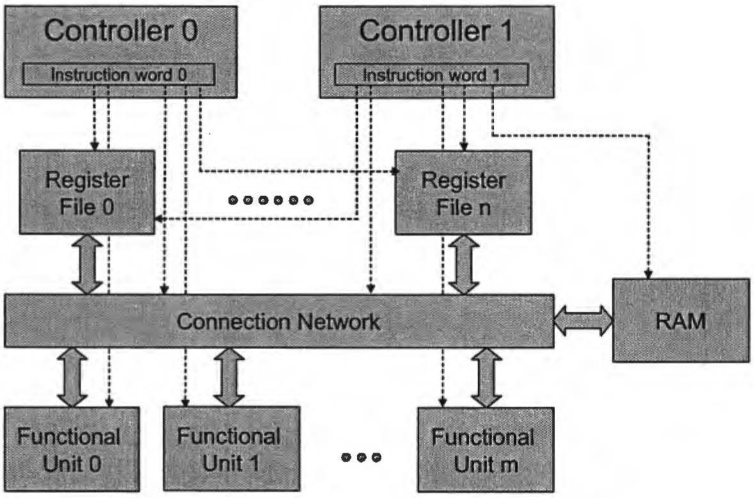


Figure 1-4: A VLIW data path with multiple controllers

2. Multi-Threaded Processor Architectures: an Overview

The target architecture as described in the introduction contains a data path that is controlled by multiple controllers. Such an architecture belongs to the class of multi-threading architectures: an architecture that is able to pursue two or more threads of control in parallel. For that reason, we present in this chapter an overview of the research that has been performed in the past concerning multi-threaded processor architectures.

We start this chapter with a short overview of the history of multi-threading architectures. Next to that, in Section 2.2, we present the main architectural approaches, which are of interest for our work. Finally, in Section 2.3, we describe the projects that are closest to our approach and mention the main differences with those projects.

2.1. Origins of multi-threading

The continuing effort to exploit the further advance of VLSI technology enables designers to put more transistors on a chip and to increase the clock speed of on-chip operations each year. Due to this increasing technology, we are now able to incorporate several processors into complex microprocessor designs on a single chip. A major step in this evolution has been the introduction of the *reduced instruction set computer* (RISC) architecture. This RISC concept emerged from the desire for a simple architecture, which enables a clear division of function between hardware and software.

The main goal of the RISC architecture has been to develop processor designs that come close to initiating one instruction on each clock cycle of the machine. Several products have met this goal, e.g., the MIPS and the Sparc processors ([Hen96]). Two architectural features have made that achievement possible: instruction pipelining and cache memories. Pipelined instruction execution allows subsequent instructions to begin execution before previously issued instructions have finished execution, as illustrated in Figure 2-1. This figure shows how instruction processing would proceed sequentially

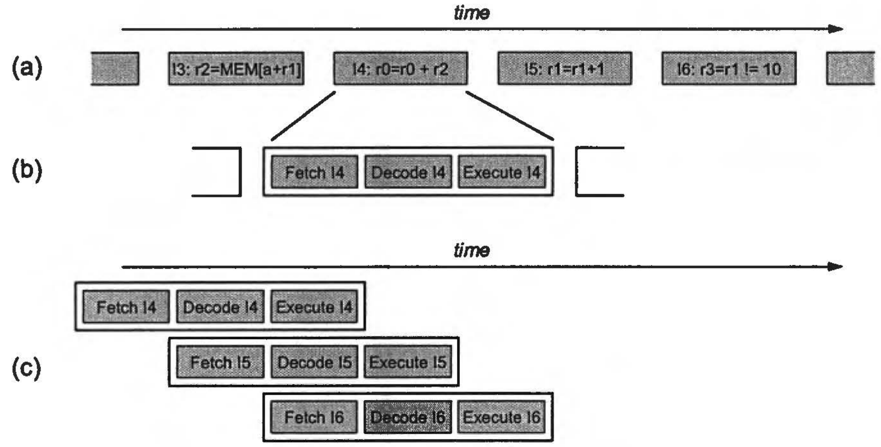


Figure 2-1: (a) Sequential instruction processing; (b) Steps required for executing a single instruction; (c) Execution with pipelining: the various executions of different instructions overlap

(Figure 2-1a). As shown in Figure 2-1b, processing a single instruction involves a number of micro-operations, for example fetch, decode and execute. In the case of instruction pipelining, the micro-operations required to execute different instructions overlap, as shown in Figure 2-1c. Instruction processing now proceeds in a number of steps, or pipeline stages, in which the various micro-operations are executed, resulting in a shorter instruction processing latency.

Cache memories allow instruction execution to continue, in most cases, without waiting the full access time of the main memory. However, if certain data is not available in the cache memory, a cache miss will occur. This cache miss causes the pipeline to stall, waiting on the data to become available from the main memory, thereby decreasing the performance of the processor. The latency becomes a problem if the processor spends a large fraction of its time sitting idle and waiting for the access to the main memory to complete, because it results in a loss of performance.

In order to increase the performance of a RISC processor and further decrease the *clock cycles per instruction* (CPI) of the processor, the superscalar architecture has been introduced. A processor based on a superscalar architecture model is capable of issuing multiple instructions from a single instruction sequence during a single clock cycle ([Mos01]), exploiting the *instruction-level parallelism* (ILP) available in a program (see Figure 2-2). In such an architecture, enough hardware resources are provided to allow multiple, independent instructions to be in the same stage of processing simultaneously.

Nevertheless, instruction throughput in both RISC and superscalar architectures falls well short of ideal ([Cha99]). Cache misses, partial issue cycles, branch mispredictions and insufficient ILP in the program ([Wal91]) are among the factors preventing full utilization of the available issue bandwidth. A solution for increasing the instruction throughput is *multi-threading*: the processing of instructions from several different threads in parallel. The co-existence of multiple active threads allows a multi-threading processor to improve the overall instruction throughput by taking advantage of *thread-level parallelism* (TLP): instructions from different threads are independent of one another and thus can be executed in parallel, leading to greater functional unit utilization and greater tolerance of execution latencies. According to [Den94], a multi-threaded architecture differs from a single-threaded architecture in the sense that there may be several enabled instructions from different threads, which are all candidates for execution. Similar to the single-threaded architecture, the state of a multi-threaded architecture consists of a memory state (program memory, data memory and stack) and a processor

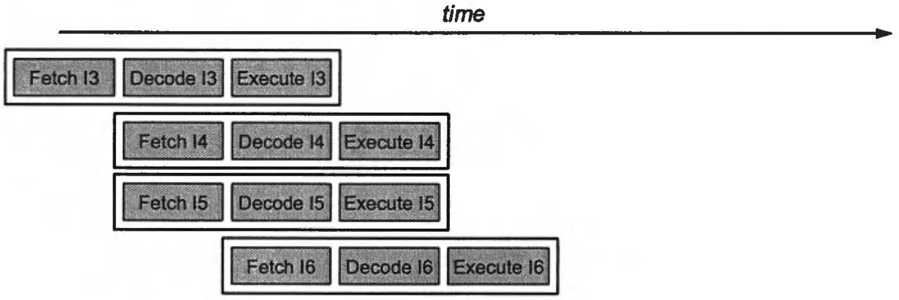


Figure 2-2: Superscalar execution of instructions (using the sequential instruction stream of Figure 2-1a as example). I4 and I5 are executed in parallel. I4 has to wait for I3, since it requires the value in r2. I6 has to wait for I5, as it needs the new value of r1 produced by I5

state (program counter, stack pointer and register context). However, in case of a multi-threaded architecture, the processor state consists of a collection of activity specifiers (program counters and stack pointers) and a collection of register contexts, whereas for a single-threaded architecture this state consists of a single activity specifier and a single register context.

Another root of multi-threading comes from dataflow architectures. The dataflow architecture originated from the concept of using dataflow program graphs as a machine-level program representation ([Ian94]). Viewed from a dataflow perspective, a single-threaded architecture is characterized by the computation conceptually moving forward one step at a time through a sequence of states, each step corresponding to the execution of one enabled instruction. However, direct implementation of processors based on a dataflow model is a difficult task. For this reason, the conventional (RISC-like) control-flow thread execution was incorporated into the dataflow approach, which resulted in a multi-threaded architecture ([Ian94]).

After this short introduction into the history of multi-threaded architectures, we discuss in the next section several common approaches for implementation of multi-threading techniques.

2.2. Multi-threading approaches

According to [Ung02], multi-threading can be applied either to increase the performance of a single program thread by implicitly utilizing parallelism, which is more coarse-grained than ILP (so-called *implicit multi-threading*), or to increase the performance of a multi-programming or multi-threaded workload (so-called *explicit multi-threading*), depending on the architectural approach chosen.

The term implicit multi-threaded architecture refers to any architecture that is able to execute several threads, which are dynamically generated from a single-threaded program, concurrently. Examples of such architectural approaches are the multi-scalar processor ([Fra93]), the trace processor ([Rot97]) and the speculative multi-threaded processor ([Mar98]). In these examples, a single processing unit with a single or multiple-issue pipeline that is able to process instructions of different threads concurrently may characterize the multi-threaded processor. As a result, some of these approaches may rather be viewed as very closely coupled *chip multi-processors* (CMPs), which integrate two or more complete processors on a single chip, because multiple groups of subordinate processing units each execute different threads under the control of a single sequencer unit.

An explicit multi-threaded architecture interleaves the instructions of different (user-defined) threads of control in the same pipeline. Therefore, multiple program counters are available in the instruction fetch unit and the multiple contexts are often stored in different register sets on the chip. The available execution units are multiplexed between the thread contexts, which are loaded in the register sets. These types of architectures tolerate memory latencies that arise in the computation of a single instruction stream by overlapping the long-latency operations of one thread with the execution of operations from other threads.

Since we aim at an architecture with multiple program counters and multiple register contexts, we restrict ourselves to the explicit multi-threaded approaches in the following discussion.

According to [Ung02], explicit multi-threaded processors fall into two categories, depending on whether they issue in a given cycle instructions from a single thread only or from multiple threads.

If in a given cycle instructions can be issued from a single thread only, the following two principal techniques of explicit multi-threading are used (see Figure 2-3):

- *Interleaved multi-threading:* at each processor cycle, an instruction of a different thread is fetched and fed into the execution pipeline;
- *Blocked multi-threading*: the instructions of a thread are executed successively until an event occurs that (possibly) causes latency. This event induces a context switch to another thread.

When instructions can be issued from multiple threads in a given cycle, the following technique can be used:

- *Simultaneous multi-threading*: the superscalar instruction issue approach is combined with the multiple-context approach. Instructions are issued simultaneously from multiple threads to the execution units of a superscalar processor.

Before we present the different multi-threading techniques in more detail, we briefly introduce the main principles of architectural approaches that exploit instruction-level parallelism and thread-level parallelism.

Figure 2-4a until Figure 2-4c demonstrate the different approaches possible with scalar processors (i.e., single-issue processors like a RISC processor): single-threaded (Figure 2-4a), with interleaved multi-threading (Figure 2-4b) and with blocked multi-threading (Figure 2-4c).

Figure 2-4d until Figure 2-4i demonstrate the different approaches possible with four-issue processors: single-threaded superscalar (Figure 2-4d), single-threaded VLIW (Figure 2-4g), superscalar with interleaved multi-threading (Figure 2-4e), superscalar with blocked multi-threading (Figure 2-4f), VLIW with interleaved multi-threading (Figure 2-4h) and VLIW with blocked multi-threading (Figure 2-4i).

The number of instructions that might be processed while the pipeline is interlocked (so-called *opportunity cost* ([Ung02])) can be easily determined in a single-threaded superscalar processor as the number of empty issue slots (Figure 2-4d). It consists of *horizontal losses* (the number of empty places in a partially filled issue cycle) and the even more harmful *vertical losses* (cycles where no instructions at all can be issued). In a VLIW processor (Figure 2-4g), horizontal losses appear as *NOPs* (N; no operations). The opportunity cost of a VLIW architecture is about the same as a single-threaded

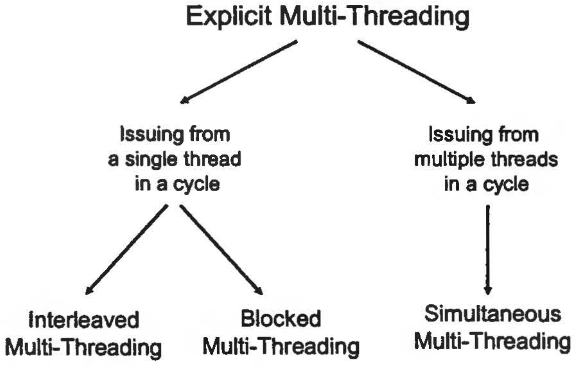

Explicit Multi-Threading
Issuing from a single thread in a cycle
Issuing from multiple threads in a cycle
Interleaved Multi-Threading
Blocked Multi-Threading
Simultaneous Multi-Threading

Figure 2-3: Explicit multi-threading approaches

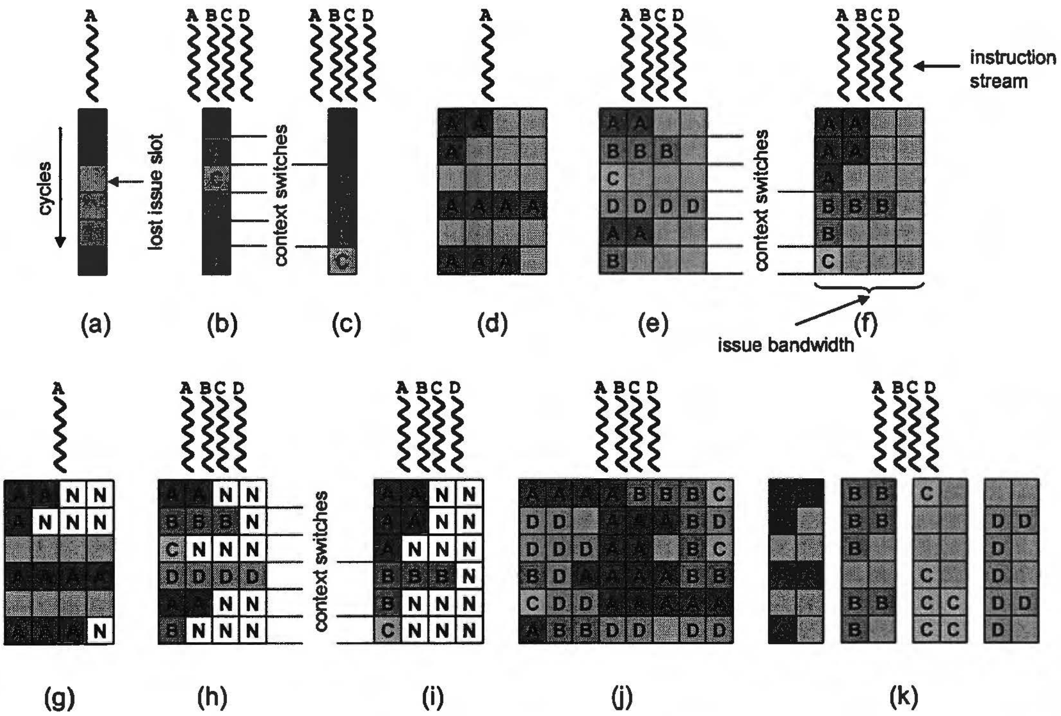


Figure 2-4: Different approaches possible with scalar processors: (a) single-threaded scalar; (b) interleaved multi-threading scalar; (c) blocked multi-threading scalar. Different approaches possible with multiple-issue processors: (d) superscalar; (e) interleaved multi-threading superscalar; (f) blocked multi-threading superscalar; (g) VLIW; (h) interleaved multi-threading VLIW; (i) blocked multi-threading VLIW; Other approaches: (j) simultaneous multi-threading; (k) chip multi-processor. Each row represents the issue slots for a single execution cycle

superscalar processor. The interleaved multi-threaded superscalar (Figure 2-4e) and the interleaved multi-threaded VLIW (Figure 2-4h) are able to fill the vertical losses of the single-threaded models with instructions of other threads, but are not able to fill the horizontal losses. Other design possibilities, like the blocked multi-threading superscalar (Figure 2-4f) and the blocked multi-threading VLIW (Figure 2-4i) models, would fill several succeeding cycles with instructions of the same thread before performing a context switch. This switching event is more difficult to implement and as a result, a context-switching overhead of one to several cycles might arise.

Figure 2-4j and Figure 2-4k demonstrate a four-threaded eight-issue *simultaneously multi-threading* (SMT) processor and a *chip multi-processor* (CMP) with four two-issue processors, respectively. The processor model in Figure 2-4j exploits ILP by selecting instructions from any thread (four in this case) that can potentially issue. If one thread has high ILP, it may fill all horizontal slots with instructions of this thread, depending on the issue strategy of the SMT processor. If multiple threads each have low ILP, instructions of several threads can be issued and executed simultaneously. In the CMP with four two-issue CPUs on a single chip that is represented in Figure 2-4k, each CPU is assigned a thread from which it can issue up to two instructions each cycle. Thus, each CPU has the same opportunity cost as in a two-issue superscalar model. The CMP is not able to hide latencies by issuing instructions of other threads. However, a CMP of four two-issue processor will reach a higher utilization than an eight-issue superscalar processor ([Egg97]), because the horizontal losses will be smaller for two-issue than for high-bandwidth superscalars.

2.2.1. Interleaved multi-threading

In the interleaved multi-threading model (also called *fine-grain* multi-threading ([Ung02])), the processor switches to a different thread after each instruction fetch. In principle, an instruction of a thread is fed into the pipeline after the retirement of the previous instruction of that thread. This eliminates control and data dependencies between instructions in the pipeline. Therefore, pipeline hazards cannot arise and the processor pipeline can easily be built without the necessity of complex forwarding paths. This leads to a fast pipeline: no hardware interlocking or data-forwarding is necessary. As a result, the context-switching overhead is zero cycles. Memory access latency is tolerated, since a thread is not scheduled until the memory transaction has completed. This model requires at least as many threads as pipeline stages in the processor, in order to prevent the occurrence of vertical losses. Interleaving the instructions from many threads on a cycle-by-cycle base limits the processing power accessible to a single thread, thereby degrading the single-thread performance. There are two possibilities to overcome this:

- *Dependence look-ahead technique*: this technique adds several bits to each instruction format. The compiler uses the additional bits to state the number of instructions that follow in program order and are not data- or control-dependent on the instruction being executed. This allows the instruction scheduler of the interleaved multi-threading processor to feed non-data- or control-dependent instructions of the same thread into the pipeline successively;
- *Interleaving technique*: this technique adds caching and full pipeline interlocks to the interleaved multi-threading approach. Contexts are interleaved on a cycle-by-cycle basis, yet a single-thread context is also efficiently supported.

The most well known example of interleaved multi-threaded processors is used in the Heterogeneous Element Processor (HEP, [Smi81]), the first commercial computer system employing a multi-threaded architecture.

2.2.2. Blocked multi-threading

The blocked multi-threading approach (also called *coarse-grain* multi-threading ([Ung02])) executes a single thread until it reaches a situation that triggers a context switch. Usually, such a situation arises when the instruction processing reaches a long-latency operation or a situation where latency may arise (e.g., a branch or an access to the cache memory). Compared to the interleaved multi-threading technique, a smaller number of threads is needed and a single thread can execute at full speed until the next context switch arises. The single-thread performance on a blocked multi-threading processor is similar to the performance of a comparable processor without multi-threading: if a single threads runs on a blocked multi-threading processor, no context switches occur and the processor performs just like a processor without multi-threading.

According to [Ung02], the blocked multi-threading technique can be classified into static and dynamic models, depending on the type of event that triggers the context switch. In the static model, a context switch occurs each time the same instruction is executed in the instruction stream. The compiler encodes this context switch and therefore context-switching can be triggered already in the fetch stage of the pipeline, resulting in a lower context-switching overhead (close to zero cycles). On the other hand, in the dynamic model, a dynamic event triggers the context switch. Examples of such dynamic events are cache misses, specific signals (interrupts) or an attempt to use the still missing value of a load operation. In general, all the executions between the fetch stage and the stage that triggers the context switch are discarded, leading to a higher context-switching overhead than for static context-switching models.

Examples of processors that use the blocked multi-threading approach are the MIT Sparcle ([Aga93]) and the MSparc ([Mik96]).

2.2.3. Simultaneous multi-threading

Interleaved multi-threading and blocked multi-threading are techniques, which are most efficient when applied to scalar RISC or VLIW processors, since both types of processor are able to issue instructions from a single thread only every clock cycle. Combining the superscalar technique with multi-threading leads to a technique where several hardware contexts can be active simultaneously, competing each clock cycle for all available resources. This technique, called *simultaneous multi-threading* (SMT, see for example [Egg97] or [Tul95]), inherits from superscalars the ability to issue multiple instructions every clock cycle, and like multi-threaded processors it contains hardware resources for multiple contexts. The result is a processor that can issue multiple instructions from different threads each cycle, exploiting both ILP and TLP. In that way, instructions from several threads can fill unused issue slots within one clock cycle, as well as unused clock cycles that occur due to latencies. In principle, the full issue bandwidth of the processor can be utilized every clock cycle. The SMT fetch unit can take advantage of the inter-thread competition for instruction bandwidth in two ways: first, it can partition this bandwidth among the threads and fetch instructions from several threads each cycle. This increases the probability of fetching non-speculative instructions only. Second, the fetch unit can be selective about from which threads to fetch instructions. For example, it may only fetch instructions from those threads that will provide the most immediate performance benefit.

The SMT processor can be organized in two ways:

- *Resource sharing*: instructions of different threads share all resources like the fetch buffer, the physical registers for renaming of different register sets, the instruction window and the reorder buffer. Thus SMT adds minimal hardware functionality to conventional superscalar architectures; it consists of a fast single-threaded superscalar processor with multi-thread capability added on top;
- *Resource replication*: all internal buffers of a superscalar processor are replicated, such that each buffer is bound to a specific thread. Instruction fetch, decode, rename and retire units may be multiplexed between the threads or be duplicated themselves. The issue unit is able to issue instructions of different instruction windows to the execution units simultaneously. This form of organization adds more changes to the organization of superscalar processors, but leads to a natural partitioning of the instruction window as well as a simplification of the issue- and retire-stages.

Thread-level parallelism (TLP) can come from either multi-threaded, parallel programs or from multiple, independent programs in a multi-programming workload, while ILP is utilized from individual threads. Because an SMT processor simultaneously exploits both fine- and coarse-grained parallelism, it uses its resources more efficiently than single-threaded superscalar processors for multi-threaded workloads and thus achieves a better instruction throughput.

Chip multi-processors (CMP) represent a competing approach to SMT. A CMP integrates two or more complete (super-) scalar processors on a single chip. In that way, every unit of a processor is duplicated and used independently of its copies on the chip. Because of that, a multi-processor is easier to implement and better scalable to a large number of threads. However, only the SMT approach has the ability to hide latencies.

2.3. Related work

The architectural approaches presented in the previous section all apply to general-purpose processors. Obviously, this has consequences for the architectural design, since a general-purpose architecture needs to be flexible. This flexibility reflects itself for example in the presence of cache memories, which are used to speedup the access to variables from the main (shared) memory. Another example of a typical general-purpose feature is out-of-order execution, which relaxes the ordering constraint on the execution of instructions, thereby increasing the concurrency ([Mos01]). However, the most important property of a general-purpose architecture is the presence of a resource that is central to all other resources present in the architecture: the instruction memory.

In an *application domain specific processor* (ADSP), which is the type of processor we aim at in this project, it is possible to remove much of the overhead of general-purpose architectures, since often the behaviour of the application is well known at compile-time. Because of this, for example, there is a strong control over the memory architecture, which allows us to replace the caches and (part of) the main shared memory with small, distributed local memories. Furthermore, for the type of architecture we have in mind, it is possible to remove the bottleneck of a central instruction memory, since in our architecture each of the controllers will have its own independent instruction memory.

Looking at the architectural approaches presented in the previous section, the simultaneous multi-threading approach –more specifically, the SMT Multimedia Processor– is closest to the type of architecture we aim at. The reason for this is the fact that we aim at an architecture, which is able to pursue multiple threads in parallel. Furthermore, our architecture contains *application specific units* (ASUs), just like the SMT Multimedia Processor.

The SMT Multimedia Processor model (see Figure 2-5, [Oeh99]) features single or multiple *instruction fetch* (IF) and *-decode* (ID) units, a single *rename/issue* (RI) unit, multiple, decoupled reservation stations and multiple execution units. In particular, these execution units consist of several combined integer/multi-media units, a complex integer/multi-media unit, a branch unit, separate local and global load/store units, a single *retirement* (RT) and *write-back* (WB) unit, rename registers, a *branch target address cache* (BTAC) and separate instruction- and data-caches (I-cache and D-cache, respectively) that are shared by all active threads. Thread-specific instruction buffers (between IF and ID), issue buffers (between ID and RI) and reorder buffers (in front of RT) are employed in the pipeline. Each thread is executed in a separate register set.

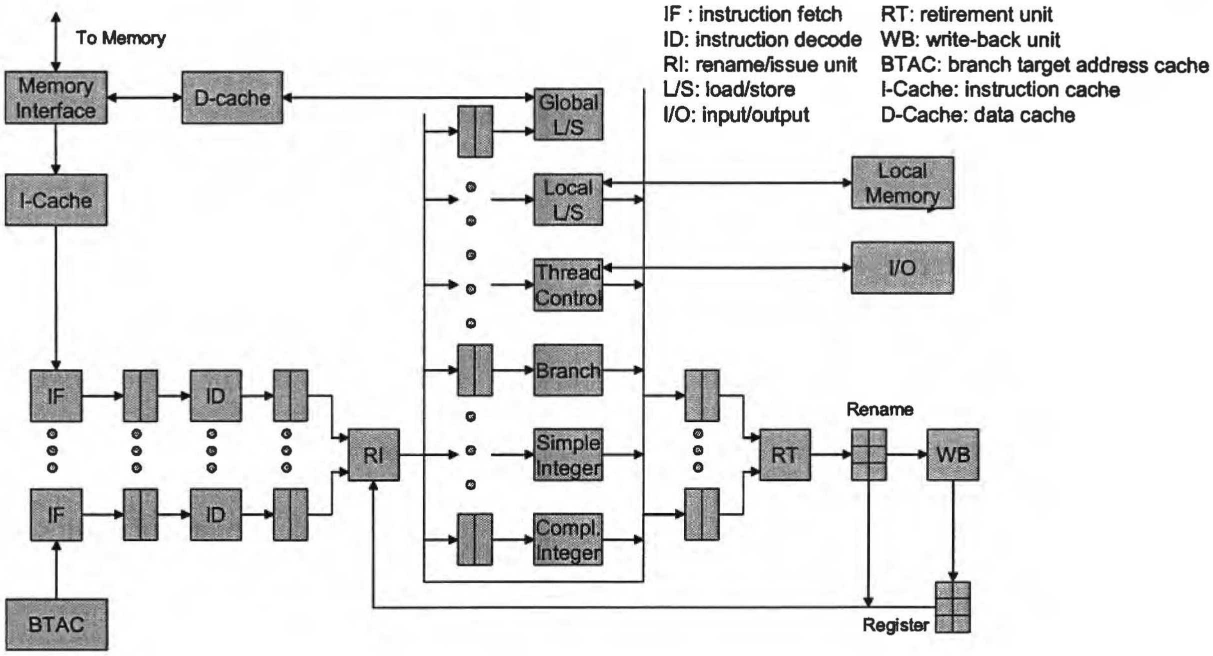


Figure 2-5: The SMT Multimedia Processor model

However, there is no fixed allocation between threads and (execution) units. The pipeline performs in-order instruction fetch, decode and rename/issue to reservation stations, out-of-order dispatch from the reservation stations to the execution units, out-of-order execution and, finally, in-order retirement and write-back.

The rename/issue stage simultaneously selects instructions from all issue buffers up to its maximum issue bandwidth (which is a feature of the SMT approach). The integer units are enhanced by multi-media processing capabilities. A thread control unit is employed for thread start, stop and synchronization operations, as well as for I/O operations. Furthermore, a local RAM memory is included, which is accessed by the local load/store unit.

The main difference with the architecture presented in Figure 2-5 and the architecture presented in the introduction is that our architecture contains multiple instruction memories (instead of one central instruction cache). Each of those instruction memories holds the instructions for a single instruction stream and controls part of the execution units, as illustrated in Figure 2-6. Because of these multiple instruction memories, the

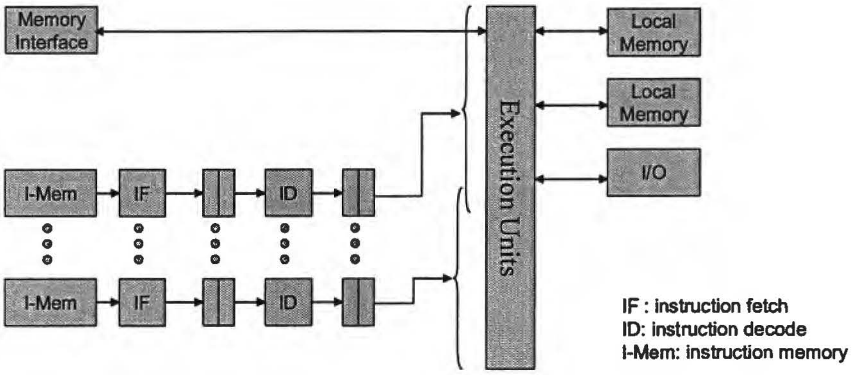


Figure 2-6: Schematic view of differences with SMT Multimedia Processor

rename/issue unit (RI) of the SMT Multimedia Processor becomes superfluous. Among the execution units are also ASUs, since we target at an ADSP architecture. Two other differences are the absence of a data-cache and the presence of multiple (small) distributed local memories in our architecture (instead of one lärge shared memory).

The Phideo tools ([Ver95]) create another architecture, which has a close resemblance to the target architecture as presented in the introduction. The Phideo architecture is presented in Figure 2-7. It consists of a number of *processing units* (PU), a number of *memories* (M), *address generators* (AG) for those memories and a controller. The memories can be static or dynamic RAMs, register files, separate registers or flip-flops. The role of the memories is to take care of the data transport between the processing units, which produce and consume the data. Note, that the number of memories may differ from the number of processing units.

Memory places can be reused between several streams to reach a more efficient implementation when the schedule allows this. A routing network is needed for the data transport between processing units and memories. A similar cooperation exists between address generators and memories. It is the role of the address generators to generate the right addresses for the memories at every point in time. Obviously, the number of address generators may differ from the number of memories. A second routing network takes care of the transport of the generated addresses.

It is the task of the controller to generate the necessary control signals, such as the selection of the correct function on a processing unit, the next address signals for the address generators and the control for both routing networks.

The similarity between the architecture generated by the Phideo tools and the architecture presented in Section 1.1 lies in the fact that they both consist of multiple application specific function units and local, distributed memories, to enable multiple threads to be executed in parallel. The main difference between the Phideo architecture and the one described in the introduction is the fact that the controller of the Phideo architecture is implemented based on hardware counters instead of program counter addressed instruction words and that the Phideo architecture is not programmable. Furthermore, the

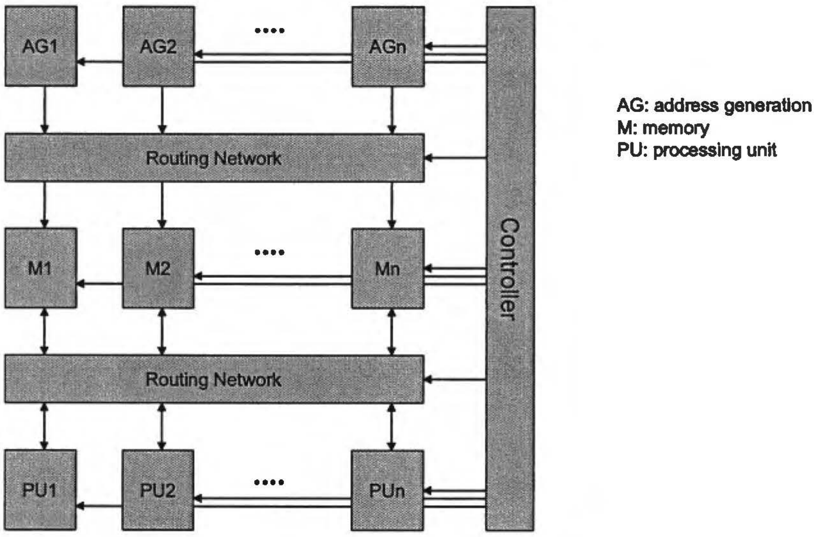


Figure 2-7: The Phideo architecture

Phideo architecture has one central controller (which is synthesized in a distributed manner), whereas we aim at an architecture with multiple, independent controllers. Finally, in the Phideo architecture, as shown in Figure 2-7, address generators generate addresses for all memories, even for the register files. This addressing can be useful, if, for example, there is a need to access a register file in a cyclic manner. However, often this address generation for register files adds extra and unnecessary hardware complexity to an architecture, since it is for example also possible to address the register files using absolute addresses in the machine code. The type of architecture we aim at is an architecture without (hardware) address generation for the register files.

In [Jay02], a clustered L0 or loop buffer organisation for a VLIW processor is proposed, in order to reduce the energy consumption in the instruction memory. The essentials of this clustered L0 buffer organisation are illustrated in Figure 2-8. The loop buffers are partitioned and functional units are logically grouped in the VLIW data path to form instruction clusters. In each of the clusters, the buffers store only the operations of a certain loop, destined to the functional units in that cluster. Typically, these buffers are used to store those operations that are executed often (such as loop operations), in order to reduce the energy consumption of the instruction memory.

By default, the L0 buffers are not accessed during the normal phase of execution, and a program executes via the normal (L1) instruction memory. Parts of the program that are to be fetched from L0 buffers should be marked explicitly either by the programmer or the compiler. For this, a special instruction is used. Once this instruction is encountered, the necessary instructions are pre-fetched and distributed over the different L0 partitions by the instruction dispatcher. Since the distributed organisation allows the restriction of accesses to partitions that are not active in a certain instruction cycle, the local controller (*index and translation control*, ITC) of each cluster is provided with an activation trace. While the operations of each instruction in the loop are pre-fetched and distributed among the partitions, a zero or one is stored in the activation trace register indicating that a partition is inactive or active, respectively, for the corresponding instruction cycle. During the execution of the operations in the loop buffers, the *loop buffer control* (LBC)

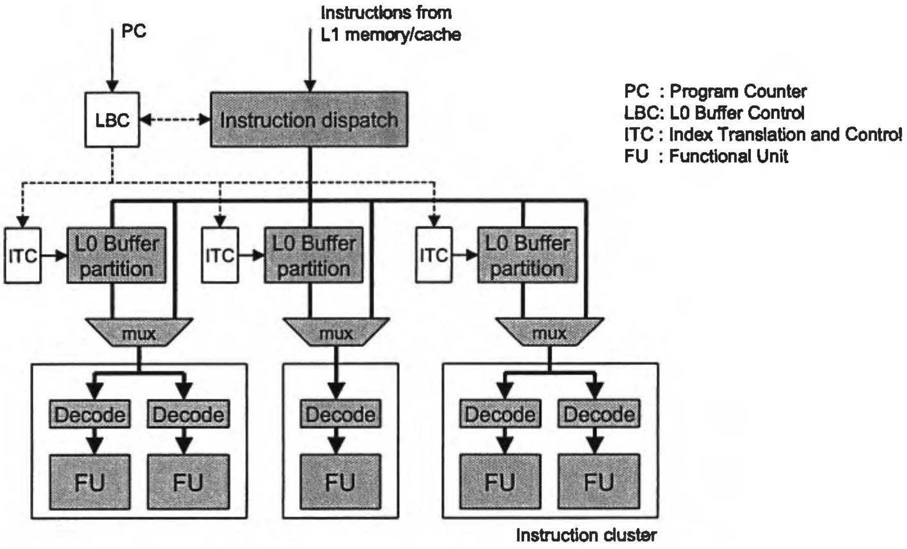


Figure 2-8: The clustered L0 buffer organisation, according to [Jay02]

unit controls the ITCs of each buffer and selects the appropriate inputs of each of the multiplexers, such that the operations are indeed fetched from the L0 buffers instead of from the L1 instruction memory.

Currently, this clustered buffer organisation is being extended to support the execution of multiple loops in parallel ([Jay03]). For this, each of the loops will be mapped and executed on a certain L0 cluster, which is possible since each cluster has its own fetch mechanism (program counter). If no data dependencies between the instructions of one loop to another loop exist, then the execution of each loop can be independent of the others and will be synchronised after the execution of the loops. However, if data dependencies between loops are present, the compiler will have to take care of proper synchronisation between the loops, as well as providing data moves from one cluster to another. This is different from the architecture we propose, where shared data will become available in shared register locations, without the need of explicit data moves or explicit synchronisation. Another difference with the architecture proposed in the introduction is that in the clustered buffer organisation, no sharing of data path resources between different loops is possible, since each of the loops is mapped onto a separate cluster.

3. Design Tools

As stated in the introduction, the main goal of this project is to use the infrastructure of design tools that are currently at the market to automatically generate schedules for multiple controllers. Within Philips Research, two design tools are available for the generation of VLIW architectures and schedules for them: A|RT Designer and Silicon Hive. In this chapter, we provide a user perspective on the similarities and differences between those two tools. Therefore, in the first two sections, a short overview of the architecture models and design flows used by both tools are presented. Finally, in Section 3.3, we discuss the main differences between the A|RT Designer and Silicon Hive tools.

3.1. A|RT Designer

Adelante Technologies ([Adelan]), which was founded in 2001 by Philips Semiconductors and Frontier Design, originally developed the A|RT Designer tool. In June 2003, Adelante sold its A|RT DSP coprocessor technology to ARM ([ARM]). The A|RT Designer tool assists the designer in the development of a processor or a processor-like architecture, customizing it for the algorithm that has to be executed on this architecture. The generated architectures consist of a set of data path resources, controlled by a "Very Long Instruction Word"-type (VLIW) controller. This results in a design that is configurable and scalable for parallelism as well as performance.

3.1.1. Architecture overview

The A|RT Designer tool maps a C algorithm into a *register-transfer level* (RTL) hardware description of an architecture ([ART]). This architecture can be described as a collection of register files, the paths between those register files and the resources along these paths that pass, modify, or store the data. The set of hardware execution units and busses together form the so-called data path. All the resources are managed by a controlling mechanism: the controller or the control path. The controller and the data path are the two main parts of the architecture. In Figure 3-1, an example of A|RT Designers' architecture model is shown.

The data path can best be described in terms of the paths it uses to transfer data from one register field to another and the resources used along those paths. The reason for this is that any register transfer starts by reading data from one or more fields in different register files and finishes by writing derived data into one or more fields in different register files. In between the reading and writing of data from and to register files are busses and functional resources that perform data transformations. Examples of functional resources are ALUs, ACUs, RAMs and ROMs.

Figure 3-2 shows the used data path model in more detail. This figure shows that the outputs of the register files feed the inputs of the resources directly. Furthermore, each of the resource inputs is assigned its own register file. The outputs of the resources are put on the busses either directly or via tri-state buffers, depending on whether multiple

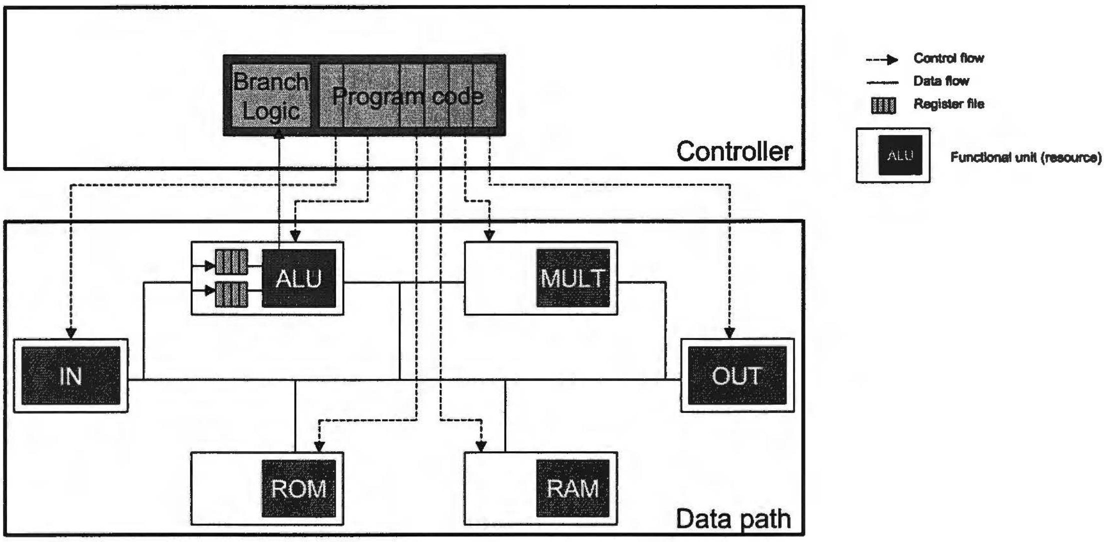


Figure 3-1: Example of A|RT Designer's architectural model ([ART])

resources are able to access the bus (a tri-state buffer is used to control which resource is allowed access to the bus). In turn, the busses will feed the register files. For that purpose, a layer of multiplexers connects the busses with the register files (unless only one bus has access to a certain register file).

Next to the data path of the architecture, is the control path with the controller. The control path is completely orthogonal to the data path. The controller controls all resources on the data path separately by a partition of the controller, as shown in Figure 3-1. This illustrates the VLIW aspect of the controller.

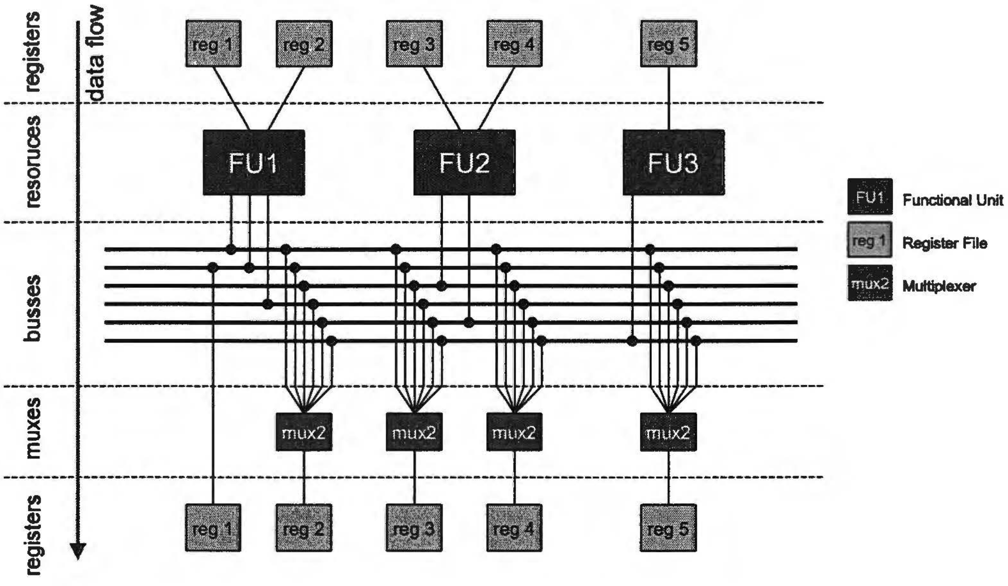


Figure 3-2: Data path of A|RT Designer's architecture in more detail ([ART])

3.1.2. Design flow overview

The main design steps within the A|RT Designer tool are shown in Figure 3-3. The tool is built around a central data structure that represents the design. During each step, this data structure is accessed and updated. At the end of all design phases, the complete design is available as a register transfer level description.

The design steps are executed in the order as indicated by the numbers in Figure 3-3, but the user can return to each of the previous steps at any time and influence each step by altering options or specifying pragmas. In the following, a brief description of each of the design steps is given, using the step numbers as present in the figure.

1. **Edit/Compile Source**. In this step, the algorithm to be implemented is specified and compiled. The specification is done in a subset of the C language. During the compile-step, the algorithm is converted into an internal representation. The Edit/Compile step uses methods of analysis to identify and represent the parallelism available in the source algorithm (data flow analysis). The information derived from this is exploited in the other design steps to achieve an optimal use of the architecture.
2. **Create architecture**. The architecture is specified by means of pragmas that instantiate core resources and a controller. Auxiliary resources, like register files, busses, tri-state buffers and multiplexers are introduced automatically. More pragmas can be inserted for fine-tuning the architecture to specific needs. Examples of such fine-tunings are merging register files, or restricting the bus network.
3. **Map to Architecture**. In this step, all operations in the C description are assigned to the data path resources specified in the Create Architecture step and translated into register transfers. Additionally, all variables used in the C description are assigned to the available memory types (register files, RAM, ROM). This yields a yet unordered RT level representation of the C source onto the target architecture. If a resource has multiple instances (e.g., two ALUs), the user can define which operations should be mapped onto what resource, using a pragma.

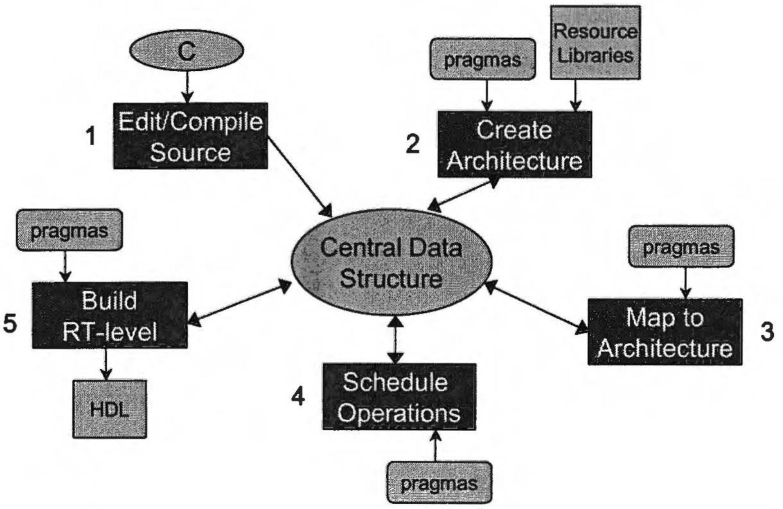


Figure 3-3: Internal design flow of A|RT Designer ([ART])

4. **Schedule Operations.** During scheduling, the register transfers are ordered along the time axis in as few clock cycles as possible, while taking into account clock-cycle timing constraints and hardware constraints. In addition, all variables are assigned to individual register fields in such a way that the register requirements are minimized. Scheduling is done by means of a basic scheduler, a set of optimisation techniques for improving the basic schedule and a set of pragmas to customize the schedule (e.g., to assign operations to a fixed potential or for specifying a fixed pipelining factor).
5. **Build RT Level.** This final step generates the complete design, data path and controller (including the program code), in a *hardware description language* (HDL, such as VHDL or Verilog).

3.2. Silicon Hive

Silicon Hive is a business entity created within the Philips Technology Incubator, an organization that creates new businesses based on technologies invented by Philips. Silicon Hive develops embedded reconfigurable architectures. Its reconfigurable computing technology differentiates itself from those of other players in the field by the fact that "it supports multiple styles of parallelism, allows for high-level programmability, allows cost-effective domain-specific solutions and supports the computational demands of the entire DSP spectrum, from very high and/or dynamic data-rates to relatively lower data-rates" ([SiHive]).

As in the previous section, we first show an overview of the target architecture of the Silicon Hive tool, followed by an overview of the design flow used in these tools.

3.2.1. Architecture overview

The basic component of architecture of Silicon Hive's architecture is the *Processing and Storage Element* (PSE), as shown in Figure 3-4 ([SiHive]). A PSE is a VLIW-like data path consisting of several *interconnect networks* (IN), one or more *issue slots* (IS) with associated *function units* (FU), distributed register files and, optionally, local memory storage (MEM).

A matrix of one or more (possibly different) PSEs, together with a VLIW-like *controller* (CTRL) and *configuration memory* (CONFIG MEM) make up a cell. A cell is a fully operational processor capable of computing complete algorithms. PSEs within a cell can communicate with each other via data *communication lines* (CL). Typically, one application function at a time is mapped onto the matrix of PSEs. The more PSEs are present, the more the function can be spread over the available PSEs, in a data-flow manner.

An array of one or more cells, connected together via a data-driven communication mechanism, forms a streaming array. The communication across cells takes place through blocking FIFOs accessed from *load/store* (LD/ST) units within the cells. Multiple functions can be concurrently mapped onto the streaming array, each one occupying a non-overlapping sub-set of the cells.

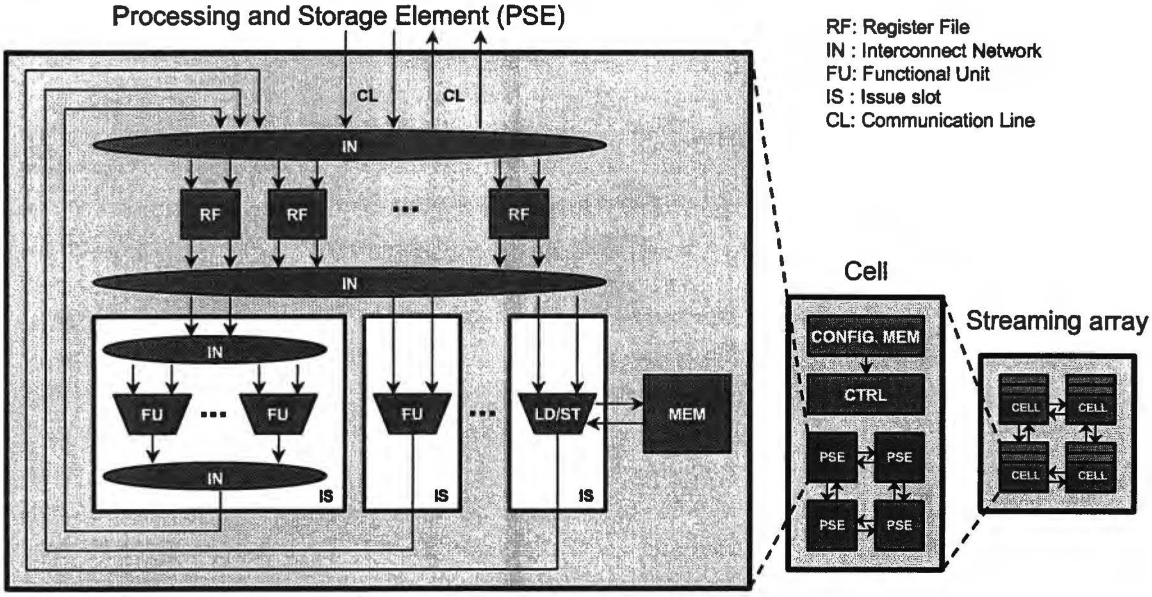


Figure 3-4: Silicon Hive's architecture design space overview ([SiHive])

3.2.2. Design flow overview

In Figure 3-5, an overview of the design flow used in the Silicon Hive tool is presented. A high-level C program and a machine description serve as input to the tools. The C program is, together with the architectural information present in the machine description, compiled into an intermediate representation (*hierarchical data flow C* (HDFC) description), consisting of machine operations and explicit control flow (do/while loops etc.). As presented in Section 4.1, this HDFC description consists of multiple basic blocks. Each of those basic blocks is extracted, one by one, and temporarily stored in a *data flow C* (DFC) description, starting with the deepest nested basic block. The Hive scheduler schedules the basic block, after which the scheduled basic block is inserted back into the original HDFC-description. This is repeated several times, until the complete HDFC-description is scheduled.

The scheduled HDFC-description, finally, is input to the assembler/linker, which generates the program code for the scheduled algorithm. This program code, together with a *hardware description language* (HDL) model of the machine description (generated by the processor generator) can be used for hardware synthesis and verification. Simulation is also possible during the other design steps, which allows for immediate verification of the results obtained during different design steps.

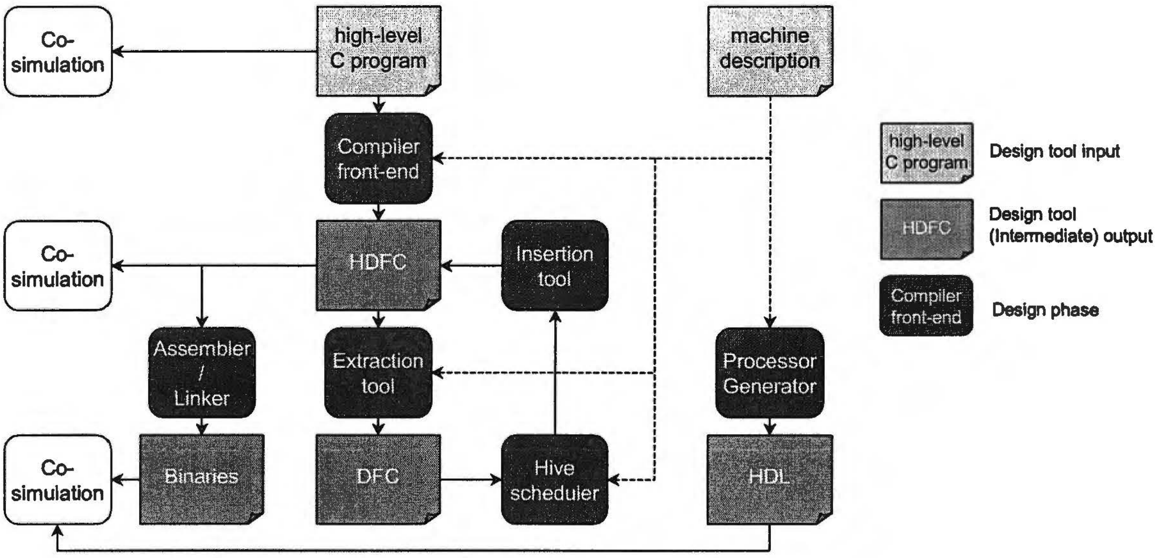


Figure 3-5: Silicon Hive's design flow ([SiHive])

3.3. A|RT Designer versus Silicon Hive: a user perspective

The main differences between A|RT Designer and the Silicon Hive tool are in the architecture model that both tools apply. In the following, these differences are discussed in more detail.

One of the major differences between the architectures of the A|RT Designer and Silicon Hive tools is in the way that both deal with immediate values for functional units. Immediate values are constant values that are not produced by some operation (thus not available in a register location) and are used in an operation directly (e.g., the addition of the value in register location R1 with the constant 4). In the A|RT Designer architecture, all immediate values are present in a so-called ROMCTRL memory. On the other hand, in the Silicon Hive architecture, the immediate values are encoded in the instruction bits that control a resource. The main advantage of the latter approach is that in every clock cycle the immediate values for all functional units can be loaded at once. In the A|RT approach, first all immediate values have to be loaded from the ROMCTRL memory, one by one, taking as many clock cycles as immediate values have to be loaded. This introduces quite an overhead when many immediate values are used in the application. On the other hand, the immediate encoding in the instructions results in longer instruction words, and thus in a larger program code size for the Silicon Hive architecture.

Opposite to the A|RT Designer tool, the Silicon Hive tool exposes all pipeline stages that are present in functional units to the compiler. This allows the elimination of complex bypass networks, forwarding and pipeline control logic. Furthermore, by splitting pipelined operations into separate operations for each pipeline stage, those operations can be handled as simple single cycle operations during assignment and scheduling.

One of the main features of A|RT Designer is the possibility for the user to design its own complex functional units (application specific units). With the Silicon Hive tool, such units can also be designed, with one major difference: the Silicon Hive tool does not allow functional units to have an internal state, as is the case for A|RT Designer. This is illustrated in Figure 3-6. That figure shows a multiply-accumulate (MAC) unit, which has the two variables *a* and *b* as its inputs, multiplies these two variables and adds the result to the accumulated result of previous multiplications. The accumulation register is initialised to zero before use, and after a (fixed) number of iterations, the accumulation result c is stored in a register. Figure 3-6a shows how the hardware of this functional unit can be designed using the A|RT Designer tool: the accumulation register is inside the unit, hidden from the user or compiler. In that way, the unit is in a certain state at any point in time. This state is determined by the content of the accumulation register. Figure 3-6b shows how this unit is designed using the Silicon Hive tools: the accumulation result is stored in a register field outside the functional unit, which is also an input to the functional unit. Now the state is explicitly available to, and thus controllable by, the compiler. Furthermore, using this approach the compiler can handle a lot of the complexity of a functional unit, instead of leaving it to the designer to cope with this complexity. On the other hand, this approach leads to a larger instruction width, since the MAC-unit designed with the Silicon Hive tool has three inputs and two outputs, which is one input and one output more than in the case of the A|RT design.

The lack of internal state for functional units also results in the absence of memories in the Silicon Hive architecture model; these are modelled as memory mapped I/Os (MMIO), using load/store units to access them. Using this approach, a cache for example has to be implemented as an application specific unit (ASU), with an MMIO-port to access a (external) cache memory.

A final distinct difference in architecture is the use of issue slots (clusters or groups of functional units) in the Silicon Hive architecture. In the A|RT architecture, this clustering is not possible; each functional unit is controlled by a part of the controller, which results in a larger amount of resources that can be used in parallel, but also in a larger total instruction width.

If we look at the usage of the tools, there are also a few distinct differences between the A|RT Designer tool and the Silicon Hive tool. The main difference is in the architecture description: in the Silicon Hive tool, this description has to be written completely by hand, whereas the A|RT architecture is described in terms of pragmas. Each of the

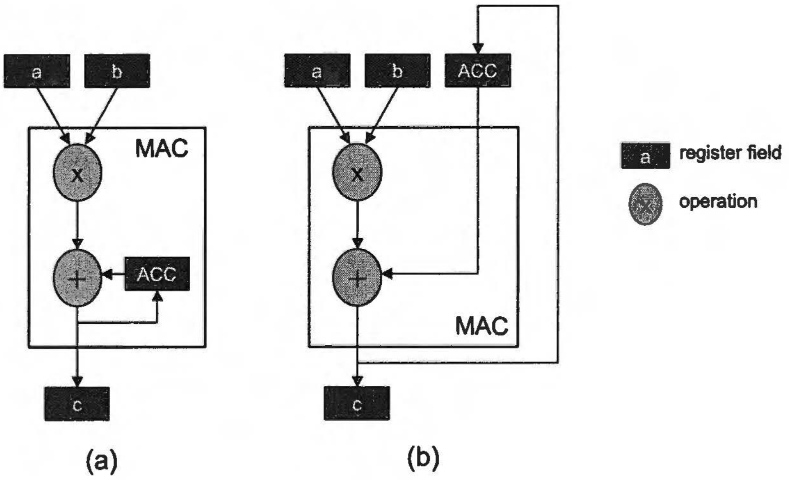


Figure 3-6: Modelling of state in functional units: (a) in A|RT Designer and (b) in Silicon Hive

pragmas instantiates a functional resource from a resource library, e.g., a RAM, an ALU or a controller. Depending on the application that has to be mapped onto the architecture, A|RT Designer adds all registers, busses and multiplexers automatically in the "Map to Architecture" phase (see Section 3.1.2). In case of the Silicon Hive tool, the user has to specify all the resources, busses and register files and their interconnections.

In terms of influencing the design flow, A|RT Designer is a winner: options are provided for all design steps, which offer a global handle on those steps. In addition, for every step performed by the tool, a set of pragmas can be created (see Figure 3-3), which provide a more fined-grained control on the way a certain step is performed. In case of the Silicon Hive tool, the user is only able to modify the architecture description to its needs and to put some extra constraints in the algorithmic description (e.g., total schedule latency, pipelining factor (initiation interval length) or timing constraints).

Finally, looking at the output generated by both tools, there is also a difference: for the Silicon Hive tool, the schedule (both graphical and textual) and the resource-, register- and bus-usage are all provided in one single file, whereas A|RT Designer generates a separate file for each of them. Furthermore, in case of the Silicon Hive tool, simulation of the algorithm can be performed after every design phase. With the A|RT Designer tool, this simulation has to be done separately. However, only the algorithm on C-level or HDL-level can be simulated, using an extra tool in case of the HDL-level (HDL simulator).

4. Scheduling for Multiple Controllers

As stated in Section 1.1, the main goal of this master's project is to generate static schedules for multiple controllers, in such a way that loop executions can be overlapped in time. Since no tools with an instruction scheduling scope spanning multiple controllers are available yet, this project aims at using currently available tools to generate such schedules. In this chapter, we show how this can be achieved. Therefore, in Section 4.1, we first show the main concepts of scheduling that are used in design tools like A|RT Designer and Silicon Hive. Next to that, in Section 4.2, we present how these concepts can be used to generate schedules for multiple controllers, taking into account the data dependencies that are present in an application, as well as the resource conflicts that occur during the parallel execution of loops. In Section 4.3, we present the results of a simple case study that has been performed to show the feasibility of the method and the (dis-) advantages of having multiple controllers in a VLIW data path. We end this chapter with some conclusions on the presented method and case study.

4.1. Scheduling basics

Any algorithmic description can be partitioned into basic blocks. A *basic block* is a sequence of consecutive statements of a program in which flow of control enters at the beginning and leaves at the end without halt or the possibility of branching (jumping to another basic block due to a conditional statement), except at the end of the block. Examples of basic blocks are `for`-loops, `if-then-` and `else-`statements.

Each basic block can be modelled by a *data flow graph* (DFG), which describes the primitive operations performed in that block, as well as the dependencies between those operations. Formally, a DFG is defined as follows ([Ku92]).

Definition 1 (Data Flow Graph) A data flow graph (DFG) is a triple $(V, E_s \cup E_d \cup E_l, w)$, where

- V is the set of vertices (operations);
- $E_s \subseteq V \times V$ is the set of sequence precedence edges;
- $E_d \subseteq V \times V$ is the set of data precedence edges;
- $E_l \subseteq V \times V$ is the set of loop-carried data precedence edges;
- w: $E_s \cup E_d \cup E_l \rightarrow Z$ is a function describing the timing delay (in clock cycles) associated with each precedence edge.

For reasons of simplicity, we assume that all operations have an execution delay of one clock cycle. A data precedence edge with weight $w(A, B)$ between node A and node B represents the consumption in node B of data produced by node A $w(A, B)$ clock cycles after the production of this data. A sequence precedence edges between node A and B represent that node A has to be executed $w(A, B)$ clock cycles before node B. A loop-carried data precedence edge with weight $w(A, A)$, finally, represents that the value produced by node A in iteration i of a loop is consumed $w(A, A)$ clock cycles later by the same node A in iteration $i+1$.

Two dummy nodes are always assumed to be present in the DFG: the *source* and the *sink*. The source node represents operations from outside the basic block, which produce data that is consumed by operations inside the basic block. The sink node represents operations from outside the block, which consume data that is produced inside the basic block. Both source- and sink nodes have no execution delay, and represent respectively the first and the last operation to be executed in the basic block. Usually, the ssource- and sink nodes are not shown when depicting a DFG.

The task of scheduling is to assign each operation $v \in V$ a start time $s(v)$. These start times are constraint by means of the precedences present in the DFG. A precedence edge $(v_i, v_j) \in E_s \cup E_d \cup E_l$ states that

$$s(v_j) \geq s(v_i) + w(v_i, v_j) \tag{1}$$

An example of a DFG is shown in Figure 4-1. Each of the operations in the DFG has a type and can be executed on a data path resource that supports this particular type. Examples of data path resources are functional units. When an operation is executed it uses a data path resource.

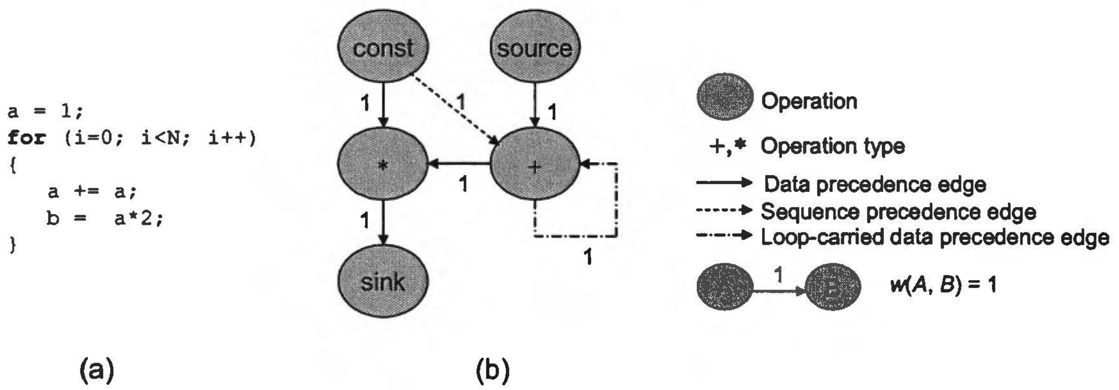
Obviously, two operations that are to be mapped onto the same functional unit cannot execute in parallel. Another example of why two operations cannot execute in parallel is the transport of the result of their computations over the same bus simultaneously, i.e., at the same moment. These constraints preventing the parallel execution of operations are called *resource constraints*, and are given as the function $rsc(v_i, v_j)$: $V \text{x} V \rightarrow \{0, 1\}$. This function is defined as follows:

$$rsc(v_i, v_j) = \begin{cases} 1 & \text{if } v_i \text{ and } v_j \text{ use the same resource} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

A resource constraint $rsc(v_i, v_j)$ expresses that

$$rsc(v_i, v_j) = 1 \quad \Rightarrow \quad s(v_i) \neq s(v_j) \tag{3}$$

Obviously, a valid schedule has to satisfy the resource constraints.



```
a = 1;
for (i=0; i<N; i++)
{
   a += a;
   b =  a*2;
}
```

Figure 4-1: An example of a data flow graph (DFG): (a) algorithmic description and (b) DFG representation of the algorithmic description

Timing constraints are specified as the latency and the initiation interval of the schedule ([Mes01]). The *latency L* of a schedule is the number of clock cycles after which all operations in the DFG (and all iterations of the DFG) are executed[2]. A constraint on the latency can be expressed in the DFG as a sequence precedence edge from the sink to the source with weight $-L$, as shown in Figure 4-2a. According to equation (1), this is interpreted as

$$s(source) \geq s(sink) - L \tag{4}$$

Because the source is always scheduled in cycle 0, and all other operations precede the sink, this implies that all operations have to finish execution within the first L clock cycles.

For loops, an initiation interval can be specified. The *initiation interval (II)* of a schedule is the number of clock cycles after which the next execution of the same DFG is started. If $II < L$ then the loop is folded, which means that operations from iteration i can be executed in parallel with (or even after) operations from iteration $i+1$ ([Bac94]). In general, this leads to faster schedules. In order to allow the folding of a loop, this loop is broken down into a preamble, a loop body or loop kernel, and a post amble. The loop kernel consists of operations that are executed repeatedly; the pre- and post amble consist of the remaining operations, which are necessary to allow the repeated execution of the loop kernel.

Just like the latency constraint, a translation to the precedence model is possible for the initiation interval: if loop iterations overlap, we have to ensure that data belonging to the current iteration is consumed before it is overwritten by the production of data in the next iteration. Thus, the operation that consumes the data produced by P should execute within II clock cycles after the operation P. Therefore, for each data precedence edge $(P, C) \in E_d$, an edge $C \rightarrow P$ with weight $-II$ is added to the DFG, as illustrated in Figure 4-2b.

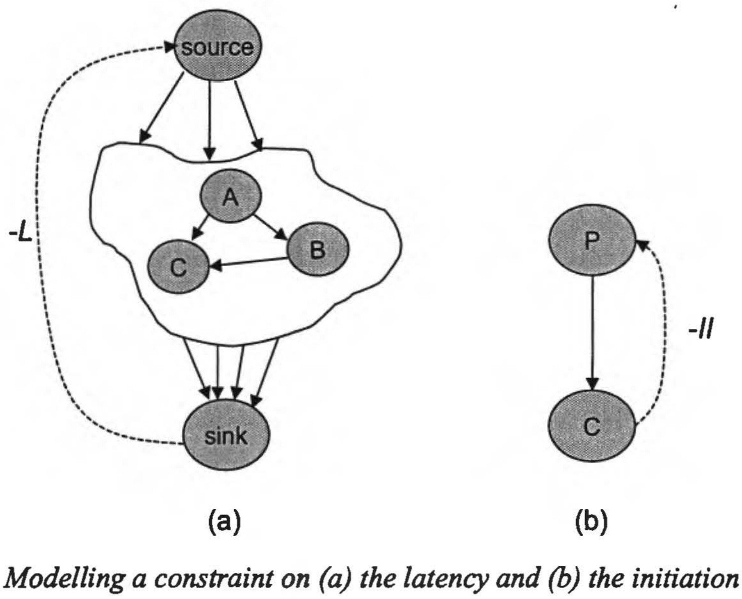


Figure 4-2: Modelling a constraint on (a) the latency and (b) the initiation interval

[2] Notice that the latency is a constant since the DFG describes a basic block.

After the introduction of these concepts we are now able to give a definition of scheduling.

Definition 2 (Scheduling) Given a DFG, a function $rsc(v_i, v_j)$, an initiation interval II, a constraint on the latency L (completion time) and an abstract description of the target architecture. Find a valid schedule s that satisfies the precedence constraints, the resource constraints and the timing constraints II and L.

Recall from the beginning of this chapter, that an algorithmic description can be partitioned into basic blocks. The composition of those basic blocks is in general modelled by means of a hierarchical data flow graph ([Lam88]). In such a graph, control-flow primitives such as branching and iteration are modelled by means of hierarchy. The edges in such graphs represent data precedence or sequence precedence constraints, whereas the nodes represent different basic blocks. Obviously, a hierarchical data flow graph can only be scheduled for architectures with a single flow of control.

In general, scheduling a hierarchical data flow graph consists of first scheduling the operations in the deepest nested basic block. Then this scheduled basic block is scheduled with the other operations of the surrounding block operation. This is repeated until all blocks are scheduled.

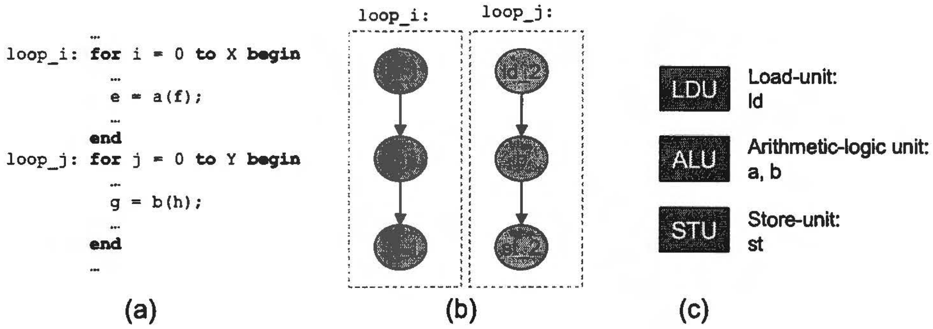
In order to come to a valid schedule and its corresponding program code, the scheduler takes as input a DFG of a basic block, an abstract description of the target processor and the timing constraints. The so-called *code generation* task of the scheduler consists of a number of phases, which can be distinguished: operation assignment, value lifetime serialization, scheduling and register binding. During the operation assignment phase, operations are assigned to functional units and the resource constraints are analysed. This results in extra sequence precedence edges in the DFG. During the lifetime serialization phase, operations are ordered in time in such a way that a valid register binding exists after the scheduling phase. Register binding determines how variables in the algorithm subject to scheduling can share common register files in an implementation. Finally, during the scheduling phase, the start times of operations are determined and the register binding is performed, in order to select a register location for every intermediate value in such a way that a minimum number of register locations is needed. These steps are repeated for every basic block that is present in the application, until a complete and valid schedule is found.

The presented phases are mutually dependent, that is, decisions in one phase can affect decisions made or still to make in other phases. Furthermore, during all code generation phases the precedence edges in the DFG are analysed and optimised in such a way that at the end a valid and as good as possible schedule exists. For this, analytical techniques (e.g., constraint analysis ([Mes01])) are used.

4.2. Scheduling for multiple controllers

So, how can we use the knowledge of how a single controller scheduler performs its job to generate folded schedules for multiple controllers in such a way that resource sharing during the parallel execution of loops is guaranteed to be without conflicts?

Consider an algorithm that consists of two loops, `loop_i` and `loop_j`, of which we



```
...
loop_i: for i = 0 to X begin
           ...
           e = a(f);
           ...
        end
loop_j: for j = 0 to Y begin
           ...
           g = b(h);
           ...
        end
...
```

Figure 4-3: Example of two parallel loops: (a) algorithmic description; (b) data flow graphs for the basic blocks; (c) functional units available in the target architecture and the operations which they are able to perform

assume that they can be executed in parallel, see Figure 4-3a. The algorithm is broken down into two basic blocks, which can be represented as data flow graphs (Figure 4-3b). The available functional units and the operations they are able to perform are shown in Figure 4-3c. Only one instance of each of the functional units is available in the target architecture, therefore all functional units will have to be shared among the basic blocks.

On an architecture with a single controller, these two basic blocks are scheduled sequentially (remember that the flow of control enters a basic block at the beginning of the block, and leaves at the end; thus multiple basic blocks cannot be scheduled in parallel with a single flow of control). Therefore, no overlapping of the operations in the basic blocks is possible. Because of this, we will have to schedule the two loops separately, if we want to be able to execute them in parallel. However, how can we than guarantee that no resource conflicts occur during the parallel execution?

The solution we propose is the following: combine the two loops, as if they were a single loop, and schedule this combined loop. During scheduling, the timing and resource constraints of the two loops are taken into account, and therefore the generated schedule will be conflict-free (provided of course that the target architecture and the imposed timing constraints allow a feasible schedule). The generated schedule will not be a valid schedule, in the sense that the execution of the schedule on a processor would not produce valid results, since inter-loop data dependencies are ignored. However, it allows us to determine which operations of the two loops have to be executed in what clock cycle (relative to the beginning of their corresponding basic blocks), in order to prevent resource conflicts from happening during the parallel execution of the two loops. This knowledge can than be used for the generation of the individual schedules for the loops.

By writing the two loops as one loop, their corresponding DFGs are also combined. During the different scheduling phases, extra sequence precedence edges are added to the DFG as a result of resource conflicts and inter-loop data dependencies, which represent data that is produced in one loop and consumed in another loop. In the example of Figure 4-3, sequence precedence edges as a result of resource conflicts will be added, as is illustrated in Figure 4-4a: the two loops both use the three available resources (LDU, ALU and STU), and therefore each of those operations has to be scheduled (at least) one clock cycle apart from the other. Applying normal scheduling techniques on the DFG with an initiation interval of two results in the folded schedule for the operations in the loop kernel that is shown in Figure 4-4b (the reader can easily verify that this is the minimum initiation interval possible for this example).

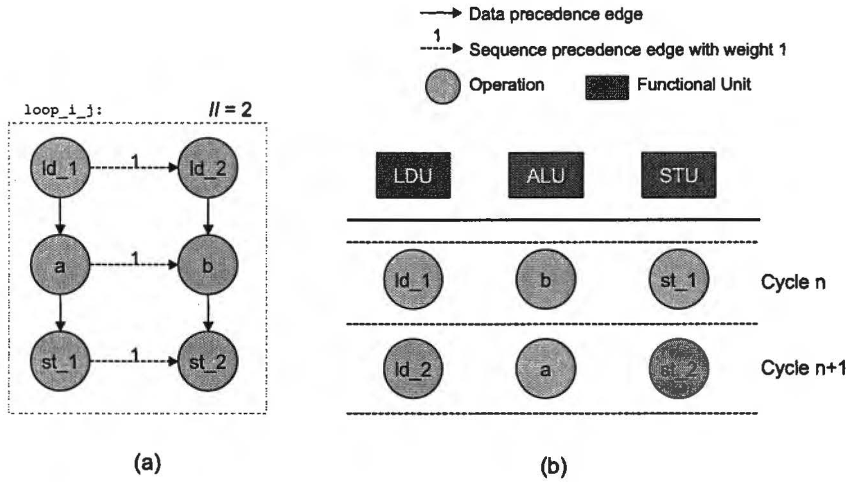


Figure 4-4: (a) Combined and updated DFG; (b) Folded schedule for the loop kernel of the two loops that are combined into one basic block

If we have been able to generate a schedule, which guarantees that the kernels of loops can run in parallel, without any resource conflicts, than, in general, the pre- and post amble of the two loops can also be executed in parallel. The reason for this is that in general the pre- and post amble of a loop are nothing more than an unrolling of the loop kernel. This is illustrated in Figure 4-5. That figure shows the executions of the two loops on two separate controllers, where each of the loops is executed with the same schedule for the loop kernel as was determined in Figure 4-4. The operations of every loop iteration in the pre- and post amble are still *II* cycles apart from each other, and are under constraint of the same data- and sequence precedence edges as the operations in the loop kernel. Because of that, no resource conflicts occur in the pre- and post amble, if the loop kernels are without any conflicts.

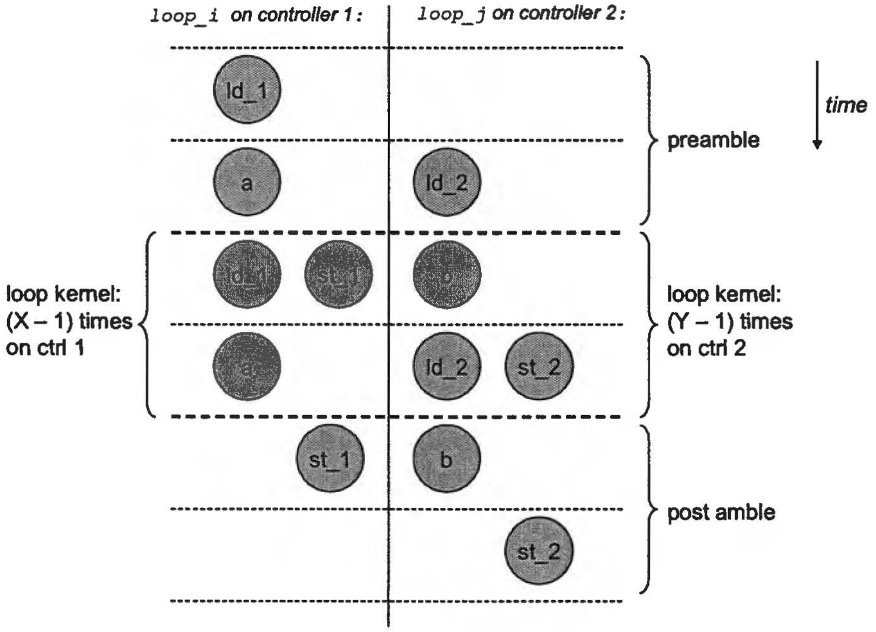


Figure 4-5: Schedules for both loops on separate controllers

Once the clock cycles in which the operations inside the loop kernels have to be executed are determined, scheduling each of the loops separately generates their individual schedules. Therefore, based on these observations, we come to the following steps for generating schedules for multiple controllers, which allow the overlapping of loop executions in time:

1. Determine which loops in the algorithm can be executed in parallel (based on knowledge of the algorithm);
2. Combine the operations of those loops into one loop, which the design tool will translate into one basic block;
3. Schedule this combined basic block;
4. Extract from the generated schedule "scheduling pragmas" for the loop kernels of the individual loops, which constrain the operations of the loops in fixed clock cycles;
5. Re-schedule the operations of each of the loops separately, with the corresponding scheduling pragmas, to generate the correct pre- and post ambles of the loops.

In the following section, these steps are illustrated by means of a simple case study.

4.3. Simple case study

For this simple case study, we have implemented the algorithm as presented in Figure 1-2 with the Silicon Hive tools. In Appendix A, the interested reader can find the HDFC-description of the algorithm that is used for the Silicon Hive tools, as well as the architecture onto which the algorithm is mapped. We restrict ourselves here to the data flow graphs for the two basic blocks of the algorithm as presented in Figure 1-2. These DFGs are shown in Figure 4-6. The labels inside the nodes correspond to the labels in the code presented in Appendix A.1; the sequence precedence edge from node `LD` to node `ST1` represents the data precedence between the two loops, as was illustrated in Figure 1-3.

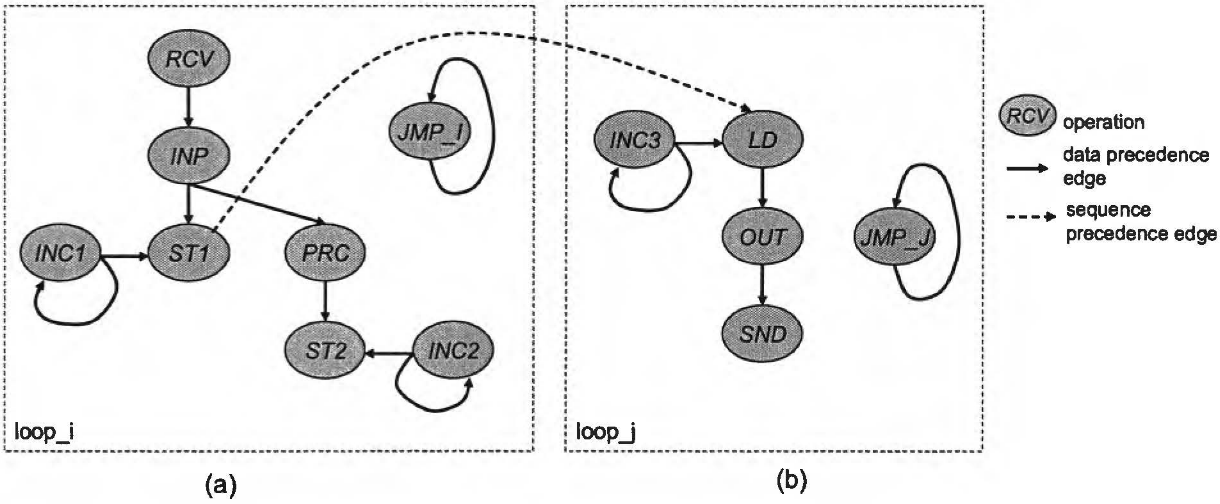
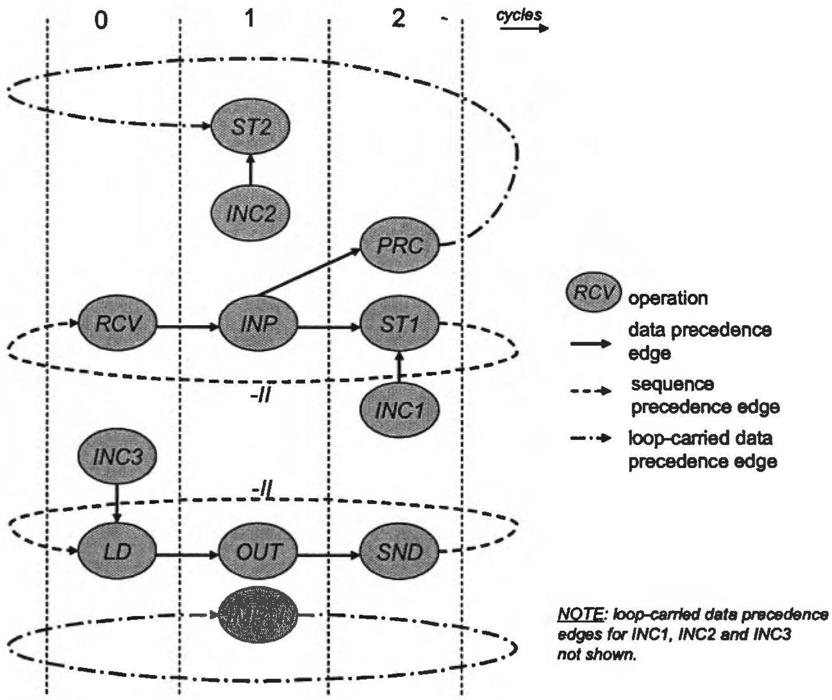


Figure 4-6: DFGs for the two basic blocks: (a) DFG for the loop with i as loop counter and (b) DFG for the loop with j as loop counter (see Figure 1-2)

In order to find a (folded) schedule that allows the overlapping execution of both loops, we combine the two loops into one basic block (in other words, in one loop with one loop counter) and schedule this combined basic block. Figure 4-7 shows the result of scheduling this combined basic block using the Silicon Hive tools. As illustrated in the figure, the initiation interval of the combined loop equals three, which means that every three clock cycles a new iteration of the corresponding data flow graph is started.

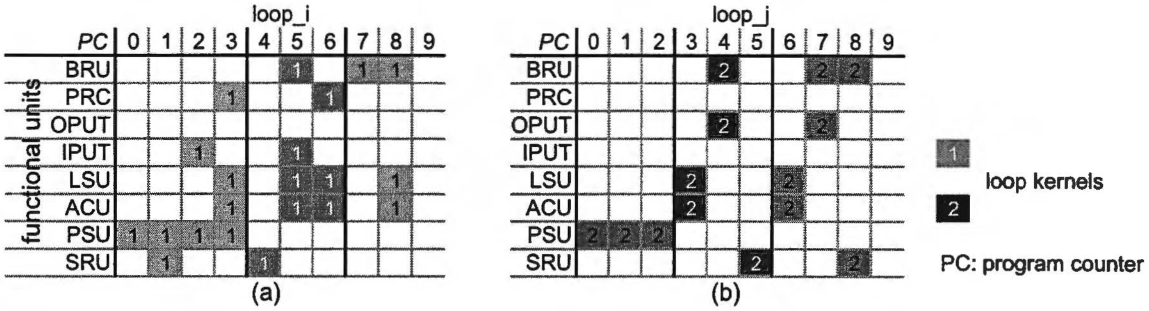
The schedules for executing each of the loops on a separate controller need to obey the timing constraints as imposed by scheduling the two loops as one combined loop. For that reason, we derive from the schedule shown in Figure 4-7 scheduling pragmas, which fix every operation in the DFG to a certain clock cycle. Using the Silicon Hive tools, annotating an operation v with an `IN_CYCLE(x)` pragma fixes the operation to a certain clock cycle x (in other words, $s(v) = x$), relative to the beginning of the basic block. For example, in order to let operation `RCV` be scheduled in the first cycle of a loop iteration, an `IN_CYCLE(0)` pragma is added in the algorithmic description for this operation. In that way, operations in the loop kernels (and thus also the operations in the pre- and post ambles of the loops) can be fixed to the clock cycles that allow parallel execution of both loops, as determined by scheduling the two loops as one.

Figure 4-8 shows the individual schedules for both loops, when mapped on the architecture presented in Appendix A.2. Note that the schedules contain operations, which are not present in Figure 4-6; that figure only shows the operations that are present inside the basic blocks (so in the loops). The extra operations in Figure 4-8 belong to the operations that are performed before the basic blocks, e.g., initialisation of loop counters, base- and offset addresses for memories, etc. These operations are not taken into account by the presented method. Because of these initialisation operations, still resource conflicts can occur, as is illustrated next.



NOTE: loop-carried data precedence edges for INC1, INC2 and INC3 not shown.

Figure 4-7: Schedule for the combined loop kernels



PC	0	1	2	3	4	5	6	7	8	9
BRU						1		1	1	
PRC				1			1			
OPUT										
IPUT			1			1				
LSU				1		1	1		1	
ACU				1		1	1		1	
PSU	1	1	1	1						
SRU		1			1					

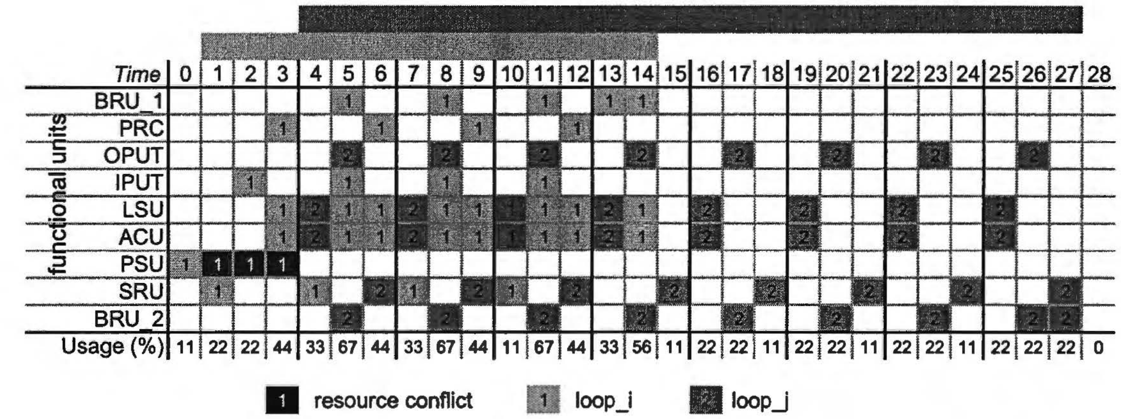
PC	0	1	2	3	4	5	6	7	8	9
BRU					2			2	2	
PRC										
OPUT					2			2		
IPUT										
LSU				2			2			
ACU				2			2			
PSU	2	2	2							
SRU						2			2	

Figure 4-8: Schedules for (a) loop_i and (b) loop_j, derived from the combined loop schedule

If the schedules were executed as presented in Figure 4-8, resource conflicts during the parallel execution of the loops would still occur, when both controllers start at clock cycle 0 with the execution of program counter (PC) 0: both controllers try to perform an operation on the LSU for PC = 3. Therefore, assuming that a controller is able to start at a program counter equal to zero for a clock cycle not equal to zero[3], the designer has to define a mapping of the program counters on the time, such that all resource conflicts due to initialisation operations are solved. For the presented example, a shift of one clock cycle for the second schedule, corresponding to satisfying the sequence precedence edge present between the node ST1 and LD as shown in Figure 4-6, is sufficient. In general, this mapping consists of shifting the program counters in such a way, that the kernels of the loops that have to be executed in parallel on different controllers start in the same clock cycle (which was a result of the combined schedule).

In Figure 4-9, the execution of both schedules of the previous figure on a single VLIW data path with multiple controllers is shown as a function of the time. The two bars above the diagram show during which clock cycles the operations of the two basic blocks are executed. Furthermore, they demonstrate that both loops run in parallel, without any resource conflicts during the parallel execution of the loops.

However, during the first four clock cycles in Figure 4-9, resource conflicts occur on the PSU unit. The operations executed during these clock cycles are initialisations of the



Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
BRU_1						1			1			1		1	1														
PRC				1			1			1			1																
OPUT						2			2			2			2			2			2			2			2		
IPUT			1			1			1			1																	
LSU				1	2	1	1	2	1	1	1	1	1	2	1		2			2			2			2			
ACU				1	2	1	1	2	1	1	1	1	1	2	1		2			2			2			2			
PSU	1	1	1	1																									
SRU		1			1		2	1		2	1		2			2			2			2			2			2	
BRU_2						2			2			2			2			2			2			2			2	2	
Usage (%)	11	22	22	44	33	67	44	33	67	44	11	67	44	33	56	11	22	22	11	22	22	11	22	22	11	22	22	22	0

Figure 4-9: Resource usage in time

[3] In a "normal" processor the program counter is reset to zero at clock cycle 0, so the controller will start executing the instruction word at PC equals zero after the reset.

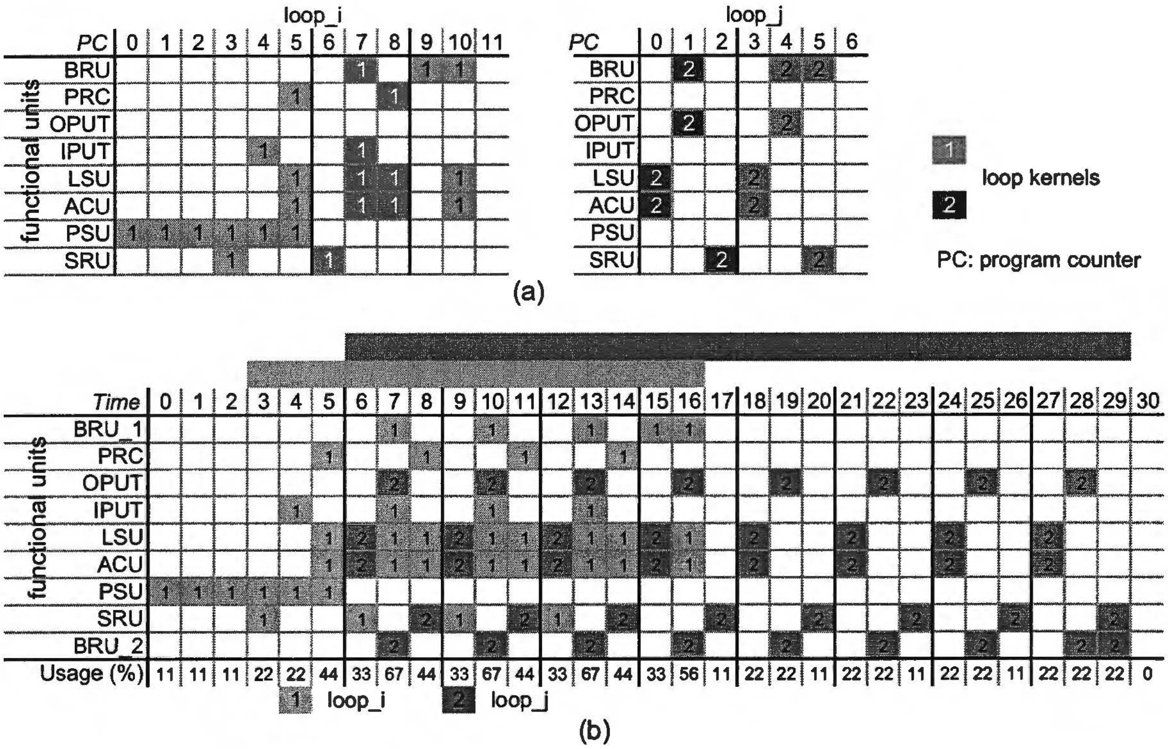

loop_i
PC 0 1 2 3 4 5 6 7 8 9 10 11
functional units
BRU
PRC
OPUT
IPUT
LSU
ACU
PSU
SRU
loop_j
PC 0 1 2 3 4 5 6
(a)
1
2
loop kernels
PC: program counter
Time 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
BRU_1
BRU_2
Usage (%) 11 11 11 22 22 44 33 67 44 33 67 44 33 67 44 33 56 11 22 22 11 22 22 11 22 22 11 22 22 22 0
1 loop_i
2 loop_j
(b)

Figure 4-10: Conflict-free schedule: (a) individual schedules for both loops and (b) resource usage in time

base- and offset addresses for the different memories, and the initialisation of the loop counters. In order to prevent such resource conflicts from happening, two options are possible: either the designer has to shift the schedules by hand in such a way, that the conflicts during initialisation are solved, or all initialisation operations have to be performed on one controller. We have chosen to perform the initialisation operations on a single controller, since in that way the compiler, instead of the designer, can solve the conflicts. Figure 4-10 shows the updated, conflict-free schedule.

As mentioned in Section 1.1, one of the goals of this project was to show the (dis-) advantages of having multiple controllers executing loops in parallel on a single VLIW data path. In order to show these (dis-) advantages, we have compared the total execution time, the memory requirements and the program code size of our solution to the results of a sequential and a loop-merged implementation of the algorithm on a VLIW with a single controller. In the loop-merged implementation, the presented algorithm is divided into three loops: one loop for the first few iterations of `loop_i`, one loop in which `loop_i` and `loop_j` are executed in parallel, and one loop for the final iterations of `loop_j`, as can be derived from Figure 1-3 ([Bac94]). This implementation allows the overlapping execution of the two loops using a single controller.

We expect our solution to have a shorter latency of the schedule (in terms of clock cycles) with respect to a sequential implementation (recall Figure 1-3b). From this figure, we also conclude that the memory requirements for a multiple controller implementation are expected to be smaller than for a sequential implementation. Furthermore, due to this shorter schedule, we expect a smaller program code size for our implementation.

Since a loop-merged implementation allows the overlapping execution of loops, we expect such an implementation to have the same latency and memory requirements as our approach, but a larger program code size: the algorithmic description has become more complex (three instead of two loops).

The results of the comparison are summarized in Table 4-1. The generated schedules for the sequential and loop-merged implementations are shown in Figure 4-11.

	Sequential	*Loop-Merged*	*Multiple Controllers*
Clock Cycles	24 cycles	23 cycles	30 cycles
Program Code Size	15 words x 50 bits (750 bits)	22 words x 54 bits (1188 bits)	Ctrl1: 12 x 48 bits Ctrl2: 7 x 46 bits (total: 898 bits)
Memory Requirements	Registers: 15 fields (86 bits) RAM: 8 fields (64 bits)	Registers: 16 fields (94 bits) RAM: 4 fields (32 bits)	Registers: 16 fields (91 bits) RAM: 4 fields (32 bits)

Table 4-1: Results of the case study summarized

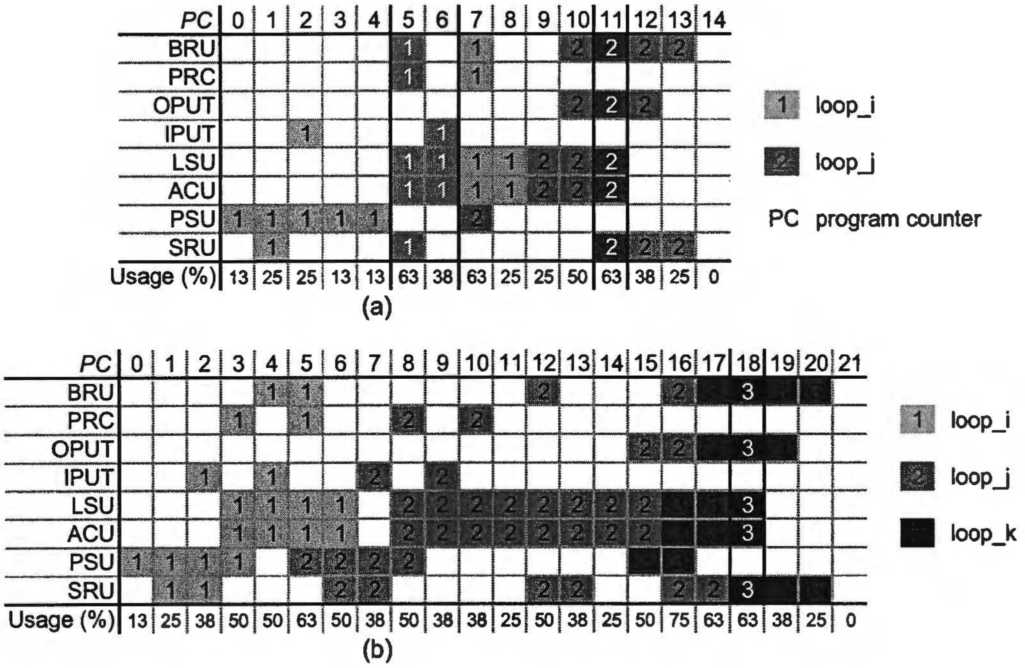
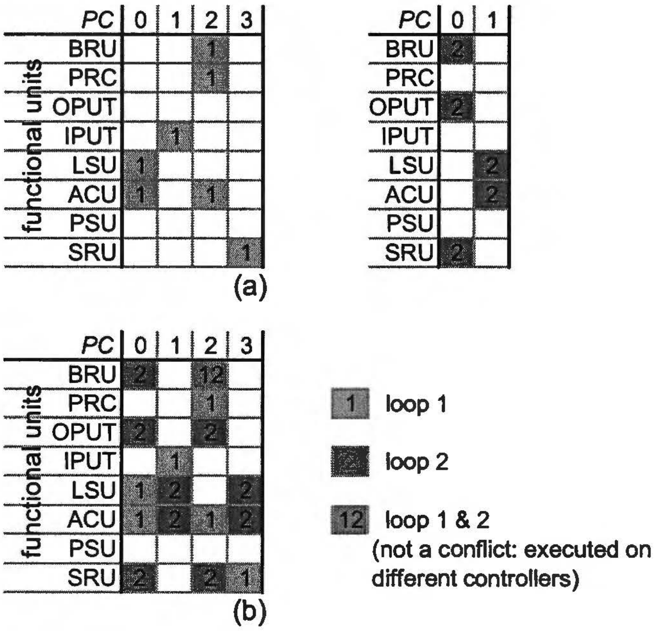


Figure 4-11: Schedules of (a) sequential implementation and (b) loop-merged implementation on a VLIW data path with a single controller

4.4. Conclusions of the case study

From Table 4-1, we conclude that the total number of clock cycles for a multiple controller implementation is larger than the number of clock cycles for both a sequential and a loop-merged implementation. This longer schedule is due to the overlapping execution of the two loops: in order to be able to generate conflict-free schedules for the loops, they have to be executed with an initiation interval of three, as shown in Figure 4-8. However, as soon as one of the loops is finished, the other loop continues its execution with an initiation interval of three, as shown in Figure 4-10b. If we compare this for example with the schedule of a sequential implementation (Figure 4-11a), we see that the loops can be executed with initiation interval smaller than three. In other words, if one of the loops finishes execution in our approach, the other loop continues execution with an initiation interval that is not minimal, resulting in the longer schedule.

In some cases, unrolling one of the loops generates a shorter schedule. Consider for example the sequential loop kernel schedules of two loops, as presented in Figure 4-12a, where the initiation intervals are a multiple of each other (4 and 2). If the second loop is unrolled one time *–unrolling* is the process of expanding a loop in such a way that every new iteration contains several of what used to be an iteration ([Bac94])– the schedule for parallel execution presented in Figure 4-12b is possible (assuming the dependencies between the loops allow this). Now, as soon as the first loop finishes execution in a multiple-controller implementation, the other loop continues to execute with an initiation interval of four for two iterations instead of only one. This results in a faster execution of the second loop and therefore in a shorter total schedule than for a sequential implementation. However, this is only possible if the loop kernels allow such overlapping.



PC	0	1	2	3
BRU			1	
PRC			1	
OPUT				
IPUT		1		
LSU	1			
ACU	1		1	
PSU				
SRU				1

PC	0	1
BRU	2	
PRC		
OPUT	2	
IPUT		
LSU		2
ACU		2
PSU		
SRU	2	

(a)

PC	0	1	2	3
BRU	2		12	
PRC			1	
OPUT	2		2	
IPUT		1		
LSU	1	2		2
ACU	1	2	1	2
PSU				
SRU	2		2	1

(b)

1 loop 1

2 loop 2

12 loop 1 & 2
(not a conflict: executed on different controllers)

Figure 4-12: Loop unrolling for reduction of the latency of the schedule: (a) loop kernel schedules for two loops; (b) schedule for execution on two controllers

Comparing the program code sizes in Table 4-1, we see that the program code size of the loop-merged implementation is larger than for a multiple controller implementation, as was expected. However, the program code size for a multiple controller implementation is larger than for a sequential implementation. We expected it to be smaller, since each of the controllers controls only a part of the VLIW data path, which results in smaller instruction words per controller. However, since each of the loops is executed with a non-optimal initiation interval, the total number of instruction words for the multiple controller implementation is larger than the number of instruction words necessary for the sequential implementation, which results in the larger total program code size for our approach.

With respect to the memory requirements, Table 4-1 shows that the total number of register locations is almost the same for each implementation. Furthermore, the loop-merged and multiple controller implementations both reduce the RAM requirements by a factor of two, due to the overlapping execution of the two loops (as was expected).

From these observations, we can conclude that for this case study a single controller approach achieves the best results. In general, if an application is to be mapped onto a single VLIW data path, a single controller implementation will probably achieve better results than a multiple controller implementation. This can be observed in Section 4.3: in order to be able to execute multiple loops in parallel, their corresponding schedules have to use an initiation interval, which is larger than could be achieved with a single controller implementation, in order to allow the overlapping execution of operations in the loops. The reason for this is the fact that the resources available in the single VLIW data path have to be shared among the different controllers. Only in those cases where loops have a strong control-dependency between successive loop iterations, a multiple-controller implementation will achieve better results. That is because the strong control-dependency results in a not too tight initiation interval, allowing unused resources to be used by other controllers instead of other loop iterations (in case of a tight initiation interval, most of the resources will be active, leaving no room for sharing them with other controllers).

In brief, the choice for mapping an application on an architecture with a single- or with multiple controllers heavily depends on several parameters of the application that is to be mapped (e.g., the initiation interval or the loop-boundaries).

4.5. Hardware implementation

As mentioned in Chapter 3, design tools like A|RT Designer and Silicon Hive can be used to generate an HDL description of a design. This HDL description can than be used to generate a net list, and to perform hardware synthesis and verification. However, the HDL descriptions generated by those tools contain a single controller, whereas for a hardware description of our implementation, multiple controllers have to be present. Therefore, in order to be able to perform hardware synthesis and verification on the design of a VLIW data path with multiple controllers, first an HDL description of the data path with a single controller has to be generated using a design tool like A|RT Designer or Silicon Hive. The designer than has to add the extra controllers and the corresponding control lines, as well as the program codes, to this design by hand. Finally, “OR” gates have to be added to the control-input for those resources that are controlled by multiple controllers. In that way, the control input for a certain resource is always a valid control signal, as already mentioned in Section 1.1: the generated static schedules prevent the simultaneous activity of multiple control signals for a certain resource, and therefore the “OR” gate always produces a valid control signal.

5. Multiple Controllers: a Case Study

The main goal of the case study presented in the previous chapter was to show the applicability of the method presented in the same chapter. In order to show the results of mapping a "real-life" application on a VLIW data path with multiple controllers, we present in this chapter another case study: the mapping of the Viterbi algorithm on a VLIW data path with multiple controllers.

Since the Viterbi decoding algorithm is an efficient method (in terms of search complexity) for decoding a convolutional code, we first present the principles of convolutional encoding (Section 5.1) and decoding (Section 5.2). In Section 5.3 we show how the Viterbi algorithm is implemented in general, and why it can be mapped on an architecture with multiple controllers. Finally, Section 5.4 presents the results of the case study, whereas Section 5.5 presents the conclusions of the case study.

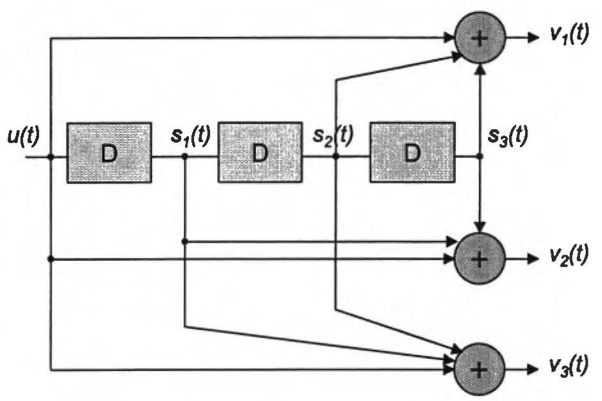
5.1. Convolutional coding

Convolutional encoding is based on generating linear combinations of delayed input samples, as illustrated in Figure 5-1. This encoder consists of three connected delay elements, and three modulo-2 adders. The output of a delay element at time *t* equals the input at time *t* - 1, where the time *t* is integer. For the input *u*(*t*) of the encoder we assume that

$$\begin{aligned} &u(t) \in \{0,1\}, \quad \mathit{for}\ t = 1,2,\ldots,T \\ &u(T+1) = u(T+2) = u(T+3) = 0 \end{aligned} \tag{5}$$

In this formula, T is the number of binary symbols that is to be encoded. The codewords are now created as follows:

$$\begin{aligned} &\mathit{inputword} = u(1), u(2), \ldots, u(T) \\ &\mathit{codeword} = v_1(1), v_2(1), v_3(1), v_1(2), v_2(2), v_3(2), \ldots, v_1(T+3), v_2(T+3), v_3(T+3) \end{aligned} \tag{6}$$



$$s_x(t) = u(t-x)$$

$$\begin{aligned} v_1(t) &= u(t) + u(t-2) + u(t-3) \\ &= \sum_{\tau=0,1,\ldots} u(t-\tau) g_1(\tau) = u * g_1(t) \quad , \mathit{for}\ g_1 = [1011] \\ v_2(t) &= u(t) + u(t-1) + u(t-3) \\ &= \sum_{\tau=0,1,\ldots} u(t-\tau) g_2(\tau) = u * g_2(t) \quad , \mathit{for}\ g_2 = [1101] \\ v_3(t) &= u(t) + u(t-1) + u(t-2) \\ &= \sum_{\tau=0,1,\ldots} u(t-\tau) g_3(\tau) = u * g_3(t) \quad , \mathit{for}\ g_3 = [1110] \end{aligned}$$

Figure 5-1: Example of a convolutional encoder

The key parameters for a convolutional encoder are the constraint length K and the rate R. The constraint length is equal to the number of delay elements plus one (the input), and determines the total number of values the state of the encoder can assume at any time (2^{K-1}). The rate denotes the number of encoder input bits per encoder output bit. For the encoder of Figure 5-1, K equals four and R equals 1/3.

Note, that a finite-state machine description characterizes a convolutional encoder. Figure 5-2 shows the state diagram corresponding to the encoder of Figure 5-1. In this figure, the states are denoted by $s_1s_2s_3(t)$; along the branches that lead from the current state $s_1s_2s_3(t)$ to the next state $s_1s_2s_3(t+1)$ we find the input that triggers the corresponding state transition, as well as the output generated by this transition ($u(t)/\ v_1v_2v_3(t)$).

5.2. Viterbi algorithm

To see what states can be reached as a function of the time t, we can take a look at the *trellis diagram*. The trellis diagram is a state diagram extended in time, such that for each possible input sequence the values of the states constitute a valid path through the diagram. A part of the trellis diagram corresponding to the encoder of Figure 5-1 is shown in Figure 5-3. Each codeword corresponds to a path in the trellis diagram. This path starts at time t = 0 in state $s_1s_2s_3(0)$ = 000, and wanders along T+3 branches. The steps taken along the time axis of the diagram are called *trellis steps*.

Since the convolutional encoding is based on the last K inputs affecting the output of the encoder, the decoding of these codes includes extracting the most likely encoder input making use of the received sequence. For this purpose, the trellis diagram is used.

Representing the encoding history in a trellis diagram makes it possible to define the decoding of convolutional codes as the problem of finding the most likely path taken during the encoding steps, given the input received at the decoder (which is called maximum likelihood decoding). This problem has been solved using a dynamic programming approach: the Viterbi algorithm ([Vit67]).

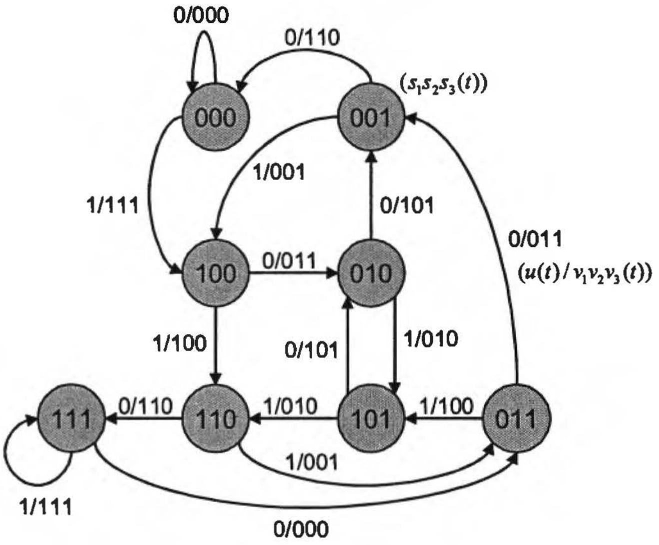


Figure 5-2: The state diagram

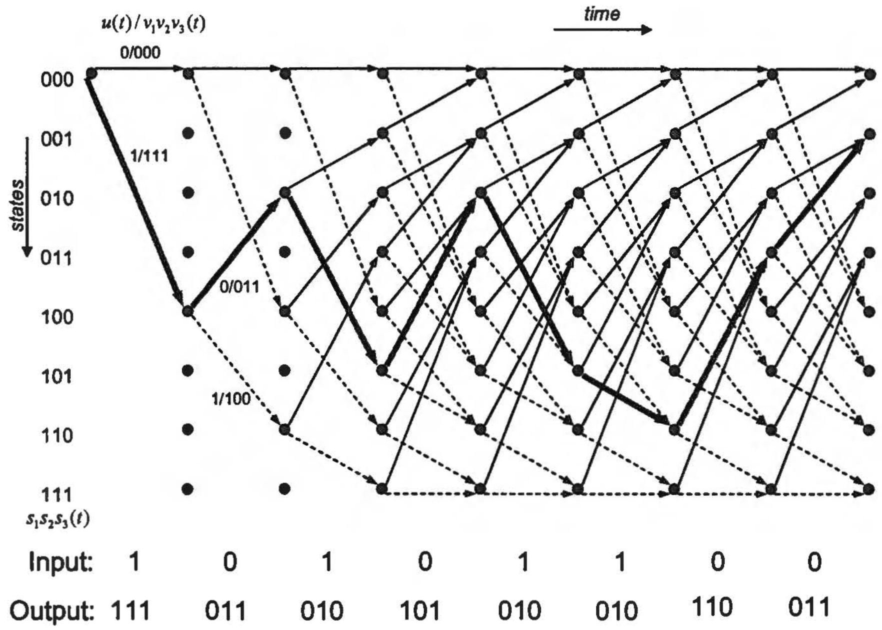


Figure 5-3: The trellis diagram corresponding to the encoder of Figure 5-1.

The main principle of the Viterbi algorithm is illustrated in Figure 5-4. Assume that a certain state $s(t+1)$ at time $t+1$ can be reached from two states $s'(t)$ and $s''(t)$ at time t via the branches $v'(t)$ and $v''(t)$, respectively. Then, according to [Vit67], the following holds:

$$\begin{aligned} \textit{a best path to } s(t+1) &= \textit{a best of } (\textit{all paths to } s'(t) \textit{ extended by } v'(t), \\ &\qquad \textit{all paths to } s''(t) \textit{ extended by } v''(t)) \\ &= \textit{a best of } (\textit{a best path to } s'(t) \textit{ extended by } v'(t), \\ &\qquad \textit{a best path to } s''(t) \textit{ extended by } v''(t)) \end{aligned} \tag{7}$$

Therefore, if we have already determined a best path leading from the start state $s(0)$ to state $s'(t)$ and a best path leading form the same start state to $s''(t)$, it is easy to find a best path leading to state $s(t+1)$: extend these paths with $v'(t)$ and $v''(t)$ respectively and chose the best one out these two alternatives. This best path to state $s(t+1)$ is called the *survivor* of state $s(t+1)$.

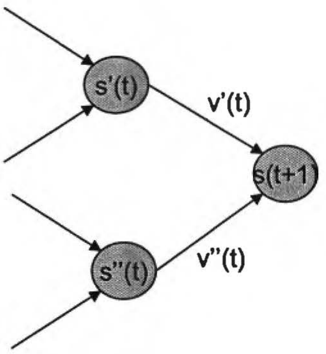


Figure 5-4: Paths leading to state s(t+1)

Following this principle, the Viterbi algorithm consists of tracing the trellis diagram forward in time (so-called *trellis construction*), and determining for every state at the corresponding time step the best path leading to that state, according to equation (7). For this, the decoder input is used to calculate the Hamming distance between the received codeword and the ideal encoder output for the branches (called *branch metrics*) leading to the current state, after which the one that has the minimum distance is chosen as survivor (the *Hamming distance* is the number of symbols that disagree between two codes). At the end of the decoder input data, the end state is determined by selecting the state with the smallest *path metric* (the sum of the branch metrics along the path leading to the selected state). Starting from this end state, the trellis diagram is traced back to recover the most likely encoder input sequence, until the first trellis step is reached. For this, the survivor information stored for each trellis step is used. When this best path is completely calculated, generating the inputs corresponding to the selected path generates the decoder output. This is called the *traceback* phase of the Viterbi algorithm.

5.3. Implementation of the Viterbi algorithm

Although the theoretical Viterbi decoding approach assumes performing the trellis construction and traceback for the whole decoder input all at once, this approach has a large latency and requires a large amount of memory. For that reason, in practice, the traceback step is started earlier and kept shorter. This is not in contradiction with getting a good error-correction performance, since the paths arriving at the various stages tend to converge already at a depth, which is far smaller than the length of the complete decoder input ([For73]). This approach is called the *sliding window approach*, and is illustrated in Figure 5-5. Each execution window (one vertical slice in the diagram) consists of D_{update} forward- and D_{trace} backward steps, which can be performed in parallel. At the end of

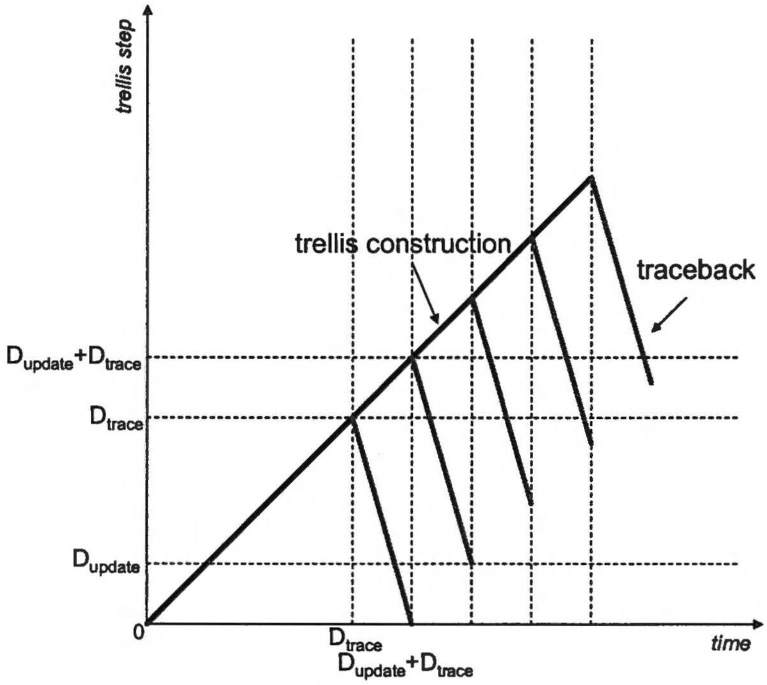


Figure 5-5: Sliding window approach

each traceback phase, D_{update} decoded symbols are produced.

As seen in Figure 5-5, for each of the D_{update} steps of trellis construction, D_{trace} steps of the traceback phase have to be completed. As the trellis construction is the most computation intensive part of the Viterbi decoding algorithm, it is not desired that the traceback phase costs more clock cycles than the trellis construction phase. Let $T_{trellis}$ be the number of clock cycles required for the calculations of one trellis step, and T_{trace} the number of clock cycles required for the calculations of one traceback step (which does not include the clock cycles needed for producing the decoded bits). Then the previous observation results in the following requirement for the sliding window implementation of the Viterbi algorithm:

$$D_{update} x T_{trellis} \geq D_{trace} x T_{trace} \tag{8}$$

From Figure 5-5 it follows that the sliding window approach for the Viterbi algorithm is a good candidate for the implementation on a VLIW data path with multiple controllers, since the trellis construction and traceback phases can be executed in parallel on different controllers. For that reason, we present in the following section the results of mapping the Viterbi algorithm on a VLIW data path with two controllers.

5.4. Results for the Viterbi algorithm

For this case study, we have implemented the Viterbi algorithm using the sliding window approach. A pseudo-code representation of the implemented algorithm is presented in Appendix B.1; Figure 5-6 provides an overview of the main structure of this algorithm. Both algorithmic descriptions only present the part of the Viterbi algorithm where the trellis construction and traceback phases are executed in parallel (see Figure 5-5), since this is the most interesting part for our work.

In the algorithmic description of Figure 5-6, the `init_tb` and `init_tc` functions represent the initialisation operations necessary for the traceback and trellis construction phases, respectively. D_{trace} traceback steps have to be performed in the loop `tb_loop_t` in parallel with D_{update} trellis construction steps in `tc_loop_t`. Each trellis construction step consists of the calculation of the best paths to every state according to equation (7) (performed in `tc_loop_s`). Finally, after the D_{trace} traceback steps have been performed, D_{update} decoded output bits are produced in `tb_loop_out`.

```
VITERBI
while()
  init_tb(); /* traceback initialisation operations */
  for t ← 0 ... $D_{trace}$-1                              tb_loop_t
    { /* traceback operations */ }
  for i ← 0 ... $D_{update}$-1                             tb_loop_out
    { /* decoder output operations */ }
  init_tc(); /* trellis construction initialisation operations */
  for t ← 0 ... $D_{update}$-1                             tc_loop_t
    for s ← 0 ... S-1                                       tc_loop_s
      { /* trellis construction operations */ }
```

Figure 5-6: Pseudo-code representation of the used Viterbi algorithm

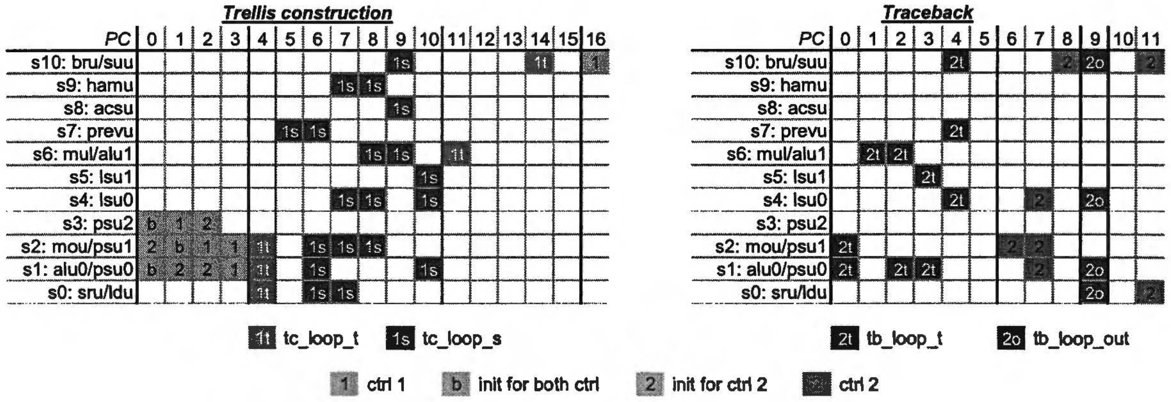
An efficient implementation of the Viterbi algorithm has to meet the requirement mentioned in equation (8). $T_{trellis}$ in this equation equals the number of clock cycles needed for the execution of the complete loop `tc_loop_s`. This number of clock cycles grows exponentially as a function of the constraint length K, since $S = 2^{K-1}$. On the other hand, T_{trace} equals the number of clock cycles necessary for one traceback step (one iteration of `tb_loop_t`) and is independent of the constraint length. We take $D_{update} = 8$ and $D_{trace} = 64$ (16*K); T_{trace} equals 4 and $T_{trellis}$ equals S*6 in our implementation. Therefore, in order to make sure that the traceback and trellis construction phases consume approximately the same amount of clock cycles and to prevent an explosion in the number of clock cycles necessary for the execution of the trellis construction phase for our current implementation of the Viterbi algorithm, we take $K = 4$. With this value for K we still meet the requirement mentioned in equation (8):

$$D_{update}xT_{trellis} \geq D_{trace}xT_{trace} \;\; \Rightarrow \;\; 8x(8x6) \geq 64x4 \tag{9}$$

In order to execute the Viterbi algorithm as presented in Figure 5-6 on a VLIW data path with multiple controllers, we first generate a combined static schedule for `tb_loop_t` and `tc_loop_s`. Based on this combined schedule we generate scheduling pragmas, which are used as constraints for the generation of the schedules for the traceback and trellis construction phases. Due to overlapping loop-boundaries, we also have to ensure that `tc_loop_t` and `tb_loop_t` can be executed in parallel on separate controllers. Figure 5-7 shows the final, conflict free folded schedules for each controller. Again, as already mentioned in the previous chapter, we have chosen to perform all initialisation operations (`init_tb` and `init_tc`) on one controller (in our case the controller for the trellis construction phase, since this phase starts executing before the traceback phase).

In Figure 5-8, the execution of the schedules on a VLIW data path with multiple controllers is shown as a function of the time. The bars below the diagram show, which (iterations) of the loops present in Figure 5-6 are executed during which clock cycles. Furthermore, they demonstrate that both trellis construction and traceback phases are indeed executed in parallel, without any resource conflicts.

Again, we have compared the total execution time of the generated schedules, the memory requirements and the program code sizes of our multiple controller implementation to the results of a sequential and a loop-merged implementation of the algorithm. The results of this comparison are presented in Table 5-1; in Appendix B.2 the schedules



Trellis construction

PC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
s10: bru/suu										1s					1t		1
s9: hamu								1s	1s								
s8: acsu										1s							
s7: prevu						1s	1s										
s6: mul/alu1									1s	1s		1t					
s5: lsu1											1s						
s4: lsu0								1s	1s		1s						
s3: psu2	b	1	2														
s2: mou/psu1	2	b	1	1	1t		1s	1s	1s								
s1: alu0/psu0	b	2	2	1	1t		1s				1s						
s0: sru/ldu					1t		1s	1s									

Traceback

PC	0	1	2	3	4	5	6	7	8	9	10	11
s10: bru/suu					2t				2	2o		2
s9: hamu												
s8: acsu												
s7: prevu					2t							
s6: mul/alu1		2t	2t									
s5: lsu1				2t								
s4: lsu0					2t			2		2o		
s3: psu2												
s2: mou/psu1	2t						2	2				
s1: alu0/psu0	2t		2t	2t				2		2o		
s0: sru/ldu										2o		2

Figure 5-7: Folded schedules for the trellis construction and traceback phases

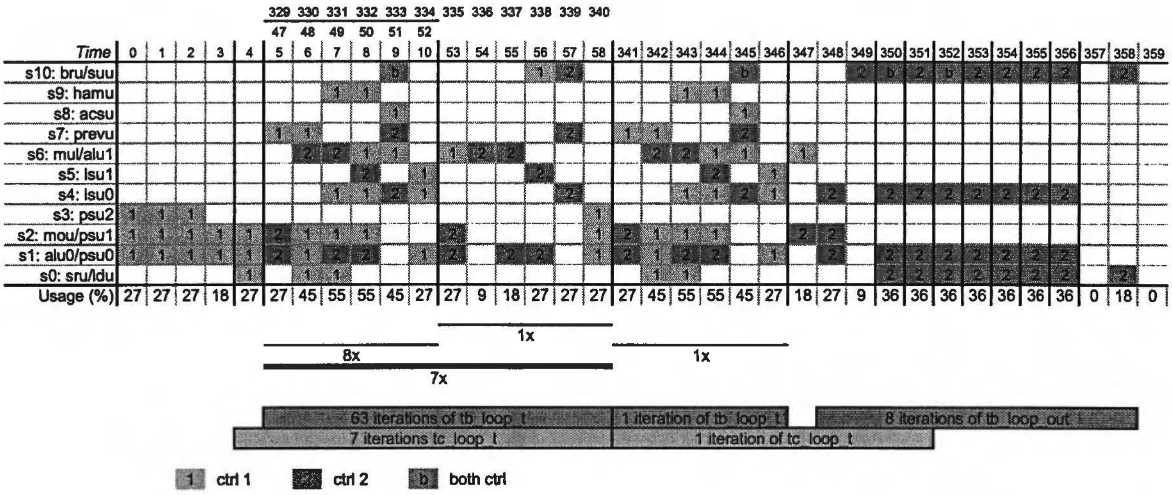


Figure 5-8: Execution of the Viterbi algorithm as a function of the time

	Sequential	*Loop-Merged*	*Multiple Controllers*
Clock Cycles	678 cycles	423 cycles	360 cycles
Program Code Size	25 words x 180 bits (4500 bits)	18 words x 197 bits (3546 bits)	Ctrl1: 17 x 180 bits Ctrl2: 13 x 177 bits (Total: 5335 bits)
Memory Requirements	Registers: 30 fields RAM: 656 fields	Registers: 38 fields RAM: 656 fields	Registers: 39 fields RAM: 656 fields

Table 5-1: Results of the Viterbi case study

for both sequential and loop-merged implementations of the Viterbi algorithm are presented for the interested reader.

5.5. Conclusions

From Table 5-1 we conclude that executing the Viterbi algorithm as presented in the previous section on a VLIW data path with multiple controllers achieves a better performance in terms of clock cycles than a pure sequential or loop-merged implementation on a VLIW data path with a single controller. The reason for this is the fact that both trellis construction and traceback phases are executed simultaneously. This parallel execution is possible, since `tc_loop_s` has a strong control-dependency, resulting in an initiation interval that leaves “room” for the operations of `tb_loop_t` to be executed in parallel.

Comparing the program code sizes, we see that the multiple controller implementation has the largest program code size, despite the fact that it needs the least amount of clock cycles for execution. The reason for this is the fact that both controllers still control a large part of the available data path, resulting in long instruction words per controller. Furthermore, both `tb_loop_t` and `tc_loop_t` run with an initiation interval that is larger than their minimum possible initiation interval (see Appendix B.2), which results in a lot of extra NOPs in the program code. These larger initiation intervals are necessary to allow the overlapping execution of the trellis construction and traceback phases, and result in a larger total sum of instruction words. Together with the large instruction width, this results in a larger total program code size than for a sequential or loop-merged implementation.

With respect to the requirements for the register files, the total number of register locations is almost the same for both multiple controller and loop-merged implementation, whereas those implementations both require more register locations than a sequential implementation. The reason for this is straightforward: in both loop-merged and multiple controller implementations, more variables are alive at the same time due to the parallel execution of multiple loops, which results in the larger amount of register locations.

With respect to the memory requirements for the RAM, all implementations are equal. This is a result of the way the Viterbi algorithm has been implemented in the sequential case: during the trellis construction steps, survivor information is stored for each state. This information is used during the traceback phase. To minimize the total amount of survivor information that needs to be stored in the memory, this information is stored in a circular manner, since only the information for the last $D_{trace}+D_{update}$ steps has to be stored (illustrated by Figure 5-5): at any point in time, the survivor information of D_{trace} steps is necessary for performing the traceback phase, and, in parallel, the survivor information for the next D_{update} steps has to be stored in the trellis construction phase. After the traceback phase, D_{update} bits are decoded, and the survivor information for the first D_{update} steps that were stored in memory are not needed anymore. This results in only having to store the survivor information for the last $D_{trace}+D_{update}$ steps in the RAM. In case of the multiple controller information, this does not change, and thus still the survivor information for $D_{trace}+D_{update}$ steps has to be stored, and therefore no gain in memory storage is achieved.

6. Conclusion

In this final chapter, we summarize the results of the two case studies that have been presented in the previous chapters, and provide some recommendations for future work.

6.1. Conclusions

In Chapter 4, we have presented a method for the generation of schedules for a VLIW data path with multiple controllers, based on design tools that generate schedules for a VLIW data path with a single controller. This method has been applied to two case studies, in order to show the feasibility of the method and the (dis-) advantages of having a VLIW data path with multiple controllers. From the case studies we can conclude that the presented method for the generation of schedules for multiple controllers, using design tools that are not meant for that, works. Furthermore, from those case studies we can conclude that it depends on the type of application whether a multiple-controller solution achieves the best results. More specifically, it depends on the amount of control-dependency between successive loop iterations, and on the loop boundaries.

In the case study of Section 4.3, both loops have to be executed with an initiation interval that is larger than the initiation intervals possible in case of a sequential (single-controller) implementation. In other words, both loops have a weak control-dependency, which results in a small initiation interval for a sequential implementation. However, due to this small initiation interval, insufficiently unused resources are available to allow the execution of multiple loops in parallel. This results in the larger initiation intervals for the parallel execution of both loops on separate controllers. Furthermore, one of the loops continues execution with a large initiation interval after the other loop has finished execution (see Figure 4-10b). This is the main reason why for many applications the implementation on a VLIW data path with multiple controllers will result in a larger number of clock cycles than an implementation on a VLIW with a single controller.

On the other hand, in case of the Viterbi algorithm (Section 5.4), one of the loops (`tc_loop_s`) has a strong control-dependency, resulting in the fact that the resources, which are unused during the execution of one iteration of that particular loop, can be used to execute operations of other loops in certain clock cycles in parallel (`tb_loop_t` in this case). Furthermore, the loop boundaries of the two main loops that have to be executed in parallel for the Viterbi case study "match": during the iterations of `tc_loop_t`, all iterations of `tb_loop_t` can be completed, which results in a reduction of the total execution time with respect to a single-controller implementation by almost fifty percent.

In general, the total program code size for a multiple-controller implementation will be larger than the program code size for a sequential implementation. The reason for this is the fact that each controller in most cases still controls a large part of the available data path, resulting in instruction words that have almost the same width as in a sequential implementation. Furthermore, since in a multiple-controller implementation the loops are often executed with an initiation interval that is not the minimum possible initiation interval, the total number of instruction words is larger and contains more NOPs than the total number of instruction words for a sequential implementation. Obviously, these two factors result in a larger program code size. However, in some specific cases it is possible that the program code size for a multiple-controller implementation is smaller than for a single-controller implementation, e.g., if the controllers each control a separate part of the available data path (no resource sharing).

With respect to the total number of register locations, the multiple-controller implementation performs more or less the same as a loop-merged implementation, and worse than a sequential implementation. The reason for this is straightforward: due to the parallel execution of multiple loops, more variables are alive at the same moment, which need to be stored in register fields.

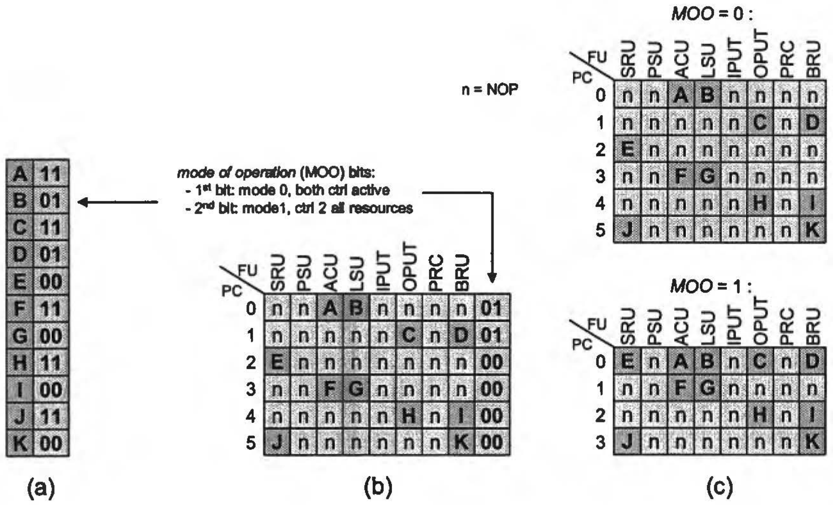
Finally, a multiple-controller implementation does not guarantee that the total amount of RAM necessary for a certain application is smaller than for a sequential implementation, as is shown by the case study of Chapter 5. It depends on the implementation of the algorithm whether any savings are possible for a multiple-controller implementation.

6.2. Recommendations for future work

From the previous section it follows that the two main issues that still have to be solved are the large amount of NOPs in the program code and the "initiation interval problem". The initiation interval problem refers to the continuation of the execution of a loop with a non-optimal initiation interval after other controllers have finished execution. This problem is illustrated in Figure 4-10b: after the first controller has finished execution, the second controller continues executing `loop_j` with an initiation interval of three, whereas an initiation interval of one is the minimum possible initiation interval. Obviously, if we were able to switch to a different initiation interval, the total execution time would decrease.

A solution to this problem is to use an instruction encoding approach similar to the one used in the Texas Instruments C6xx family ([Dil99]). In this approach, the program code consists of a stream of operations. Each of those operations has to be executed on a certain issue slot. Instead of extending every operation with an extra bit that indicates whether the next operation should be executed in the same clock cycle or in the next one (C6xx approach), we propose to extend every operation with one or more bits, which correspond to a certain *mode of operation* (MOO). These MOO-bits indicate for all possible operation modes whether the next operation should be executed in the same or in the next clock cycle as the current operation. However, the main difference with the C6xx approach is that multiple MOO-bits are possible, depending on the number of possible operation modes. In order to use the different operation modes efficiently, the controllers have to be able to switch operation mode during execution.

The concept of operation modes is illustrated in Figure 6-1a. This figure takes the schedule for `loop_j` as presented in Figure 4-10a as an example (Figure 6-1c for *MOO* = 0 shows this schedule). From the resource usage as a function of the time, presented in Figure 4-10b, we conclude that we have two operation modes, since we want to execute the schedule of `loop_j` with an initiation interval of one once the first controller (for `loop_i`) has finished execution. Therefore, the operations are extended with two MOO-bits. Depending on the current operation mode of the controller, the schedules as


n = NOP
mode of operation (MOO) bits:
- 1st bit: mode 0, both ctrl active
- 2nd bit: mode1, ctrl 2 all resources
(a)
(b)
(c)

A	11
B	01
C	11
D	01
E	00
F	11
G	00
H	11
I	00
J	11
K	00

PC \ FU	SRU	PSU	ACU	LSU	IPUT	OPUT	PRC	BRU	
0	n	n	A	B	n	n	n	n	01
1	n	n	n	n	n	C	n	D	01
2	E	n	n	n	n	n	n	n	00
3	n	n	F	G	n	n	n	n	00
4	n	n	n	n	n	H	n	I	00
5	J	n	n	n	n	n	n	K	00

MOO = 0 :

PC \ FU	SRU	PSU	ACU	LSU	IPUT	OPUT	PRC	BRU
0	n	n	A	B	n	n	n	n
1	n	n	n	n	n	C	n	D
2	E	n	n	n	n	n	n	n
3	n	n	F	G	n	n	n	n
4	n	n	n	n	n	H	n	I
5	J	n	n	n	n	n	n	K

MOO = 1 :

PC \ FU	SRU	PSU	ACU	LSU	IPUT	OPUT	PRC	BRU
0	E	n	A	B	n	C	n	D
1	n	n	F	G	n	n	n	n
2	n	n	n	n	n	H	n	I
3	J	n	n	n	n	n	n	K

Figure 6-1: Modes of operation illustrated: (a) proposed instruction encoding in case of data-stationary instruction encoding; (b) proposed instruction encoding in case of time-stationary instruction encoding; (c) resulting schedules for different modes of operation

presented in Figure 6-1c are possible.

Note, that an interesting side effect of this approach is a more compact program code than in the case of Figure 6-1c, since all NOPs are removed from the program code to form the stream of operations, as illustrated in Figure 6-1a. However, this method is only applicable if the controller supports *data-stationary instruction encoding*, meaning that the instruction words consist of clusters, which each specify an operation for one issue slot, as well as the source- and destination register locations ([Lap94]). Thus, the instruction word contains as many clusters as there are issue slots in the architecture. In case of the stream of operations, every instruction word consists of exactly one cluster.

An important disadvantage of data-stationary instruction encoding is that no efficient encoding scheme is known that supports multi-casting, since multi-casting requires the option to specify a variable number of destination addresses per operation on a certain issue slot. If the length of the instruction words is fixed, then the worst-case number of destinations must be specified for every operation. This increases the instruction word size dramatically. Therefore, if multi-casting is desired, an alternative encoding scheme has to be used: *time-stationary instruction encoding* ([Lap94]). In such an encoding scheme, the instruction words typically consist of an opcode cluster, a source address cluster and a destination address cluster. The opcode cluster determines the operations that have to be executed on each of the functional units present in the data path. The source- and destination address clusters specify per register file the register locations where data is read from or written to, respectively, for all issue slots at once, as well as the busses that have to be used for the data-transport. In that way, all register files are present in the instruction word only twice, instead of for every operation: once to address the source registers for all functional units, and once to address the destination registers for all functional units.

In case of time-stationary instruction encoding, the previously mentioned approach of extending operations with MOO-bits does not work, since the instruction bits are not grouped per issue slot anymore. However, a similar approach is possible: extend every instruction word with MOO-bits. Each of those MOO-bits then indicates whether it is possible to execute the next instruction word in the same or in the next clock cycle as the current instruction word. If multiple instruction words can be issued in the same clock cycle, the bit-wise "AND" of those instruction words is executed. To allow this, the controller fetches the instruction words in an instruction fetch buffer (FIFO), which has a size equal to the maximum number of instruction words that can be executed in one clock cycle. Due to this instruction fetch window, the controller does not have to wait for fetching more instructions if multiple instructions have to be executed in the same clock cycle: these are than already present in the buffer.

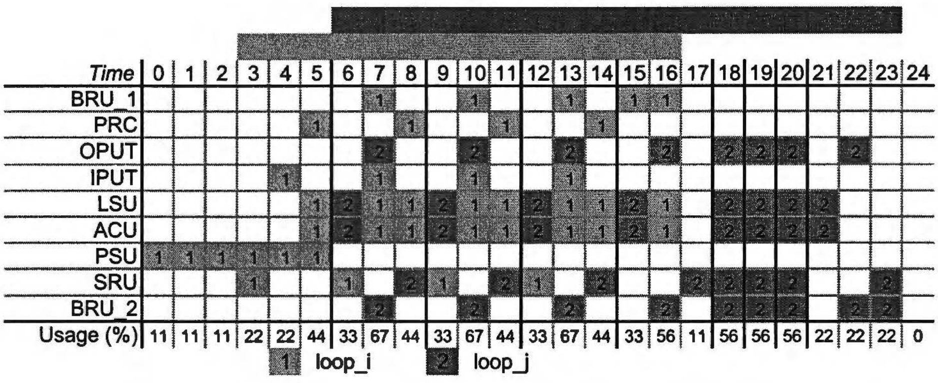
The concept of extending instruction words with MOO-bits is illustrated in Figure 6-1b. Contrary to the proposed solution for data-stationary instruction encoding, this approach does not result in a compacter program code, so still a solution for NOP-encoding has to be found when time-stationary instruction encoding is applied in order to decrease the total program code size.

The (expected) result of extending operations or instruction words (depending on the instruction encoding approach chosen) with MOO-bits is shown in Figure 6-2 for the case study as presented in Section 4.3. From this figure we can conclude, that extending the controller to support different modes of execution can result in an improvement of performance, and therefore we recommend further research into this approach.

In order to prove the practical feasibility of a VLIW data path with multiple controllers, a demonstration processor down to RT-level can be designed. This demonstration processor can also be used to determine the cost, e.g., in terms of area and power consumption, of having multiple controllers controlling a VLIW data path.

Since no (commercial) design tools with an instruction scope spanning multiple controllers exist yet, the development of such tools remains an interesting research topic. This type of tool could not only be used for the generation of schedules for a VLIW data path with multiple controllers, but also for the generation of schedules for multi-processor architectures.

With the current method, loops are supposed not to contain conditional statements. This however is not realistic, and therefore it is recommended that support for conditional statements is added to the method.



Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
BRU_1								1			1			1		1	1								
PRC						1			1			1			1										
OPUT								2			2			2			2		2	2	2		2		
IPUT					1			1			1			1											
LSU						1	2	1	1	2	1	1	2	1	1	2	1		2	2	2	2			
ACU						1	2	1	1	2	1	1	2	1	1	2	1		2	2	2	2			
PSU	1	1	1	1	1	1																			
SRU				1			1		2	1		2	1		2			2	2	2	2			2	
BRU_2								2			2			2			2		2	2	2		2	2	
Usage (%)	11	11	11	22	22	44	33	67	44	33	67	44	33	67	44	33	56	11	56	56	56	22	22	22	0

1 loop_i 2 loop_j

Figure 6-2: Schedule for case study of Section 4.3 when the instruction words are extended with mode of operation bits (the switch in operation mode is performed when the first controller finished execution)

Finally, since the proposed VLIW architecture has a close resemblance to a multi-processor architecture, it might be interesting to provide the designer with suggestions about when to chose for a VLIW data path with multiple controllers, and when to choose a multi-processor architecture.

Bibliography

[Adelan] Adelante Technologies, http://www.adelantetech.com.

[Aga93] Agarwal, A. et al., “Sparcle: an evolutionary processor design for large-scale multi-processors”, *IEEE Micro*, Vol. 13(3), May 1993, pp. 48-61.

[ARM] ARM, http://www.arm.com.

[ART] A|RT Designer Reference Manual.

[Bac94] Bacon, D.F. et al., “Compiler Transformations for High-Performance Computing”, *ACM Computing Surveys (CSUR)*, Vol. 26(4), Dec. 1994, pp. 345-420.

[Cha99] Chappell, R. et al., “Simultaneous Subordinate Microthreading (SSMT)”, *Proc. 26th Int. Symp. Computer Architecture*, May 1999, pp. 186-195. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[Den94] Dennis, J.B. and G.R. Gao, “Multithreaded Architectures: Principles, Projects and Issues”. In: **[Ian94]**.

[Dil99] Dillon, T.J. and P.B. Lopes, “TMS320C6x: a VLIW (very long instruction word) architecture for DSP applications”, *Proc. of ICMP’99 Int. Conf. on Microelectronics and Packaging*, Vol. 2, Curitiba, Brazil, 10-14 Aug. 1999, pp. 437-444.

[Egg97] Eggers, S.J. et al., “Simultaneous Multithreading: a Platform for Next-Generation Processors”, *IEEE Micro*, Vol. 17(5), Sept./Oct. 1997, pp. 12-19.

[For73] Forney, G.D., “The Viterbi Algorithm”, *Proc. IEEE*, Vol. 61, March 1973, pp. 268-278.

[Fra93] Franklin, M., “The Multiscalar Architecture”, *Computer Science Technical Report No. 1196*, University of Wisconsin-Madison, WI, USA, 1993.

[Hen96] Hennessy, J.L. and D.A. Patterson, *Computer Architecture: a Quantitative Approach*, Morgan Kaufmann Publishers Inc., San Francisco, USA, 1996.

[Ian94] Iannucci, R.A. et al. (eds.), *Multithreaded Computer Architecture: a Summary of the State of the Art*, Kluwer Int. Series in Engineering and Computer Science, Vol. 281, Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1994.

[Jay02] Jayapala, M. et al., “Clustered L0 Buffer Organization for Low Energy Embedded Processors”, *Proc. 1st Workshop on Application Specific Processors (WASP)*, held in conjunction with MICRO-35, Istanbul, Turkey, 18-22 Nov. 2002.

[Jay03] Jayapala, M. et al., “Methods to Execute Multiple Loops in Parallel”, unfinished document. More information available from http://lesbos.esat.kuleuven.ac.be/~mjayapal/.

[Ku92] Ku, D.C. and G. De Micheli, *High-Level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer Academic Publishers, 1992.

[Lam88] Lam, M., “Software Pipelining: an Effective Scheduling Approach for VLIW Machines”, *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, Atlanta, Georgia, USA, 22-24 June 1988, pp. 318-328.

[Lap94] Lapsey, P. et al., *DSP Processor Fundamentals*, Berkeley Design Technology Inc. (BDTI), 1994. http://www.bdti.com/.

[Mar98] Marcuello, P. et al., ”Speculative Multithreaded Processors”, *Proc. Int. Conf. Supercomputers*, Melbourne, Australia, 13-17 July 1998, pp. 77-84. ACM Press, New York, USA, 1998.

[Mes01] Mesman, B., *Constraint Analysis for DSP Code Generation*, Ph.D. Thesis, Eindhoven University of Technology, Electrical Engineering Department, 2001.

[Mik96] Mikschl, A. and W. Damm, “MSparc: a multi-threaded Sparc”, *Lecture Notes in Computer Science*, Vol. 1123, pp. 461-468. Springer-Verlag, Heidelberg, Germany, 1996.

[Mos01] Moshovos, A. and G.S. Sohi, “Microarchitectural Innovations: Boosting Micro-processor Performance Beyond Semiconductor Technology Scaling”, *Proc. IEEE*, Vol. 89(11), Nov. 2001, pp. 1560-1575.

[Oeh99] Oehring, H. et al., “Simultaneous Multithreading and Multimedia”, *Workshop on Multithreaded Execution, Architecture and Compilation 1999 (MTEAC99)*, Orlando, USA, 9 Jan. 1999.

[Rot97] Rotenberg, E. et al., “Trace Processors”, *Proc. 30th Int. Symp. MICRO*, Research Triangle Parc, NC, USA, 1-3 Dec. 1997, pp. 138-148.
IEEE Computer Society Press, Los Alamitos, CA, USA, 1997.

[SiHive] Silicon Hive Technology Primer. Available from http://www.siliconhive.com

[Smi81] Smith, B.J., "Architecture and Applications of the HEP Multiprocessor Computer System", *Proc. SPIE Real-Time Signal Processing IV*, Vol. 298, San Diego, CA, USA, 25-28 Aug. 1981, pp. 241-248.

[Tul95] Tullsen, D.M. et al., "Simultaneous Multithreading: Maximizing on-chip Parallelism", *Proc. 22nd Annu. Int. Symp. Computer Architecture (ISCA22)*, Santa Margherita Ligure, Italy, 22-24 June 1995, pp. 392-403.

[Ung02] Ungerer, T. et al., "Multithreaded Processors", *Computer Journal (UK)*, Vol. 45(3), 2002, pp. 320-348. Oxford University Press for British Computer Society, UK, 2002.

[Ver95] Verhaegh, W.F.J., *Multidimensional Periodic Scheduling*, Ph.D. Thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, 1995.

[Vit67] Viterbi, A.J., "Error bounds for convolution codes and an asymptotically optimum decoding algorithm", *IEEE Trans. Information Technology*, Vol. IT-13, April 1967, pp. 260-269.

[Wal91] Wall, D.W., "Limits in Instruction-Level Parallelism", *Proc. Int. Conf. 4th Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV),* Santa Clara, CA, USA, 8-11 April 1991, pp. 176-188. ACM Press, NY, USA, 1991.

Glossary

ADSP : *Application Domain Specific Processor* – microprocessor consisting of application specific units (ASUs), designed for a specific application domain.

ASU : *Application Specific Unit* – functional unit designed to perform operations for a specific type of application.

Basic block: sequence of consecutive statements of a program in which flow of control enters at the beginning and leaves at the end without halt or the possibility of branching (jumping to another basic block due to a conditional statement), except at the end of the block.

CMP : *Chip Multi-Processor* – architectural design that integrates two or more complete processors onto a single chip to serve a multi-programming or multi-threaded workload.

CPI : *Cycles Per Instruction* – the number of clock cycles that the execution of an instruction consumes.

DFG : *Data Flow Graph* – representation of a basic block in the form of a graph.

DSP : *Digital Signal Processing* – refers to various techniques for improving the accuracy and reliability of digital communications.

FU : *Functional Unit* – part of a microprocessor that performs a specific function (e.g., add, multiply, add-compare-select). Also called processing unit (PU).

HDL : *Hardware Description Language* – kind of language used for the conceptual design of integrated circuits. Examples of such languages are VHDL and Verilog.

ILP : *Instruction-Level Parallelism* – independent instructions from one block of code.

II : *Initiation Interval* – the number of clock cycles after which the next iteration of the operations in a loop are started.

Latency: the number of clock cycles after which all operations in the DFG (and all iterations of the DFG) are finished.

PU : *Processing Unit* – part of a microprocessor that performs a specific function (e.g., add, multiply, add-compare-select). Also called functional unit (FU).

RTL : *Register-Transfer Level* – design level on which complex hardware is specified in terms of clocked elements, called registers, with operations in between. An operation on this level consists of fetching the necessary operands from a register, executing an operation on a FU and storing the result in registers.

RISC : *Reduced Instruction Set Processor* – a microprocessor that is designed to perform a smaller number of types of instructions so that it can operate at a higher speed (perform more millions of instructions per second, or MIPS). Since each instruction type that a processor must perform requires additional transistors and circuitry, a larger list or set of instructions tends to make the microprocessor more complicated and slower in operation.

Superscalar: the ability to fetch, issue and execute multiple operations from a single instruction stream during one clock cycle.

SMT : *Simultaneous Multi-Threading* – the superscalar instruction-issue combined with a multiple-context approach.

TLP : *Thread-Level Parallelism* – independent instructions from multiple blocks of code.

VLIW : *Very Long Instruction Word* – a microprocessor design technology. A processor based on the VLIW paradigm is capable of executing many operations within one clock cycle. Essentially, a compiler reduces program instructions into basic operations which the processor can perform simultaneously. The operations are put into a very long instruction word which the processor then takes apart and passes the operations off to the appropriate devices.

VLSI : *Very Large Scale Integration* – the current level of microcomputer miniaturization, which refers to microchips containing in the hundreds of thousands of transistors.

Appendices

A Simple Case Study using Silicon Hive

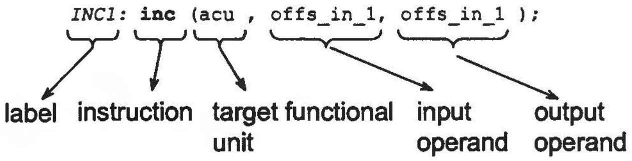
In this appendix, the algorithm description (A.1) and the architecture (A.2) used for the simple case study as discussed in Section 4.3 are presented.

A.1 Algorithm description

The following piece of code is the HDFC-description (intermediate description format of the Silicon Hive tools) for the C-code fragment that was presented in Figure 1-2.

```
int case_study (Cell me)
{
  START_FUNCTION
    BLOCK(body)
    {
      imm (psu, 1, ipg                      );
      imm (psu, 0, SPLIT(base_in, offs_in_1));
      imm (psu, 4, offs_in_2                );
      imm (psu, 0, SPLIT(base_out, offs_out));

  loop_i: for (loop_cnt_i = 3; loop_cnt_i != -1; --loop_cnt_i)
      {
     RCV: rcv  (sru , ipg, 6, tmp_in               );
     INP: inp  (iput, tmp_in, SPLIT(in_1, in_2)    );
     ST1: st8o (lsu , ipg, base_in, offs_in_1, in_1);
     PCS: pcs  (prc , in_2, tmp1                   );
     ST2: st8o (lsu , ipg, base_in, offs_in_2, tmp1);
    INC1: inc  (acu , offs_in_1, offs_in_1         );
    INC2: inc  (acu , offs_in_2, offs_in_2         );

      PIPELINING(0)
      }

  loop_j: for (loop_cnt_j = 7; loop_cnt_j != -1; --loop_cnt_j)
      {
      LD: ld8o (lsu , ipg, base_out, offs_out, tmp_out);        AFTER(ST1,1);
     OUT: outp (oput, tmp_out, out                    );
     SND: snd  (sru , ipg, 2, out                     );
    INC3: inc  (acu , offs_out, offs_out              );

      PIPELINING(0)
      }
  } END_FUNCTION
}
```

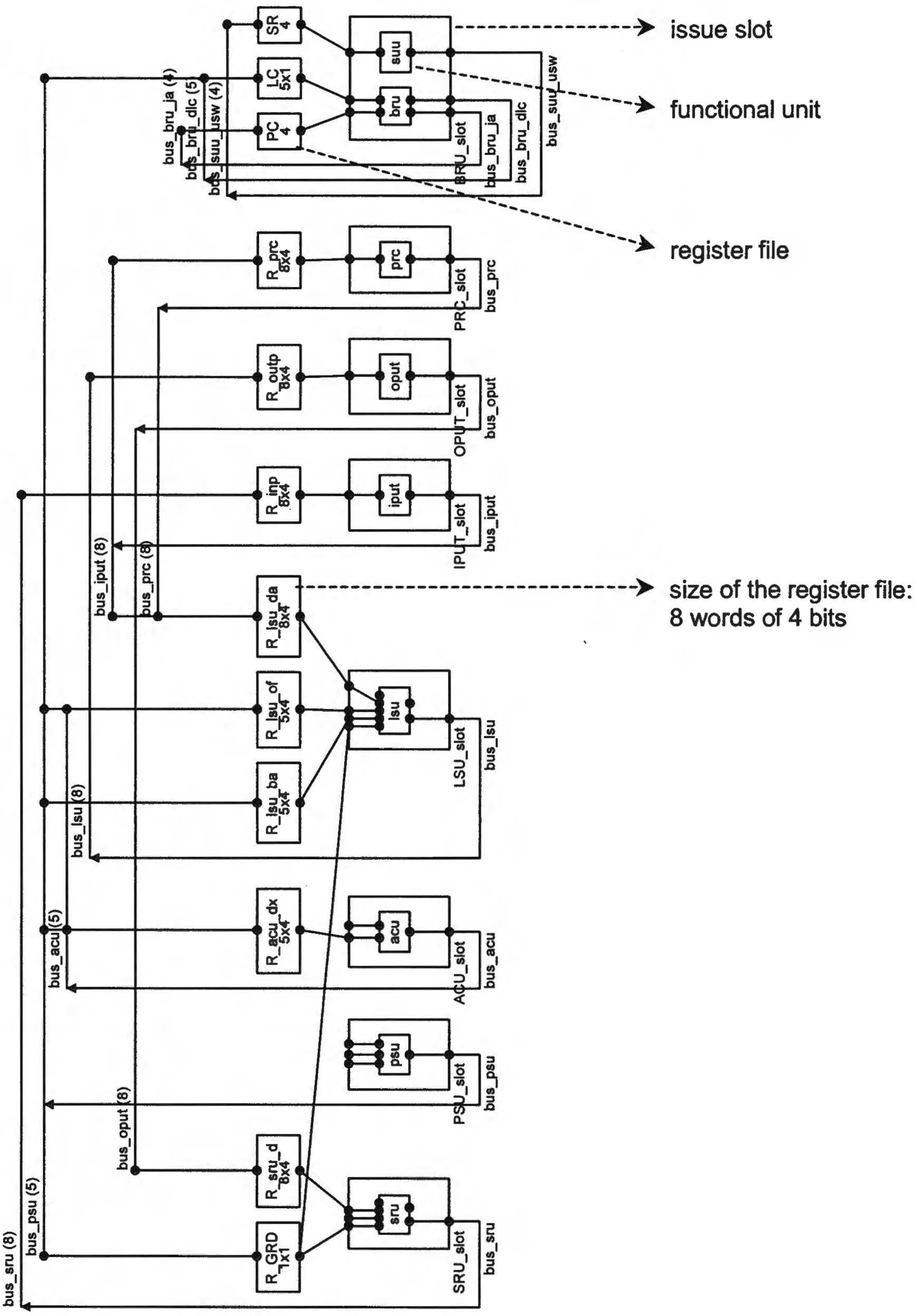
Legend for the HDFC-description:



The statement `PIPELINING(0)` instructs the scheduler to find the minimum possible initiation interval for the corresponding loop.

The statement `AFTER(ST1,1)` inserts a sequence precedence edge with weight one between the LD and ST1 operations.

A.2 Target architecture



B The Viterbi Algorithm

In this appendix, a pseudo-code representation of the Viterbi algorithm based on the sliding window execution is shown, as well as the schedules for a sequential and a loop-merged implementation.

B.1 Pseudo-code description of Viterbi algorithm

The following is the pseudo-code for the part of the algorithm in which the trellis construction and traceback phases are executed in parallel:

```
$t_0$ = 0;
while()
   tc_loop_t: for t = $t_0$+1...$t_0$+$D_{update}$-1
      tc_loop_s: for s = 0...S-1
         if (s < S/2)
            bm0 = hamm(inp[t], bc[prev(s,0)]);
            bm1 = hamm(inp[t], bc[prev(s,1)]);          } Branch metric calculation
         else
            bm0 = hamm(inp[t], 7-bc[prev(s,0)]);
            bm1 = hamm(inp[t], 7-bc[prev(s,1)]);
         end
         pm0 = pm[prev(s,0)] + bm0;                  } Path metric calculation (ADD)
         pm1 = pm[prev(s,1)] + bm1;
         if (pm0 < pm1)
            pm_next[s] = pm0;
            surv[t,s]  = 0;
         else                                        } Path metric selection
            pm_next[s] = pm1;                          (COMPARE-SELECT)
            surv[t,s]  = 1;
         end
      end
      for s = 0...S-1
         pm[s] = pm_next[s];
      end
   end

   s = 0;
   tb_loop_t: for t = $t_0$...$t_0$-$D_{trace}$+1
      if (s < S/2)
         push(0);
      else                                           } Traceback
         push(1);
      end
      s = prev(s, surv[t,s]);
   end
   tb_loop_out: for t = $t_0$...$t_0$+$D_{update}$-1
      pop();                                         } Output generation
   end
   $t_0$ = $t_0$+$D_{update}$;
end
```

In this code, S is the total number of states, D_{update} is the number of forward trellis construction steps and D_{trace} the number of traceback steps.

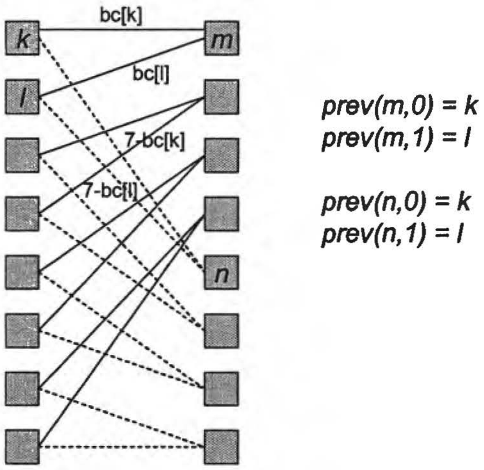
The function *hamm*(x,y) determines the Hamming distance between x and y: the number of bits in which x and y are different, assuming that x and y are binary variables.

The functions *push*(x) and *pop*() perform push and pop operations on the stack.

The function *prev*(x,y) is used to determine from what state the current state can be reached. Definition of this function is as follows:

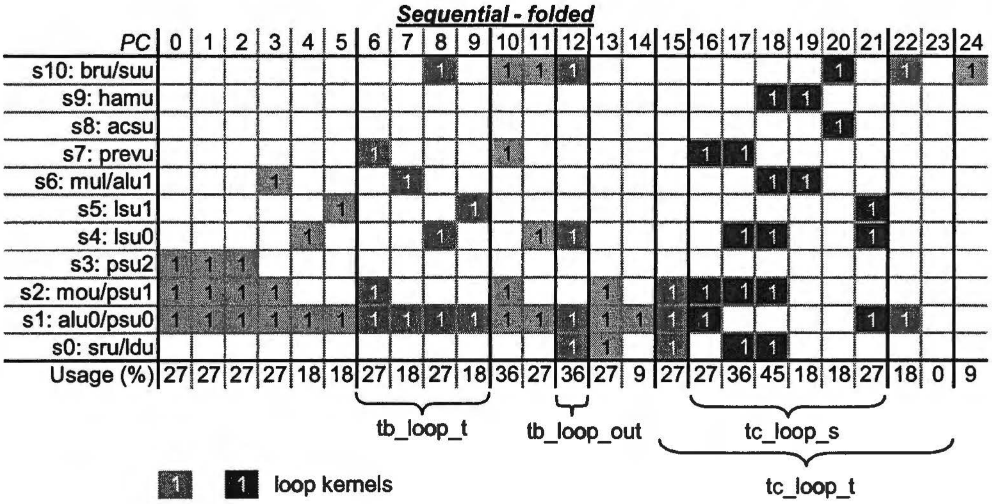
$$prev(x,y)=\begin{cases} 2x, & if\ x<\dfrac{S}{2}\ and\ y=0 \\ 2x+1, & if\ x<\dfrac{S}{2}\ and\ y=1 \\ 2x-S, & if\ x\geq\dfrac{S}{2}\ and\ y=0 \\ 2x-S+1, & if\ x\geq\dfrac{S}{2}\ and\ y=1 \end{cases} \tag{10}$$

The following is a graphical representation of this function:



B.2 Sequential execution of the Viterbi algorithm

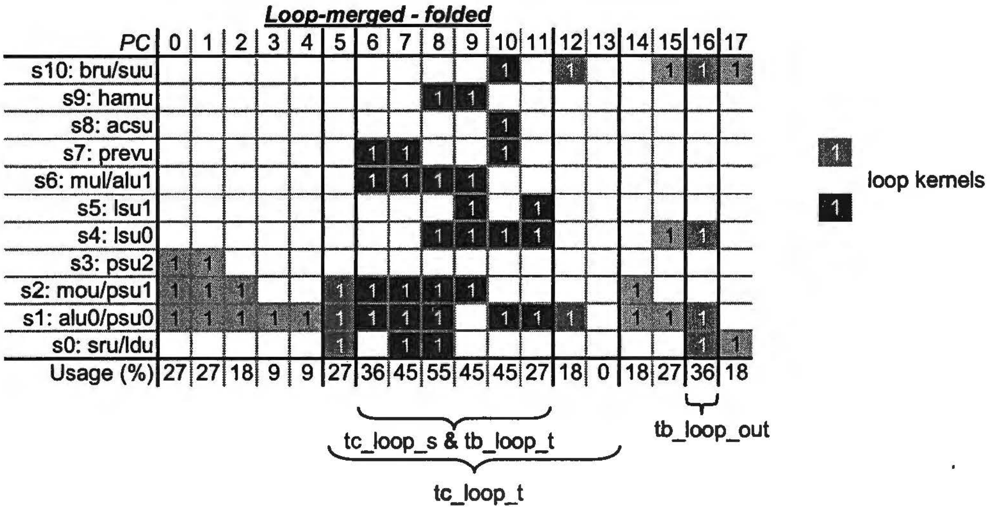
The following diagram shows the schedule for the sequential execution of the Viterbi algorithm on a VLIW data path with a single controller. The labels below the diagram correspond with the labels in the pseudo-code.



Sequential - folded

PC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
s10: bru/suu									1		1	1	1								1		1		1
s9: hamu																			1	1					
s8: acsu																					1				
s7: prevu							1				1						1	1							
s6: mul/alu1				1				1											1	1					
s5: lsu1						1				1												1			
s4: lsu0					1				1			1	1					1	1			1			
s3: psu2	1	1	1																						
s2: mou/psu1	1	1	1	1			1				1			1		1	1	1	1						
s1: alu0/psu0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1					1	1		
s0: sru/ldu													1	1		1		1	1						
Usage (%)	27	27	27	27	18	18	27	18	27	18	36	27	36	27	9	27	27	36	45	18	18	27	18	0	9

The following diagram shows the schedule for the loop-merged execution of the Viterbi algorithm on a VLIW data path with a single controller. Again, the labels below the diagram correspond with the labels in the pseudo-code.



Loop-merged - folded

PC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
s10: bru/suu											1		1			1	1	1
s9: hamu									1	1								
s8: acsu											1							
s7: prevu							1	1			1							
s6: mul/alu1							1	1	1	1								
s5: lsu1										1		1						
s4: lsu0									1	1	1	1				1	1	
s3: psu2	1	1																
s2: mou/psu1	1	1	1			1	1	1	1	1					1			
s1: alu0/psu0	1	1	1	1	1	1	1	1	1		1	1	1		1	1	1	
s0: sru/ldu						1		1	1								1	1
Usage (%)	27	27	18	9	9	27	36	45	55	45	45	27	18	0	18	27	36	18