Delft University of Technology
Faculty of Electrical Engineering

CARDIT
Department of Computer Architecture and Digital Systems

Comparative analysis between automatic design methodology and manual of an embedded system for MPEG Audio decoding

Supervisors: Prof. Ing. Alessandro de Gloria

Dr. Henk Corporaal

Author: Andrea Cilio

July 1996

Contents

Aknowledgments

I whish to express my gratitudine to professors Alessandro de Gloria and Hendrick Corporaal, the former for having given me the possibility to carry out this thesis, the latter for his constant help and availability demonstrated throughout its development.

I also thank the engineers Paul van der Arend and Marcel van der Laan of SPaSE B.V. for their precious collaboration and their cordiality. They made possible and more interesting my study.

My thanks go also to the people who have contributed with their technical support and their suggestions: Bert Meijs, Tobias Nijweide, Fabio Ancarani, Alessandro Uber, Theo Baan and Erwin Abrahamse. Thanks also to professor Stamatis Vassiliadis and the engineer Sorin Cotofana for their encouragement and their pleasantness.

Thanks finally to my parents, for their support, and to all my dearest friend who I met in Delft during these months of study, or that have been close to me: Marisa Balas, Massimo Marenzana, Joao Paulo Santos, Anna Grau, Ilian Ilkov, Mario Nunes, Hårward Brandt, Magda Cavanna, Enrico Bazan and Andrea Carbone. Without them this period of study would have not been such a pleasing experience.

Genoa, 11 July 1996

1 Introduction

In the last six years, the Computer Architecture Design research group of Delft has been developing an automated design framework based on a new architecture paradigm called *Transport Triggered Architecture* (TTA). The concepts behind TTAs were developed in the same research group, and proved themselves to be especially suited for the Application Specific Processors (ASPs) synthesis. This design framework under development, called *MOVE framework*, consists of three main parts:

1. *Optimizer*, which is responsible for searching the design space.
2. *Hardware subsystem*, generating the processor layout.
3. *Software subsystem*, generating the instruction level parallel code.

Both the generated code and the architecture configuration achievable by the MOVE framework show a remarkable performance gap when compared to manually designed ASPs and their code for the same application.

The main theme of this thesis is the analysis of the code generated by the instruction level parallel compiler of the MOVE software subsystem to research the causes of compiled code inefficiency. This analysis is based on the comparison between a hand-crafted solution for an ASP application and the solution that can be achieved by the MOVE framework for the same application. We selected MPEG-1 Audio decoding as application and a custom ASP exploiting the TTA paradigm as reference for the comparison. The documentation of the ASP was made available by the company that developed the processor, SPaSE B.V., Nijmegen. The final goal of this research is to propose modifications to the MOVE framework in order to reduce the gap of performance between custom designed and automatic generated solutions.

The scope of this research includes therefore parallel compilation techniques, hardware and software automatic cosynthesis and, as test application, MPEG audio compression. This chapter gives a brief introduction to the basic concepts of ASPs and Transport Triggered Architectures in section 1.1. In section 1.2 the problem to be solved is clearly stated, and its motivations are presented. Section 1.3 summarizes the results of our analysis and lists possible solutions and recommendations for the further development of the framework. Section 1.4 concludes this chapter with an overview of the remainder of this thesis.

1.1 Research background

Application Specific Processors represent a huge part of the microprocessor market. Their technology is highly pervasive: billions of microprocessors show up annually in all kinds of equipment used in everyday life. These processors are called embedded because they are often embedded within special purpose systems. Being designed and optimized for a particular application, as for full custom designs, ASPs allow a better performance to cost ratio than applications based on general purpose processors, especially for large production volumes.

The major drawback of a full custom design for ASPs is the design time. Nowadays some cost factors, like die area and power consumption, are less stringent for a large set of applications, as the integration scale increases. What becomes vital instead, especially in a competitive and rapidly changing market as the one of ASPs, is the time-to-market, the period of time needed for the design process, which also contributes to the largest part of ASP fixed (i.e. once per design) costs.

A method to reduce design time and costs is to build ASPs according to a template, giving up some design freedom. In this case, a single solution is the instantiation of the architecture parameters of the template. The success of automated design tools is based on their efficiency for design space exploration and for generating the processor and the code for the processor. This largely depends on the templated architecture utilized, which must satisfy the following requirements: scalability, programmability, flexibility, and modularity. Scalability and flexibility are needed to cover a wide range of solutions; programmability is needed to handle any application written in a common high level language; modularity is a basic requisite needed for efficient and automatic synthesis.

For their properties, TTAs are a good candidate for template based design tools for ASPs. Like the standard Operation Triggered Architectures (OTAs), the basic components of TTAs are function units (FUs), register files (RFs), and an interconnection network of busses. The difference between the two architecture paradigms consists in the way they are programmed:

> the instructions of an OTA specify the operations that trigger operations on FUs explicitly, and trigger data transports between FUs and RFs implicitly (as a side-effect),

while

> the instructions of a TTA specify the data transports that trigger data transports between FUs and RFs explicitly, and trigger operations on FUs implicitly (as a side-effect).

By programming data transports instead of operations the complexity of the control hardware is reduced, because the processor is relieved from the task of scheduling data transports for operations. TTAs show thus a simpler hardware structure, which allows better modularity and scalability. The explicit data transport scheduling by the compiler introduces also new possibilities for code optimizations, and make the transport capacity between FUs and RFs an independent parameter of the processor. It has been demonstrated [4] that a large part of the data transports and RF accesses performed by OTA operations are unnecessary. The inter FU and RF connectivity can therefore be substantially reduced without incurring in any performance penalty. The same applies to the number of registers. On the contrary, OTAs scalability is limited by the required connectivity between RFs and FUs, which rapidly increases with the number of such units.

The major drawback of TTAs is the complexity of programming: manual writing of TTA code is, although not impossible, a difficult and error prone task, especially for large configurations. The complexity of the compiler is consequently higher than for OTAs: a consistent part of the processor task, i.e. data transports scheduling, is now charged to the code generator; many new optimizations have to be taken into account and exploited. For example, the software bypass of a FU result into another FU input register must be detected and the unused write-backs into RFs must be suppressed. The presence of irregularly, partially connected networks presents another challenging aspect of scheduling for TTAs. The scheduler is responsible for routing the data over the available paths through the interconnection network and making maximal use of the available transport bandwidth.

1.2 Motivations

The final aim of the research performed around the MOVE software framework is to investigate code generation techniques for TTAs and to fill up the performance gap existing between code for custom designed and for synthesized ASPs.

The current MOVE Framework was first developed to demonstrate the feasibility of a compiler for TTAs. This task proved to be far from trivial, but succeeded: though lacking the tools for the final assembly of the machine instruction vectors from parallel code, the software design trajectory of the framework can be said complete and operative.

Still many limitations are present in the current MOVE software design trajectory and in the tools for code generation. Perhaps the major limitations, as pointed out throughout this report, are due to the front-end adopted for the first compilation stage from C or C++ source files, which is the Free Software Foundation GNU compiler (called *gcc*). Even though it is a good quality, widely supported and tested compiler, gcc produces intermediate code strongly oriented to typical workstation architectures, which does not fully match ASP application requirements. Some of the characteristics of the intermediate code turn out to limit the architecture template:

1. Data word width and address space are fixed to 32 bits.

2. Memory partitioning is limited to a single memory bank for data and one bank for instructions. No partitioning of data into multiple, user definable memories is possible.

3. Automatic variables are always dynamically allocated: it is not possible to allocate temporary variables into a static stack.

4. Memory is addressed by 8 bit bytes [30].

In addition to these limitations, some parts of the back-end, the MOVE instruction scheduler, present room for improvements which should significantly reflect on compiled code performance:

1. Better memory disambiguation algorithm.
2. Enhancement of the optimizations dependent on data flow analysis.
3. Introduction of automatically incremented addressing modes with prefetch.
4. Better FU assignment algorithm.

One of the obvious ways to point out the most promising extensions is to perform a detailed comparison between a generated code and a manually designed one for a custom ASP application. A large and important subset of ASP applications is the Digital Signal Processing. The MPEG Audio decoding belongs to this subset and was chosen as a test application because a custom version based on TTA paradigm was available for the comparison.

A detailed analysis of execution profiling data is necessary to expose the main causes of inefficiency of compiled code. The main risk of such analysis is to give a general relevance to the results valid for a specific application and to point out possible extensions of the framework that are unlikely to give significant improvements on most of applications.

Still, the ultimate proof for establishing if an extension is 'paying itself' is to implement it and determine when this new feature is applicable on the code and what is the improvement it can bring. This motivates the test implementation of one of the instructions found in the custom ASP, which proved to give a substantial contribution to the application performance.

1.3 Results

The main purpose of this thesis was to carry out a detailed, quantitative analysis of the performance differences between a custom designed ASP application and an equivalent solution synthesized by means of the MOVE Framework. First, the unavoidable architectural differences between the custom ASP and the 'closest' configuration possible using the MOVE template are discussed. With this preliminary step the limitations of the current template are clarified. The benefits expected from removing each of these limitations are also discussed.

The result of simulating the MOVE version and the custom one of the test application are then compared. The data available from simulation include cycle and operation counts, hardware utilization, and other profiling information. In order to achieve extended and accurate measurements, some modifications were applied both to the MOVE parallel simulator and the simulator for the assembly code developed at SPaSE.

The main figures obtained by the simulation were largely expected: the program size of the generated solution shows a factor ranging between 1.4 and 1.7 with respect of the custom design, while the execution cycle count retio rests between 2.3 and 2.5, depending on the scheduling parameters suplied to the MOVE scheduler. The simulation analysis is followed by a detailed comparison of the code of the critical (i.e. responsible for most of the execution time) parts of the application. As result of this comparison, many causes of inefficiency were found. For each of them, the contribution to the performance gap was estimated.

The subsequent step was to propose a set of changes to improve the compiled code and to reduce the performance gap. For each proposal a first study of implementation complexity and of changes required to the MOVE Framework is presented. Finally, two interesting extensions were studied in detail: the implementation of multiple data width definable by the user and the mapping of loop with a fixed iteration count into hardware loop instruction.

1.4 Overview of the rest of report

The remainder of this thesis is organized into 8 chapters as follows. Chapter 2 introduces TTAs and describes the MOVE Framework, with particular detail on the software subsystem. Chapter 3 describes the test application, MPEG Audio decoding. Only the two compression classes implemented, namely Layer I and Layer II, are treated in detail. Chapter 4 presents the implementation of the audio decoder manually designed at SPaSE B.V., that was chosen for the comparison. This solution is used as a reference both for the code design and the hardware selection for the audio decoder developed by means of the MOVE Framework, described in chapter 5. Chapter 6 describes the results from comparing the two architectures, presents the sources of inefficiencies found in the synthesized version and proposes a set of recommendations to reduce the measured gap of performance. Chapter 7 describes the partial implementation of one of these recommendations, i.e. a hardware loop instruction, and presents the results of the first tests performed on it. Chapter 8 analyzes in detail the problem of the user-definable data sizes and discusses possible approaches for its solution. A test implementation based on C++ classes is also discussed. Chapter 9 concludes this thesis with a summary of the contributions and suggests possible extensions of the work presented.

TTAs and the MOVE Framework

2

Due to the decreasing feature size of VLSI technology, the amount of hardware which can be integrated into a single chip increases. As a result, future processor chips may execute tens of operations concurrently. Many applications can profit from these huge amounts of hardware parallelism by designing an application specific instruction set processors (ASIP). Two problems emerge however: (1) the design space of ASIPs is very large; it is difficult to chose a satisfactory solution, and (2) the design complexity increases and therefore design cycle gets too long.

The alleviate these problems a design trajectory based on a *templated, transport triggered architecture* (TTA) has been developed. Using a restricted, but still very large, design space it is possible to automate the design trajectory based on a quantitative analysis of many design points. A key aspect of TTAs is the reduction of the on-chip data transport requirements; this may result in a better cost-performance ratio of the realized ASIPs. In this chapter we discuss an automated design process for ASIPs using the *MOVE framework*. The chapter is structured as follows: section 2.1 explains the MOVE framework, and discusses briefly how TTAs operate. Section 2.2 lists the major parameters of the architecture design space and shows how this space is explored. Finally, in section 2.3 several conclusions are drawn and a comparison with related work is made.

2.1 The MOVE framework

Designing ASICs based on templated application specific instruction set processors (ASIPs) is an attractive solution that offers flexibility and a short design time while still having most of the advantages of ASICs. The design process consists of finding the right architecture parameter values for the given application, such as the operation set, the amount of instruction level parallelism, and the sizes of the register files. The quality of a solution depends on the offered performance and the implementation costs.

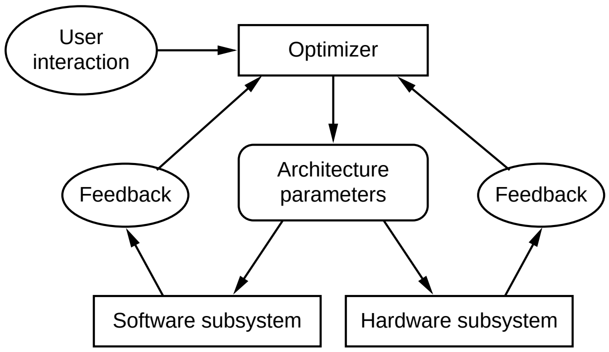


Figure 2.1: MOVE framework overview.

The synthesis framework presented in this section uses an architecture design space based on a *transport triggered architecture*, or TTA. This architecture is of the *instruction level parallel* type; it resembles the well known VLIW architectures. However, a key difference is that TTAs are programmed by specifying data transports instead of operations. This gives an extra level of control to the code generator, and allows for a more efficient use of hardware resources. Although we use a TTA template for designing ASIPs, the design space is still very large. Picking a proper solution (for a specific application) from this design space requires a quantitative analysis of many design points. This search process must largely be automated in order to reduce the design time. Therefore tools are needed, not only for making the quantitative analysis of hardware and software (generated code), but also for the automated search.

The MOVE framework consists of a set of tools for hardware and software synthesis. Within the synthesis process we use an *architecture template*, i.e. processors are built according to the pattern of a TTA. A specific TTA is defined by a set of architecture parameters, like the number and type of function units, the number of register files and registers, etc. At first sight this suggests that we restrict ourselves and therefore obtain inferior solutions. In practice however, several advantages emerge. Firstly, the template building blocks are pre-designed and can therefore be made very efficient, both in area and performance. Secondly, the architecture pipelining is worked out very carefully, alleviating many timing bottlenecks; prototype realizations learned where these bottlenecks exactly are. Finally, usage of a clearly defined design space allows the design of synthesis and evaluation tools, which not only generate a combined hardware and software solution, but also allow a quantitative analysis of the design space. Note, that the template still covers a very large design space.

The synthesis of hardware and software for a given application is done using the MOVE framework; this framework produces both the layout of an ASIP and the corresponding object code to be executed on this ASIP. An overview of this framework is shown in figure 2.1. It basicly consists of three components:

1. **Optimizer** which is responsible for searching the design space and the interaction with the designer. It determines the configuration (i.e., the set of architecture parameters) to be evaluated next.

2. **Hardware subsystem** generating processor layout, and giving information on timing, area, and power consumption.

3. **Software subsystem** generating instruction level parallel code, and giving statistical information on usage of hardware resources.

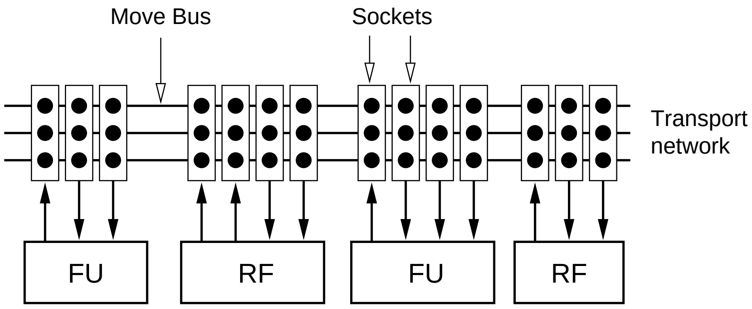


Figure 2.2: General structure of a TTA.

These components are detailed in following subsections. Before, we briefly describe how TTAs operate.

2.1.1 Transport triggered architectures

TTAs can be compared to VLIW architectures; their instructions are horizontally encoded; i.e. each instruction has a number of fields. Whereas fields for VLIWs specify RISC like operations, for TTAs they specify the required data transports. These transports may trigger operations as side effect. Programming transports adds an extra level of control to the code generator, and enables new optimizations; in particular, it allows us to get rid of many superfluous data transports to and from the register files and to reduce the on-chip connectivity[11].

A compiler views a TTA as a collection of function units (FUs), register files (RFs), *move buses*, and *sockets*; see figure 2.2. FUs perform operations, RFs provide temporary fast accessible storage, the network of move buses performs data transports between the FUs and RFs, and sockets interface FUs and RFs to move buses. Normally, each socket is connected to a different FU input/output or RF port.

To illustrate TTA programming, consider the following three operations of an operation triggered machine, or OTA:

```
add  r1, r2, r3  /* r1 = r2 + r3 */
sub  r4, r2, r6  /* r4 = r2 - r6 */
st   r4, r1      /* store r4 at address r1 */
```

These operations can be translated into the following two TTA instructions:

```
r2->add_o,   r3->add_t,    r2->sub_o; r6->sub_t;
add_r->st_t, sub_r->st_o;
```

In the first instruction the four operands of the add and subtract operations are moved from the RF(s) to the FU inputs of the FUs that perform the two operations. In the second instruction the results of the add and subtract operations are moved from the FUs that performed them to the FU that performs the store operation. From this small example we already observe a few advantages of TTAs. The results of the add and subtract operations are not written back to the RF and the operands of the store operation are not read from the RF. The former saves RF write accesses and data transports, the latter saves RF read accesses. Since TTAs do not couple move buses and RF ports directly to FUs, as is the case for many VLIW and superscalar architectures, the freed resources can be used for other operations. This makes that TTAs have a better hardware utilization, which implies less hardware for the same performance or more performance with the same hardware [11].

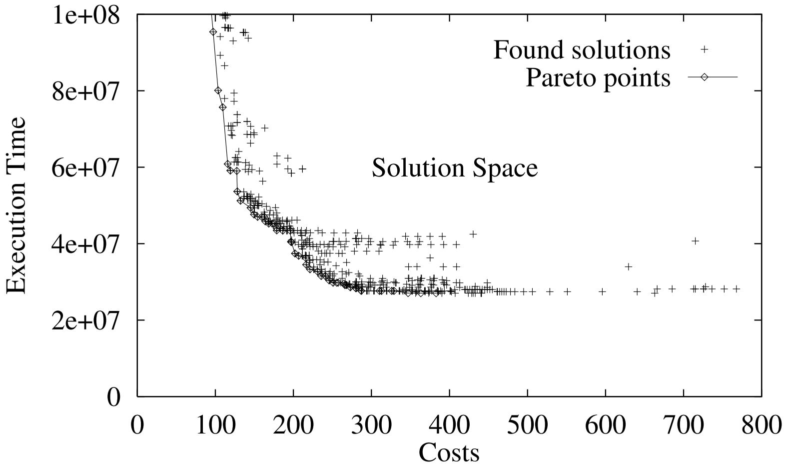


Figure 2.3: Possible solutions and Pareto points.

The interconnection network may be fully connected, in which case every socket is connected to all move buses, or partially connected. A fully connected interconnection network simplifies the code generation task, but it likely results in a high bus load on the move buses which affects the achievable cycle time. Therefore, in practice the interconnection network will be partially connected and the compiler is responsible to use the available connections as well as possible.

Besides executing operations on data, TTAs need to provide immediate operands, conditional execution, and control flow changing operations. Details on these issues can be found in [5, 8].

2.1.2 Optimizer

Two main design evaluation criteria are cost and performance, where performance is defined as the inverse of execution time. Costs may include the amount of chip area, number of pins, power dissipation, and code size[1]. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time. The *solution space* is given by all possible design points in the 2-dimensional cost-performance space. Figure 2.3 shows many generated solutions for a test application (described in [7]). As shown, the solution space is bounded by a curve connecting so called *Pareto points*. These points are local optimal solutions; they are defined by [2]:

Definition: *A point* (t_p, c_p) *is a Pareto point if*
$\forall(t, c) \in$ *Solution-Space*, $(t \geq t_p \vee c \geq t_c)$.

The optimizer finds its way through this search space by iteratively trying different architecture solutions, and letting the software and the hardware subsystems produce relevant information about these solutions, like cycle time, costs and number of cycles needed to run the application. Based on this information a next design point is chosen by updating the parameters. The initial architecture parameter values can be chosen freely by the user. He can also specify an evaluation function (e.g. minimize the product of costs and execution time), and the stop criteria.

[1] Currently are included area and pins only.

2.1.3 Hardware subsystem

The hardware subsystem of the MOVE framework is responsible for the realization of an application specific TTA in silicon. It accepts architecture parameter values, technology information and a cell library as input, and produces a VLSI layout (e.g. in CIF format) of the generated processor as output. Figure 2.4 shows its organization.

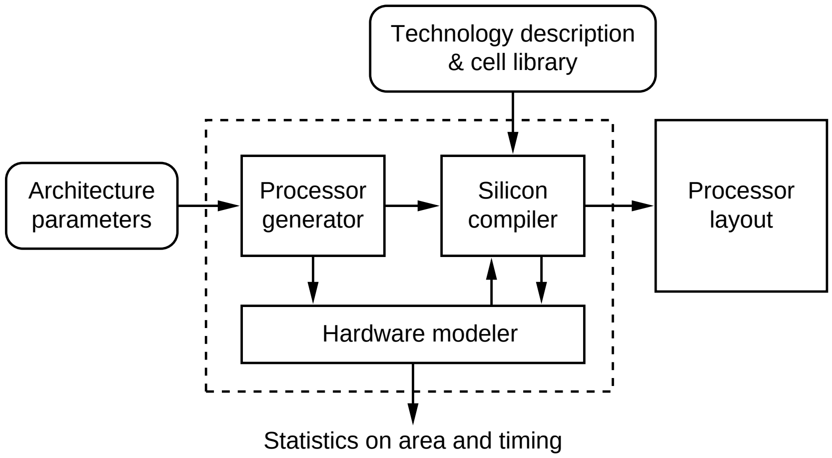


Figure 2.4: The hardware subsystem.

The hardware subsystem makes use of the ASA silicon compiler[2], or, alternative relies on Synopsys and Cadence tools. Details of this system and realized processors are described in [25].

The design space explorer makes use of a *hardware model* to estimate the cost of design points. This model is based on realizations in a CMOS 1.0μ technology. The values and expressions used for costs (area, including number of pins) are shown in table 2.1. The costs of FUs are based on a 32-bit data path width[3], and relative to an integer FU.

The minimum clock cycle time for a TTA realization is largely determined by the time needed to perform (and control) data transport. This depends on the number of connections to a bus. It is estimated to be:

$$T_{clock-cycle} = \max\{20, \max_{\forall \text{buses } b}\{12 + 0.4 \times C_b\}\}ns$$

Its minimum value is taken 20 ns, the time to do an integer operation; C_b is the number of connections on bus b.

2.1.4 Software subsystem

The software subsystem is detailed in figure 2.5. It provides the user with three main tools to develop code for TTAs. These tools are:

1. A compiler (referred to also as MOVE front-end) to translate HLL (high level language) code to sequential move code
2. A scheduler (or MOVE back-end) to schedule the sequential code and produce parallel code for a target TTA

[2] ASA is a trademark of SAGANTEC Europe b.v., Eindhoven.

[3] Although the hardware subsystem can generate processors for any data width, the software subsystem currently requires 32 bit integers.

Parameter	**Value** or **Expression**
Integer unit	1
Full shifter	3
Logic unit	1
Ifetch unit	0.5
Load-store unit	11.0
Multiplier (fixed point)	6
Divider (fixed point)	6
Compare unit (fixed point)	1
ALU (including multiply)	10
Floating point unit	20
Other Pins	4
Buses	$3.5 + W/16$
Registers	$0.0125N \times W + 0.0031 \times RP$
Sockets	$0.02 + 0.0125 \times W \times C$

Table 2.1: Example values and expressions of the hardware cost model, as used in [7]. Load-store unit cost includes pads. Bus cost includes pads for instruction fetch and on-chip wiring; W equal the bus width. RP is equal to number of register file ports; N the number of registers. C equal the number of connections.

3. A simulator and analyzer to verify and evaluate both the sequential and parallel code.

The software subsystem accepts any application coded in C or C++ and translates it into text representation of MOVE parallel code for a specific TTA. The components of the software subsystem are described in detail below.

Compiler

The MOVE front-end is a combination of three tools: proper compiler, assembler and linker. Their relationship is depicted in figure 2.6. In order to be assured of good code quality, good HLL compatibility, support for new HLLs and an extensively debugged compiler, a port of

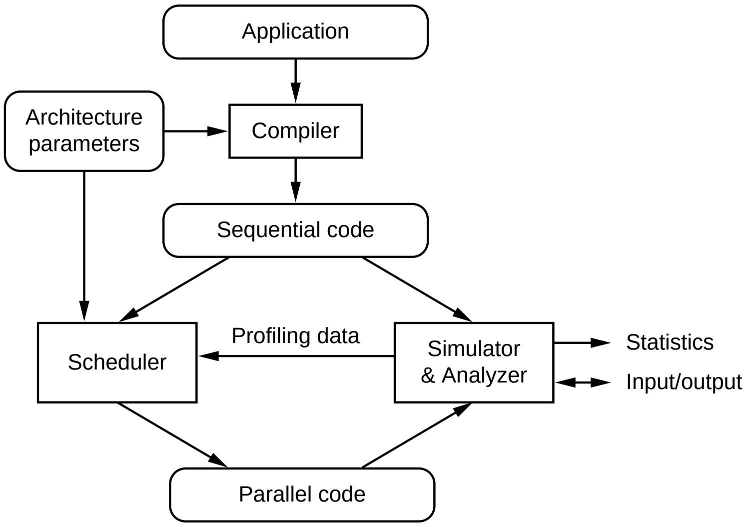


Figure 2.5: The software subsystem.

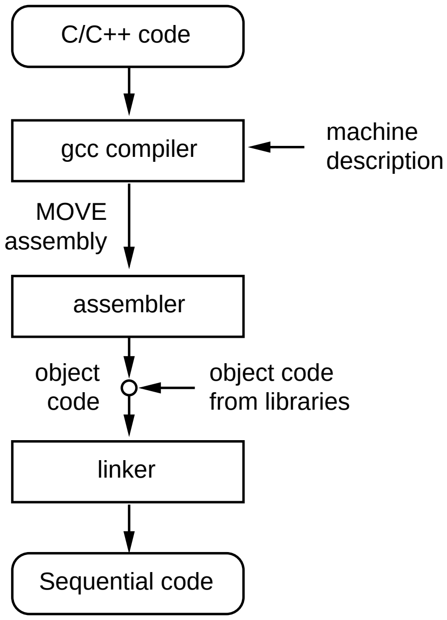


Figure 2.6: Relations between the ported GNU compiler, the assembler and the linker.

GNU C compiler (*gcc*), assembler, and linker was made. These software packages are ported to produce binary sequential MOVE code for a *MOVE generic machine*. This code is sequentially ordered by instructions; each move referring to the same operation is grouped in a single instruction, resembling OTA instructions. Sequential code is used as intermediate representation of the program and is read by the scheduler.

Scheduler

The scheduler is the most important part of the software subsystem. Its main function is to schedule moves of sequential code, i.e. to assign FUs to operations and to assign cycles, sockets and buses to moves. The scheduler has to generate instruction level parallel code, while exploiting all the available hardware resources. To this purpose, the scheduler uses profiling data (like execution frequencies) from the simulator. Several preliminary optimizations on sequential code are also applied. The scheduler uses advanced techniques like extended basic block scheduling, software pipelining, and speculative execution, in order to enhance code motions and consequently inter basic block parallelism [10]. All specific optimizations of TTAs (result bypassing, dead moves elimination, operand sharing) are performed during the scheduling process. The parallel code is fully parameterized on the template configuration, which is specified in a machine description file. In this file processor resources, like supported FUs, amount of registers and interconnection network are described. Profiling information is not strictly necessary, but helps the scheduler to work more efficiently.

Simulator

The simulator accepts either sequential MOVE code or parallel MOVE code. Its output consists of profiling information, application output and execution statistics. The simulator has three purposes in the MOVE framework:

1. To verify the compiler and the scheduler. It is virtually impossible to port a compiler and write a scheduler without simulating the produced code.

2. To evaluate architecture parameters. The results of the evaluation are cycle counts and various statistics about resource utilization and compilation events (e.g. the number of operand swaps and the number of loop scheduled using software pipelining).

3. To provide profiling data to the scheduler. Profiling data consist of execution counts for each basic block and each control flow edge between basic blocks in the program. With this information the scheduler can decide which code motion between basic blocks is most profitable.

2.2 Design space exploration

The main parameters of the template architecture are given in table 2.2. The number of sockets needed (see figure 2.2) is implied by the number and type of FUs, and the by the used register files. Connections between sockets and buses are specified individually.

Parameter	Range
#buses	≥ 1
#FUs	≥ 1
FU latencies	≥ 1
FU type	all library supported types
FU pipelining method	virtual-time or hybrid[5]
#registers, IRF	≥ 1
#input ports, IRF	≥ 1
#output ports, IRF	≥ 1
#registers, FRF	≥ 0
#input ports, FRF	≥ 1
#output ports, FRF	≥ 1
#registers, BRF	≥ 1
#input ports, BRF	≥ 1
#long immediates	≥ 1
bus socket connection	yes, no

Table 2.2: Main parameters of the architecture design space. Three kinds of RFs are available: integer register files (IRF), boolean register files (BRF) and floating point register files (FRF).

We divide the problem of finding a good application specific TTA into two subtasks, resource optimization and connectivity optimization. Resource optimization consists of finding the right match of resources, such as which FUs to use, how many buses, how many registers, and how many ports on the register files. The goal of resource optimization is to reduce costs. Connectivity optimization determines the connectivity between the buses and the sockets. The resulting connectivity is based on the communication requirements between the FUs and the register files. The primary goal of connectivity optimization is to reduce the bus load and thus the cycle time.

2.2.1 Resource Optimization

The optimizer of the MOVE framework wants to present the curve of Pareto points to the designer so that he can decide which solution matches his requirements. It does that by means of a local search algorithm.

The local search algorithm is described in figure 2.7. It starts with an initial configuration that corresponds with a oversized TTA, e.g. one with 64 registers, 16 buses, 4 load-store FUs, 4 adders, etc., and full connectivity between the buses and the sockets. From this configuration we go to a *neighbor* point by removing one resource. We repeat this until we reach a minimum configuration which is needed to compile the application successfully. Which resource is removed is determined by the following quality function that evaluates a design space point:

$$Quality = \frac{1}{ExecutionTime^{\alpha}.Costs^{\beta}}$$

where α and β are constants which express the relative importance of performance and costs. Since connectivity optimization, which takes care of the bus load, is performed after resource optimization resource optimization ignores the influence of the high bus load on the execution time.

After the local search algorithm has reached the minimum configuration we reverse the process and add resources until we reach the point where we started the search process. With this sweep back we hope to find new Pareto points. We make a few remove/add-sweeps with different values for the α and β constants.

During the remove-sweep we want to stay as close as possible to the cost axis. This is achieved by $\beta > \alpha$. Similarly, during the add-sweep we want to stay close to the execution time axis which is achieved by $\alpha > \beta$. The current version of the optimizer makes default five remove/add sweeps with α and β values between one and three. The set of used α and β values was obtained by means of experimentation. Adding more pairs to it did not lead to finding significantly more Pareto points, it only increased the exploration time.

The resources that are considered by the explorer are currently: (1) FUs, (2) move buses, (3) registers (integer, floating point, and boolean), and (4) RF ports. Removing a move bus from a configuration is done incrementally by reducing its width in steps from 64 bits to 32 bits to 1 bit to 0 bits. Adding move buses proceeds in the other direction. This can lead to configurations such as two 64-bit buses for floating point numbers, six 32-bit buses for integers, and one 1-bit bus for booleans. Registers are removed/added in groups such that the number of registers per RF is always 2^n or $2^n + 2^{n+1}, n \geq 0$. This is because the difference in cost and performance of, for instance, a 32 and a 31 integer RF configuration is very small.

The exploration time of the algorithm shown in figure 2.7 can be fairly long, especially if the initial configuration is very large and the evaluation time of a configuration is long. Several methods to reduce the exploration time are applied:

1. Evaluating the configuration only for the most critical procedures of the application. For example, the set of procedures where 99% of the execution time will be spent. This often excludes possibly long procedures which are seldomly executed.

2. Using a rule-based hardware model, as discussed in section 2.1.3 instead of a processor generator to estimate cycle time and costs.

3. Removing resources from the initial configuration that have no effect on the cycle count before the actual exploration begins. These resources will not be added during the extend-sweep.

Procedure ResourceOptimization($init\_config$)
  $curr$ = $init\_config$
  for each $\alpha \in \{1, 1.5, 2, 2.5, 3\}$ /* quality parameters */
    repeat /* reduce sweep */
      $neig = \{n \in \mathcal{P}(init\_config) \mid n = curr - \{r_i\},\ r_i \in curr,$ IsValid$(n)\}$
      $cand = \{n \in neig \mid \forall n' \in neig,$ Quality$(n, 1, \alpha) \geq$ Quality$(n', 1, \alpha)\}$
      if $cand \neq \emptyset$
        $curr$ = Select($cand$)
    until $cand = \emptyset$
    repeat /* extend sweep */
      $neig = \{n \in \mathcal{P}(init\_config) \mid n = curr \cup \{r_i\},\ r_i \notin curr\}$
      $cand = \{n \in neig \mid \forall n' \in neig,$ Quality$(n, \alpha, 1) \geq$ Quality$(n', \alpha, 1)\}$
      if $cand \neq \emptyset$
        $curr$ = Select($cand$)
    until $cand = \emptyset$
  endfor

Procedure Quality($config, \alpha, \beta$)
  return Costs($config$)$^{-\alpha}\cdot$ CycleCount($config$)$^{-\beta}\cdot$ CycleTime($config$)$^{-\beta}$

Figure 2.7: The algorithm for resource optimization. A configuration is represented as a set of resources r_i. $\mathcal{P}(init\_config)$ represents all subsets of the initial configuration. IsValid(c) determines whether configuration c has sufficient resources to compile the application successfully. Select(s) returns a member of s, either using a first fit or best fit approach. Costs(c), CycleCount(c), and CycleTime(c) return the costs, cycle count, and cycle time, respectively, by invoking the compiler back-end and hardware modeler.

4. Memorizing all configurations together with their evaluation results in a hash table to prevent unnecessary evaluations.

5. Selecting a neighbor configuration immediately if it has a better quality than the current configuration instead of evaluating all neighbor configurations and selecting the best one. We call this *first fit* instead of *best fit* exploration.

2.2.2 Connectivity Optimization

Connectivity optimization transforms the fully connected configuration found by resource optimization into a partially connected configuration that has less load on the buses and therefore a shorter cycle time. This is accomplished by removing bus socket connections from the buses in a round robin fashion. The bus socket connection that is removed from a bus is the first connection that has no influence on the cycle count. If no such connection exists we take the connection with the lowest influence on the cycle count. We repeat this process until the cycle time remains constant (limited by other factors than bus load) and the cycle count starts to increase.

By removing connections in a round robin fashion we balance the bus load, i.e., all buses have approximately the same number of connections.

2.3 Conclusions and related work

In this chapter we showed an automated design trajectory for ASIPs based on transport triggered architectures. This trajectory has two fundamental capabilities:

1. It maps arbitrary applications, written in C/C++, into a combination of hardware and software.

2. It offers the possibility to do a quantitative analysis of large parts of the design space.

The search process to find a proper solution consists of resource and connectivity optimization. Resource optimization attempts to find the cost effective set of resources. Connectivity optimization reduces the connectivity in order to reduce bus load and cycle time. As side effect it was shown that the synthesized TTAs require far less connectivity and fewer register ports than more traditional instruction level parallel architectures.

Related work
The synthesis process used within the MOVE framework has much in common with high level synthesis (HLS) systems which have been developed (see e.g. [3, 9, 27, 34]); they both aim at the automatic mapping of a functional specification into a structural description. There are several differences, though:

1. Most HLS systems do not accept 'full blown' applications written in commonly used imperative languages. Usually their usage is restricted to input from small applications written in a special language [33], containing only one or a limited number of functions.

2. The MOVE framework limits the design space by using a templated TTA. As indicated in section 2.1 this has several advantages. It also does not make sense to include design points within the design space for which we can not compile and synthesize efficiently.

3. The template TTA used is fully programmable, pipelined, and if required, interruptible. Although transport triggered processors can be optimized to run critical code parts of an application, non critical parts can run on the same processor as well. On the other hand, many HLS systems are meant to be used for synthesizing highly application specific data paths, which are controlled by either FSMs or microcode. It is therefore difficult to adapt the functional specification once a processor has been synthesized, let alone that completely unrelated functions could be processed. In this sense our approach is more flexible.

In short, the MOVE framework aims at a fully programmable solution for large applications written in commonly used HLLs, while HLS systems are more oriented towards the synthesis of special function units (SFUs), or of processors which are dedicated to process a given function. In fact a synergy between both approaches is recommended. With this synergy HLS systems are used to design SFUs which are then automatically incorporated within the MOVE framework. The MOVE framework is already prepared to handle those SFUs [6].

MPEG audio 3

One of the aims of this thesis is to produce a TTA configuration optimized for MPEG audio decoding and to develop the related machine code. This chapter presents a general overview of digital signal encoding for compression, with particular detail on the MPEG compression format, and a further degree of detail on the audio part of the MPEG standard.

The first section of this chapter deals with the general topics of digital signal encoding: their importance, applications and compression encoding are briefly analyzed. The second section is a general description of the MPEG standard, its development and its main characteristics. The third section gives a detailed presentation of the MPEG audio format; in particular, it describes two classes of compression, known as 'layers', of the three available, namely the first and the second layers.

3.1 Overview of video and audio compression techniques

This section introduces the topic of data compression, its motivation and the basic principles that lay behind it. Some fundamental techniques are briefly explained; the reader is given a list of references for more detailed information. This description is mainly focused on video and audio signals, which are the chief application domain of MPEG standard.

3.1.1 Digital signals

Why to use a digital signal to represent information? Digital encoding of continuous (or analog) signals, as the most part of natural signals are, shows one main disadvantage compared to analog representation: it maps a continuous set of values into a finite set of values, or to better say, to a set of symbols representing values. As this represents a mapping from a domain containing an infinite number of elements to a domain consisting of a limited number of elements, some precision is inevitably going to be lost. The mapping process itself, called quantization, is sometimes treated as a noise source that shifts the original sample values to the nearest quantization level in the output domain.

Nonetheless, during last years the interest in representing analog data in a digital domain has become increasingly important. The reasons are:

1. Noise and error tolerance.
2. Possibility to compress the amount of data to transmit or store.
3. Editing easiness.

On a general basis, we can say that digital signals can be more easily manipulated and with far more freedom by any digital machine, such as computers.

Without considering the type of information carried by the digital signal, two main transformations can be applied on it. First, it can be encoded before it is transmitted through a noisy transport channel so that error detection and correction are possible after reception. This leads to one of the major advantages of digital signals when compared to analog signals, that is, the former can be encoded in such a way (by adding enough control information) that, knowing the noise extent of the transmission channel, it can be practically made immune to errors due to noise during transmission. Second, a digital signal can also be compressed, so that less data has to be transmitted or stored. On the opposite end of the transmission channel (or during the retrieving process from the storage media) the original, uncompressed signal can be reconstructed by an inverse decompression process.

Furthermore, depending on the type of information they carry, digital signals can be processed in countless different ways by computers. For example, an image or a video sequence can be filtered, rescaled or further edited; an audio signal can be emphasized and deemphasized, filtered, mixed with other audio sources, synchronized to other data streams, such as audio and video sequences, and much more.

3.1.2 Encoding formats for digital signals

The digital encoding of the diverse sources of information is more and more widespread in today's life. New, broadly supported digital standard encoding formats appear. For example, the audio CD format introduced by Philips in 1982 has become the most widely spread digital audio format for storage and playing. Similarly, the CD-ROM format is taking the same dominant market position as standard format for optical data storage, especially for the personal computer market.

Another format that is currently emerging is the Compact Disc Interactive (CD-i). This format was developed by Philips as a generic data format for Compact Disc storage support. It includes a subset of MPEG-2 for video and audio data compression, as well as audio ADPCM (for details on this format, see subsection 3.1.6).

One of the most awaited standards for digital signals encoding is the high definition television (HDTV). This format should be capable to transmit data at almost twice the current video broadcasting frame rates and at a more than double resolution.

Video signals are the most demanding from the point of view of required bit rate of transmission, throughput of encoding and decoding devices and storage capacity. A huge amount of information is to be supplied at a constant bit rate; thus real-time compression techniques are particularly needed for video signals, especially for sequences made of large, high definition pictures.

In table 3.1 some digital signal standards are portrayed with their principal characteristics. The telephone digital signal is part of ISDN, a suite of international standards for end-to-end

Signal	**Frame size (pixels)**	**Bits/ sample**	**Sample Rate kHz**	**Frame rate Hz**	**Bit rate Mbits/sec**
Telephone ISDN	-	8	8	-	0.064
CD audio	-	16	44.1	-	0.704
CIF	172 × 144	12	619.2	25	7.4
SIF	352 × 288	12	2340	25	30.4
CCIR-601	720 × 576	12	10368	25	124.4
HDTV	1920 × 1250	12	120000	50	1440

Table 3.1: Main characteristics of some different digital signal formats before compressing: size of one video frame, number of bits per each sample, frames per second, samples per second and bits per second. The latter value is computed from:
(bits per sample)×(frame size)×(frame rate) or (bits per sample)×(samples/sec)
The frame sizes of CIF, SIF and CCIR-601 refer to the PAL version.

digital communication over the public telephone network. The value shown refers to a single channel, but the basic transmission rate requires three of such channel. The signal is further upwards scalable by fixed steps of 64 kbits/s channels. CCIR-601 is a standard for digital television and professional digital video equipment standard, and it is also the base for all the sub-sampled interchange formats such as SIF, CIF, suited for digital compression. The SIF format in particular, is a typical input format for MPEG-1 and it is obtained by sub-sampling both in spatial and time directions.

It is worth to remember that the spectrum of bit rates (and therefore of performances required for encoding and decoding systems) ranges from 64 kbits/sec for a telephone signal up to 1440 MBytes/sec for HDTV, roughly 22500 times higher!

3.1.3 Data compression principles

One of the most important and valuable transformations that can be performed on a digital signal is compression. The idea behind compression is that most of the signals, no matter what may be their semantic content, (pictures, sound samples or texts), contain a large amount of redundant and therefore irrelevant information. This part of information can therefore be discarded before transmission or storing, and later reconstructed when retrieving information from the storage device or from the transmission channel.

The benefits of compression are obvious: after compression only a fraction of the bandwidth needed for the original, uncompressed signal is necessary to achieve the same purpose. Thus transmitting and receiving devices can be designed to cope with much lower frequencies, and therefore can be cheaper. Conversely, more signals carrying independent information fit into the fixed bandwidth of the transmission channel, i.e. more useful information can be transmitted with the same bandwidth at our disposal.

Why are typically digital signals so redundant? If the digital signal is obtained by sampling and quantizing an analog signal there is usually a lot of spatial and/or time correlation between samples. In fact, the measurements of an analog quantity, that is the samples, to be converted into discrete numerical values must be taken at a fast enough rate (in the time domain) or enough densely in a spatial domain, (of an image) in order to preserve the original analog quality after reconversion. More specifically, the frequency of sampling should be higher than or at least equal to the double of the maximum frequency present in the signal spectrum, as stated by the

Nyquist's theorem of sampling. This means that the measurements tend to be tightly correlated, that is, samples near each other are likely to have very similar values.

If the digital signal represents some synthetic information, like a text or a binary code, still there is a lot of redundancy that can be isolated and exploited. In this case the reason is that a human produced information tends to be structured, and this means that certain data patterns are much more common than others; for example, in a prose text some words, like articles and prepositions, appear very often; the structure of program sources is even stricter, and also binary executable code contains a lot of redundancy, as some operation codes are used and some locations are accessed far more frequently than others. Equally important, the great majority of possible data patterns are very unlikely to appear in many human made information. For example, consider a text: most part of the possible character combinations delimited by space or punctuation characters are not meaningful words in any language, and will probably never appear in any document. This suggests that the word is a good unit into which the text can be split in order to locate more redundancy and therefore achieve higher compression. As we will further see, the structure of the data to be compressed greatly affects the way compression should be performed.

Compression techniques can be divided into two classes: *lossy* and *lossless* compression techniques. In the latter only redundant information is removed by encoding frequently occurring bit patterns by less bits than other, seldom occurring patterns. For lossy compression a part of the irrelevant information is irreversibly filtered out, so that the reconstructed signal is not identical to the original one.

3.1.4 Compression format standardization issues

An important aspect that must be considered about compression is the enormous effort that is spent by international committee's to standardize compression formats and algorithms, so that companies and final consumers can take advantage of the presence of one or a few well defined and widely supported standards for each specific "high technology" product.

There are many points to consider about compression standards:

1. Extent of the application field.
2. Transparency with respect to implementation details.
3. Complexity that could derive for some type of implementations.

First of all, standards are usually defined for specific application fields, so that the particular nature of the information to be encoded can be exploited, but of course at the same time the wider are their possible applications the better it is. With "application fields" we do not mean just the nature of the signal (like video, audio, or speech audio, for example) but also the required performances, the quality of the signal after reconstruction, noise tolerance, and possibly many special features that can prove themselves useful for certain application. As example of the latter, we may consider the "pyramidal" class of compression algorithms for pictures, which satisfy both the necessity to transmit a large amount of data for a detailed, high resolution picture, with the need to let the receiving user have a first, rough perception of the transmitted picture as soon as possible, so that a quick check is possible for him, for example to decide whether or not to discard the rest of the downloading. The need to diversify the fields to which applying compression algorithms leads, as we will see, to some main distinction of the compression methods.

In the second place, standards should be as transparent as possible for what concerns hardware and software implementation details, as well as algorithmic choices, so that different realizations are possible for encoders and decoders compliant to the same compression format, depending on their sophistication degree, their cost, speed and precision. Another important reason to keep implementation specific details out of the standards is that so doing a broad space is left to further scientific research for finding new, more effective algorithms to achieve a particular result required by the standard. For example, in the MPEG standard the way to compute the Discrete Cosine Transform (DCT) is left to the implementor, therefore new and better algorithms can be researched and applied to perform DCT or FFT [32]. Similarly, a lot of research is currently spent on algorithms to discover blocks and vectors for motion compensation in video compression formats.

Nevertheless, the standard should also take into account the implementation complexity that could derive from certain design choices. This is especially true for hardware implementations, and has for example lead to some format changes in MPEG-2 as compared to MPEG-1.

Today industries have at their disposal a great number of compression standards. Just to mention the most important and well known ones we could remember JPEG (Joint Photography Expert Group) for still images; MPEG-1 and MPEG-2 mainly for video and its related audio compression; H.261 and H.263 for low bit rate video and audio compression, especially suited for teleconferences and environments which can only cope with low bit rates. Among audio encoding formats a particularly complex and efficient format is Dolby AC-3.

3.1.5 Video signals and their compression principles

So far, we have pointed out that, depending on the type of signal to be compressed, there is a large amount of redundancy that can be discarded or more efficiently encoded. In this section we are restricting our attention to video signals and in the following one to audio signals to see how redundant information can be located.

A video signal can be considered as a continuous three dimensional signal $V(x, y, t)$. In practice, however, it is not possible or necessary to work with an actual 3-D signal. Since our retina is characterized by a certain persistence of the light stimulus that excites it, we can create a perfect illusion of continuous, fluid motion just presenting a sequence of still pictures, called frames, at a high enough rate.

Thus a video signal consists of a sequence of frames, that in turn can be regarded as a sequence of two dimensional signals $V(x, y)$ in a spatial domain representing a picture that is changing with time, the third component. However, we will see that taking into account also the third dimension, and therefore the strong correlation that is usually present between adjacent frames, enables us to achieve major improvements in compression ratios.

The frequency with which frames are presented ranges from 24 to 50 Hz, and is referred to as the frame rate. To obtain a perfect illusion of smooth motion at least 20 frames per second are needed. Different video formats may well differ in frame rate value: for example, movies are represented with 24 fps, while American television standard NTSC is characterized by a frame rate of 30 fps.

Each frame consists of a two dimensional array of picture elements, corresponding to samples in the spatial domain and referred to as *pels*. The human eye has 3 types of color photoreceptor cells; because of this, only 3 numerical components are necessary and sufficient to represent the color of a pel. Thus, pels can be treated as points in a three dimensional color space. Different color spaces can be chosen for different purposes, and there exist transformations to translate data from one color space to another. For example, for video signal the additive space of Red, Green and Blue components is often used, while for video compression other color spaces, in which the brightness information is separated from the actual color information, are preferable. The reason for this is that the human visual system is more sensitive to brightness changes than to changes in color. Therefore, it is possible to encode the separate brightness component (denoted by 'Y') with more accuracy and to devote less bits to the remaining components, called chrominance, without visible effect on the final image quality. This is a first example of how the knowledge about the contents of the data to be processed leads to better compression.

There is another criterion to classify compression techniques applying to video signals, that is orthogonal to the lossy/lossless criterion mentioned in the section 3.1.3: *temporal* (interframe) or *spatial* (intraframe) compression.

The temporal compression techniques decorrelate the time component of the video signal $V(x, y, t)$. A high correlation is expected to exist between pels in the same position belonging to different frames when the time interval that separates the frames is small enough. Temporal compression is currently divided in two subclasses. In the first subclass the whole preceding frame is used as a base to predict the content of the current frame to encode. Only differences between the current frame and the previous one (or a prediction based on it) are encoded. The second subclass is called motion estimation and is based on the principle that in a video sequence many areas of a frame are just translated in the following frames. An attempt is made to identify which part of the current frame are moved with respect to the previous frame, so that only the displacement vector and the size and position of the moving area must be encoded for the current frame.

Spatial compression techniques take into account only single frames at a time and exploit the spatial correlation that exists between neighboring pels. In fact, pels close to each other are expected to have similar values: especially in real images, we usually do not find very high spatial frequency, i.e. color changes tend to be smooth. An important class of compression techniques, also applicable to audio signals, exploiting this characteristic, is the class of techniques based on transformations to a frequency domain. High frequencies (that is, rapidly changing values of the signal, both in time or N-dimensional spatial domain) are unlikely to be present in many kinds of real data. Besides, human perception systems [26] are less sensitive to high frequencies. Those two principles can be exploited effectively if we apply a transformation to a frequency domain. By this way, we can identify higher and irrelevant frequencies and encode them with less quantization levels, thus a lossy compression is established. Frequency domain transformations for images are usually performed on small square areas, called blocks. A typical size for such areas, used in many compression formats, is 8×8 pels [12].

Transformed blocks correspond to two dimensional spectra of the frequencies present in the original block. Usually the highest values are concentrated around the upper left corner of the transformed block, corresponding to the continuous component, that is, the average value of the block. After the transformation a quantization matrix is applied to the transformed block to encode the samples with a different number of levels, depending on their frequency. In most cases, the highest non zero frequency sample present in a block is much lower than the maximum possible value, therefore it is possible to encode only the range of values up to the highest non zero sample without losing further information. Since it is convenient to reach such frequency sample as soon as possible after having included all the lower frequency samples, a particular zig-zag scanning path is employed. The scanning process starts with the continuous component and proceeds moving along the diagonal opposite to the starting corner, so that when we encounter a frequency sample almost all lower samples have already been scanned.

Depending on the complexity of the compression format, further processing is performed on the data. It is important to point out that the only lossy steps are typically color sub-quantization and transformed block quantization. Those two steps, but in particular the latter, are parameter-

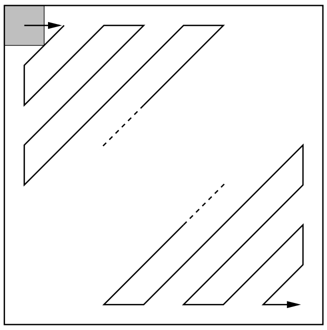


Figure 3.1: Scanning path of frequency samples in an 8×8 pels transformed block. The average value of the block, called DC sample, is highlighted.

izable depending on the trade-off between compression ratio and final quality of reconstructed video.

3.1.6 Audio signals and their compression principles

An audio signal is a continuous, one dimensional signal representing the pressure changes with time of a media (typically air) with respect of an average value. The human ear can typically perceive frequencies between 20 Hz and 20 kHz, while human voice can produce frequencies between 40 Hz and 4 kHz. This limits are important factors to remember when discussing digital audio encoding: in fact, according to Nyquist's theory, a sampling rate of 40 kHz is sufficient to faithfully represent the sound range of human hearing. Typical sampling rates range from 8 kHz (sufficient for human voice encoding) to 48 kHz.

Digital audio signals are encoded as sequences of samples, each sample is represented by a discrete, quantized value called codeword. Depending on the type of application, the number of quantization levels of digital audio signals ranges from 256 to 65536 (8 to 16 bits). In the Pulse Code Modulation (PCM) format the value of the codeword represents the amplitude quantized value of the sound wave at the instant of the sample measurement. There are five main ways to achieve compression of audio signals, listed in order of complexity:

1. Nonlinear quantizer step.
2. Exploitation of the similarity between neighboring samples.
3. Discarding of redundant information of an audio channel with respect to another.
4. Transformation in a frequency domain, to perform time frequency decorrelation.
5. Exploitation of the masking effect.

In a uniform PCM quantization levels are equally spaced. By performing a transformation, the quantizer step spacing can be changed to be logarithmic, so that low amplitude samples are encoded with greater accuracy than higher ones. Since the sensitivity curve of human ear is approximately logarithmic, such a quantizer step allows us to cover a larger range of values with the same number of bits and with no perceivable influence on sound quality. Thus we can establish a first compression stage; for example, 8 bits per sample, usually employed for logarithmic quantizers, can represent the same range of values that would be achieved with 14 bits per sample in uniform PCM.

PCM encodes each audio sample independently from adjacent ones. However, usually adjacent samples are similar to each other and the value of the next sample can be predicted with some accuracy using the value of the current one. For example, one simple prediction method is to assume that the next sample will be the same as the current sample. The Adaptive Differential Pulse Code Modulation (ADPCM) encoding methods compute the difference between each sample and its predicted value and encode only the difference (hence the term "differential") [29]. Fewer bits (typically 4) are needed to encode the difference than the complete sample value. Different flavors of ADPCM encoders are possible, depending on the way predicted samples are computed. The quantizer and the predictor can in fact adapt themselves to the signal characteristics by changing quantizing or prediction parameters (hence the term "adaptive").

Audio signals are often made of many sequences, each of them representing an independent audio channel. In this way it is possible to achieve effects like stereo or 'surround'. Values of such sequences can be transmitted through separate transmission channels or interleaved in the same channel. Audio signals are often made of a left and right stereo pair. Typically, signals of a stereo pair contain a large amount of redundancy one with respect to the other, therefore it is profitable to exploit irrelevancy. For example, is possible to encode only one channel and the difference between the two of them, or to encode only the lower frequencies separately, and couple the higher ones in a single channel, as they tend to be more similar and their differences are less perceivable to human ear.

Since the human ear is less sensitive to high frequency sound[1], the higher is the frequency of the audio signal the less relevant is the information it carries, as for video signals. Therefore, a frequency domain transformation is profitable to detect which frequencies are present in the signal and which ones can be discarded. The transformation is performed on a group of adjacent samples. The result is a discrete version of the frequency spectrum for that group of samples, as if it were a period of a periodic, discrete signal. Many different discrete transformations, like the Discrete Fourier Transform (DFT) and its more efficient counterparts, the Fast Fourier Transform (FFT) and the DCT, can be employed. After the transformation, it is possible to encode frequency samples with a different number of quantization levels. In this way we can take into account human ear and signal characteristics to decide how many bits to devolve to different parts of the frequency spectrum.

The masking effect is also an important source of compression. The human ear is not able to perceive tones if their frequency is near to the frequency of stronger tones. Therefore it is possible to raise the noise level around strong frequencies without affecting the sound perception. The masking effect depends proportionally on the distance between the two frequencies and on the difference of amplitude between the two tones.

3.2 The MPEG compression standard

This section begins with a general description of the MPEG compression standard and the history of its development. Both the currently available versions, namely MPEG-1 and MPEG-2, are presented, with particular detail on the former and the differences between the two versions. The audio part of the compression standard is treated with greater detail in the next section.

First, a brief history of the MPEG development is presented along with some general information about the structure of the standard and its official documentation. The MPEG-1 standard for System and Video streams is then presented. A summary of the differences between MPEG-1 and 2 concludes this section.

[1]The sensitivity curve of human ear is actually rather complex. It shows its peak around 2kHz and descending slopes both toward zero and high frequencies.

3.2.1 What is MPEG?

MPEG stands for Moving Picture Experts Group and is the name of an international committee that began its life in late 1988 with the goal of standardizing video and audio encoding for compact discs. By the end of 1992 a compression standard emerged, called MPEG-1, with SIF video and compact disc audio samples encoding at a bit rate of 1.5 Mbits/sec as primary target. These specifications approximate the perceptual quality of a consumer video tape (VHS).

The MPEG standard is described by means of two levels of specifications: syntax and semantics. The former can be considered as a complex language which provides an efficient and flexible way to describe the elements and the structure of the bit stream of encoded data. Along with this data all information the decoder needs to reconstruct the original “raw” data from its compressed representation is added to the coded stream. The semantics level specifies which algorithms must be adopted for each possible value of bit stream elements, that is, how to use the information provided by the syntax. The MPEG-1 standard proved itself generic enough to be successfully applied to bit rates and sample rates far higher than the ones of the original application. After MPEG-1 definition a new phase had begun to define a syntax to efficiently describe broadcast, interlaced video. The new syntax was called MPEG-2 and it was capable also to describe multichannel surround audio signal retaining backward compatibility. A third phase, called MPEG-3, was anticipated in 1991, for HDTV, but was later withdrawn as the MPEG-2 syntax proved itself capable of covering the signal specifications needed by HDTV. In late 1992 a fourth phase has been launched (MPEG-4) to explore the requirements of a more diverse set of applications and efficient methods of coding signals at low bit rate and low sample rate.

The MPEG-1 specification document consists of five parts. Each document is a part of the ISO/IEC number 11172. We are going to describe in detail only the first three documents, called *System Part*, *Video Part* and *Audio Part*.

The purpose of System specifications [13] is to define the format into which combining one or more data streams produced by a video and audio encoder along with timing information to form a single stream. Once combined in a single stream data are in a form well suited to digital storage or transmission. These specifications impose rules that make possible decoding and playback without overflow or underflow of decoder buffers.

The Video Part [14] describes the syntax and the semantics for video signals. Video is structured in a series of nested layers, each containing a finer granularity of sample or coefficient clusters. At each layer, several algorithms are made available that can be used in combination to achieve efficient compression. The syntax provides also a number of different means for assisting decoder synchronization, random access of a frame of the video sequence, buffer regulation and error recovery.

The Audio Part [15] describes the syntax and the semantics for three classes of audio compression methods, known as Layer I, II, and III. Each layer represents a different balance between syntax complexity and coding efficiency at lower bit rates.

The fourth part, called *Conformance* [16], defines the meaning of MPEG conformance for all the three parts and provides guidelines for determining compliance in bit streams and decoders. It is important to remember that the encoder rests outside the normative scope, as long as the bit streams it produces are compliant. The decoder is just implicitly specified by the syntax of compliant coded stream, but its implementation is also outside the standard.

The fifth part, *Software Simulation* [17], contains an example software encoder and decoder for video and audio, written in ANSI C. An example of system encoder and decoder is also provided.

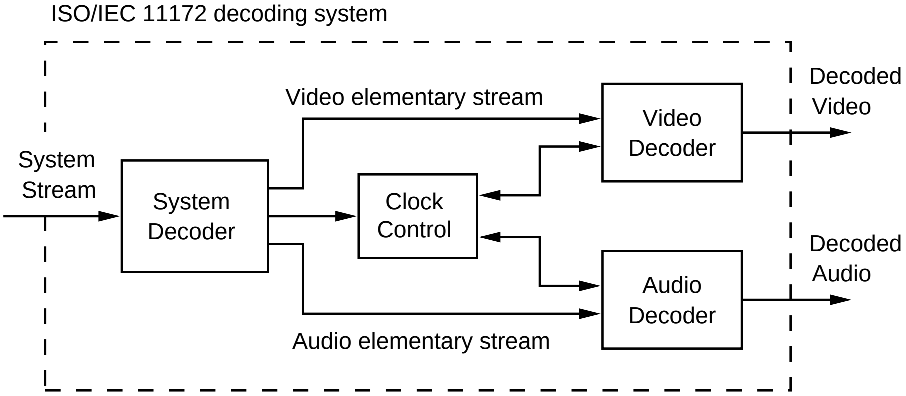


Figure 3.2: Prototypical MPEG-1 decoder

The MPEG-2 standard is defined by five analogous documents [18, 19, 20, 21, 22] and four additional parts. The seventh part [23], in particular, defines a new syntax to efficiently decorrelate multichannel surround audio. For further details see subsection 3.2.4.

3.2.2 MPEG-1 System Part

The first part of the MPEG standard provides syntax and semantics for the following purposes:

1. Transporting packets of compressed audio and video data over digital channels and storage media.
2. Synchronizing video and audio streams at presentation time.
3. Multiplexing of multiple compressed streams into a single stream.
4. Initialization of buffering for decoding startup and continuous buffer management.

Figure 3.2 depicts a prototypical MPEG-1 audio and video decoder[2]. The System Decoder demultiplexes the stream into elementary Video and Audio streams that serve as input to Video and Audio decoders. Furthermore, it extracts all the necessary timing information from the system packets to enable the synchronization of video and audio sequences. In addition to Audio and Video, three other types of elementary data streams can be multiplexed and synchronized in System streams: private, reserved, and padding. The latter is used to maintain a total data rate constant, to achieve some large block types of alignment or to prevent buffer underflow. Stuffing bytes are also allowed within each data packet, but are limited to a maximum of 16 and are especially suited to provide word alignment. The syntax of data carried by private streams is out of the standard scope, and enables implementors to add non-standard features without compromising compatibility: it should always be possible for all additional information to be ignored by a standard MPEG decoder without affecting the decoding process.

An MPEG-1 multiplexed stream consists of one or more *elementary streams* multiplexed together. Such streams of data are constructed in two layers: a system layer wrapped around a compression layer. After the system decoding and demultiplexing, only the compressed layer is presented as input to Video and Audio, Decoders. Each elementary stream consists of access units, which are coded representations of presentation units. The presentation unit for a video elementary stream is the picture; the presentation unit of an audio elementary stream is the set of time samples coded in an audio frame (see section 3.3 for details).

[2]Not shown in the picture, a private elementary stream can be present.

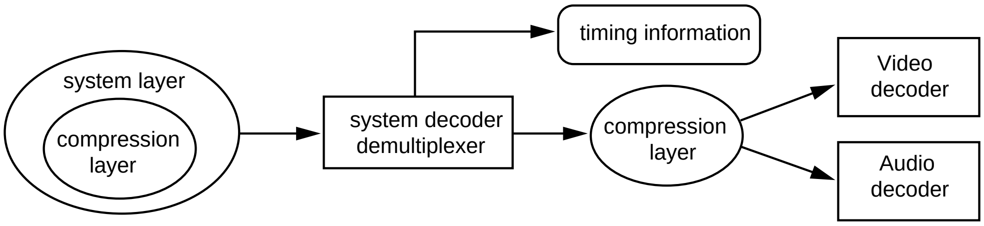


Figure 3.3: Stream layer flow through System and Video/Audio decoders.

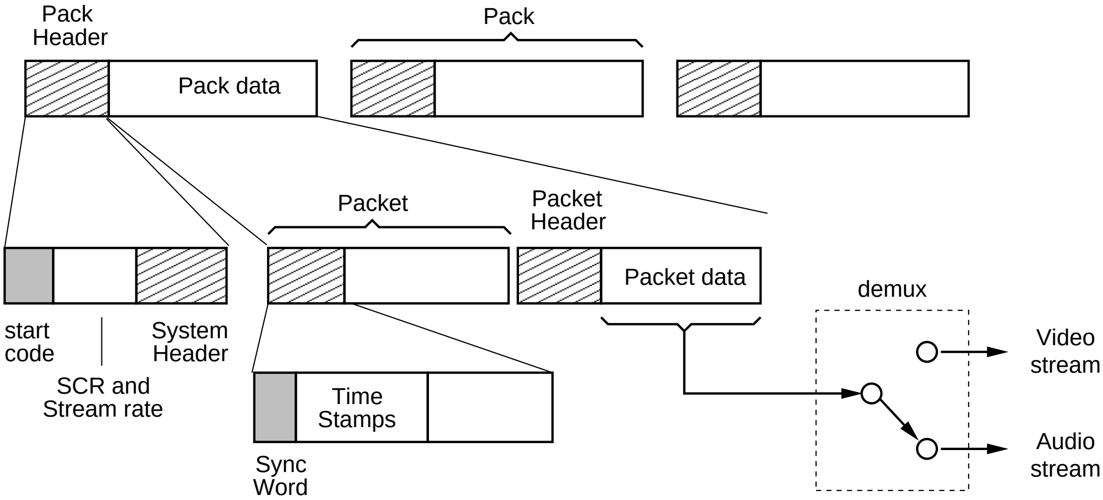


Figure 3.4: System layer structure: pack and packets disposition within the bit stream.

The system layer can be further subdivided in two sub-layers, depending on the kind of operations that the System decoder performs on it:

1. Multiplex-wide operations.
2. Stream specific operations.

The multiplex-wide operations apply to the whole multiplexed stream and include coordination of data delivery, clock adjustment and decoder buffer management. It is important to point out that if the delivery rate is not controllable by the decoding system, then Audio and Video decoders must slave their timing to the digital storage medium or the transport channel rate to avoid overflow or underflow of their buffers.

Stream specific operations are demultiplexing of elementary stream packets and synchronization among multiple elementary streams.

Multiplexing strategies for packets within the data stream may widely vary depending on their size and the relative placement of packets from different streams. The main constraints on multiplexing are the model of decoder that is considered and its buffer sizes, i.e.the necessity to avoid overflows or underflows of decoder buffers. In general, short packets require less buffering but more system overhead than large packets[3].

As shown in figure 3.4, MPEG-1 stream is made of as sequence of packs. Each pack commences with a pack header and is followed by zero or more packets. The pack header begins

[3]Other considerations, such as the need to align packets with sectors on specific storage media, like Compact Discs, influence the choice of packet length. Multiplexing may occur in conjunction with the encoding of elementary streams or the two operations may be independent.

with a 32-bit start-code and is used to store timing and bit rate information. The entire stream begins with a system header, included within the first pack header, that may be optionally repeated, carrying a summary of the system parameters defined in the stream. Among other pieces of information, the maximum data rate and the maximum number of simultaneous video channel of the incoming stream are delivered in the system header. A decoding system may use these limits to establish its ability to play the incoming stream.

Each pack contains an integer number of packets. Packets, which are used to keep elementary streams data distinct, consist of a packet header followed by packet data. As shown in figure 3.4 the packet header begins with a 32-bit start-code that also identifies the stream to which the packet data belongs. Each packet header may contain decoding or presentation timestamps (DTS and PTS). These times refer to the first access unit that commences in the packet. The packet data part contains a variable number of contiguous bytes from a single elementary stream.

3.2.3 Video layer

The Video data structure consists of several nested layers. A part from dividing the video sequence in its intrinsic sub-parts, like single pictures or blocks (the elementary units to which two dimensional DCT is applied), each layer provides a scope for some decoding parameters. The layer hierarchy is compound of the following levels.

1. Bit stream (or video sequence).
2. Group of Pictures.
3. Picture.
4. Slice.
5. Macroblock.
6. Block.

Besides Macroblock and Block, all layers are identified in the MPEG bit stream by a unique start code. Start codes are 32 bit byte-aligned codes that provide a mechanism for cheaply searching coded bit streams for commencement of various layers of video without having to perform a complete parsing of the stream. The task of re-synchronizing after an error is therefore made easier. At top level, video sequence layer, information about picture size, frame rate and bit rate are defined.

The video bit stream consists of a sequence of pictures, which are grouped in *Groups Of Pictures* (GOPs). The function of each GOP is to provide a random access point to the video sequence. For this reason, it must start with a picture that is coded independently from the other frames of the video sequence. Three types of pictures are possible, depending on the type of temporal compression technique they are encoded with:

1. *Intra coded pictures* (I pictures).
2. *Predicted pictures* (P pictures).
3. *Bidirectional predicted pictures* (B pictures).

An I-picture is compressed only in the spatial domain, so it exploits no temporal prediction and has no relation with other pictures in the video sequence. A P-picture is encoded as a reference to the past I or P pictures. A B-picture is encoded [as a reference] to the past or future I or P frame (or both). As the latter two types of pictures are dependent from previous or following ones, they can not provide random access points.

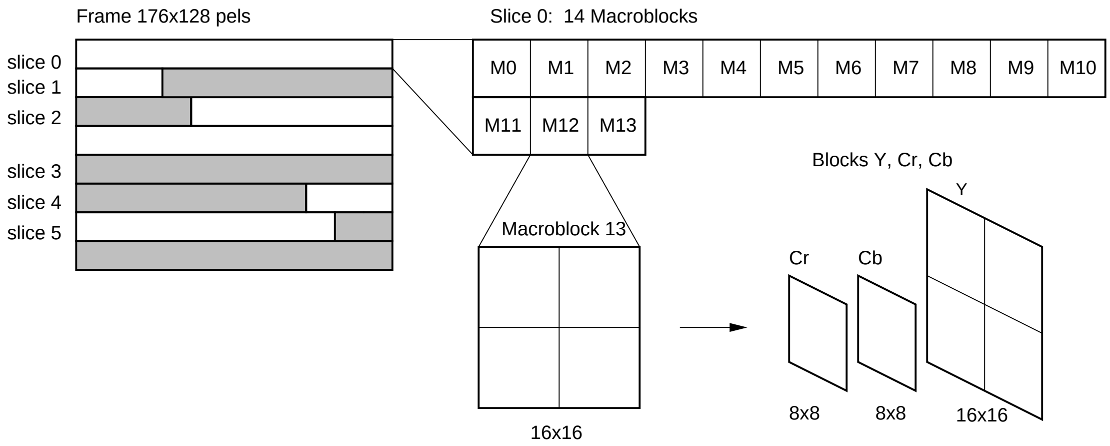


Figure 3.5: Frame inner structure, consisting of slices, macroblocks, and blocks. The example frame is 11 macroblocks wide [per] 9 macroblock high, for a total of 176×128 pels.

A GOP may contain any number of frames of any of the three types in any position. The major constraint is that it must start with an I frame. Furthermore, since B pictures are dependent on future frames, the order of the frame sequence within each GOP is rearranged during encoding so that frames which B pictures refer to are encoded and sent before them. Usually the structure of a GOP consists of I pictures separated by one or more P picture, which in turn may be separated by one or more B pictures. Such regular patterns are the industry's favorite, but it's worth to remember that any sequence is possible, and that it has not to be regular, though a regular pattern helps to maintain a constant stream bit rate. For example, sophisticated encoders may adapt the frame sequence to the video signal local properties.
For example, the display order of the GOP

$$I_1\ B_2\ B_3\ P_4\ B_5\ B_6\ P_7 \ldots$$

is rearranged into:

$$I_1\ P_4\ B_2\ B_3\ P_7\ B_5\ B_6 \ldots$$

Pictures are build up of slices, which in turn consist of an arbitrary number of consecutive macroblocks ordered in raster-scan fashion. Slices can not overlap one on another and must leave no gaps within the frame they are part of. Macroblocks are formed by 16×16 pels and are coded by means of six 8×8 blocks. Pels are represented in the color space *YCbCr*: four of them are luminance blocks, and each of their elements corresponds to a pel of the macroblock, the remaining two represent the chrominance components, Cb and Cr. Their values correspond to subsamples in the grid of the luminance matrix, taken in the middle of four luminance samples (thus an half-pel position grid is taken for them). The chrominance values in the position of pels are computed by interpolation. The sub-sampling ratio of the chrominance component with respect of the luminance is called *chroma format*. The chroma format of the sub-sampling above mentioned is 4:2:0.

The spatial compression techniques used by MPEG Video are based on two dimensional DCT. Each block of a macroblock is separately transformed from a spatial domain to a frequency domain. The DCT coefficients are then divided by an 8×8 quantization matrix. Since the human visual system is less sensitive to high spatial frequencies and the probability distribution for the high frequencies is lower, these are quantized more coarsely [12]. The quantization process is the main leverage to trade image quality for compression ratio. In fact, the quantization matrix is parameterized by a quality index that can be set by the user or by some heuristic algorithms based on the encoding constraints. After quantization the coefficients are encoded in a zig-zag sequence and are further compressed by means of run-length and entropy encoding[4].

The temporal compression technique used by MPEG is based on motion estimation. Macroblocks are used as the entities to perform the motion estimation and compensation at. A module of the encoding system, the motion estimator, is charged of finding the direction and the measure of displacement of the macroblock to be encoded with respect of the reference frame. This displacement is called motion vector, and it must minimize the difference between the reference macroblock and the macroblock it points at[5]. When an appropriate motion vector is found for a macroblock, this is encoded with its motion vector, which specifies the area of the reference frame to be copied, and the difference between the reference area (the prediction) and the actual value of the block. The difference is encoded with the spatial compression algorithm above mentioned. The interpolated motion estimation variant is analogous to the technique above described, except for the computation of the macroblock prediction. Instead of simply choosing a macroblock from the reference frame, a macroblock prediction is computed by interpolation of the same macroblock of the previous and the successive reference frame.

MPEG syntax allows each macroblock of a frame to have a different compression type, although types are closely related to the type of picture they belong to. For example, for every macroblock in a I picture the macroblock type must be intra. This means that only spatial compression is applied on the macroblock, and it is independent from other pictures.

Macroblocks in a P picture are usually coded by forward motion compensation, using a previous frame as reference, but it's also possible to encode them as *intra*. Other possible types are *null motion prediction*, when the motion estimation vector is zero, or *skipped*, when no data is encoded for the macroblock, as its result is identical to the corresponding macroblock of the reference frame after quantization. For a B picture two more types are possible, *backward* and *interpolated*. Backward motion estimation is equivalent to forward motion estimation, but the reference frame occurs at a later time than the current picture in display order. Interpolated estimation uses two references, a past frame and a future frame, to compute the estimation.

3.2.4 Summary of differences between MPEG-1 and 2

MPEG-2 provides several extensions to MPEG-1 in all the three parts of the standard, and it is designed to cover a wider range of applications. The most significant enhancement over MPEG-1 regards the Video layer and is the addition of syntax for efficient coding of interlaced video. In fact, the primary application targeted during the MPEG-2 definition was the digital transmission of broadcast TV quality video at coded bit rates between 4 and 9 Mbits/sec. However, the MPEG-2 syntax has been found to be efficient for other applications such as those at higher bit rates and sample rates.

In the system layer two types of data streams are defined: the *Transport stream* and the *Program stream*. The former carries a single *program*, i.e. a group of elementary streams logically related, all sharing a common time base for their time stamps. The latter can carry multiple programs simultaneously, and it is optimized for use in applications where data losses may be likely. Program streams are similar to the MPEG-1 Systems standard (ISO/IEC 11172-1) and are optimized for multi-media applications and for MPEG-1 compatibility. Both streams are

[4]The decoding system must have access to the same quantization matrices and entropy tables of the encoder. Therefore the syntax allows to specify which matrices and tables must be used.

[5]Only the luminance is taken into account to perform the estimation.

designed to support a large number of known and anticipated applications, and retain a significant amount of flexibility required for such applications. Transport Streams are well suited for the transmission of digital television and video telephony over fiber, satellite, cable and also for storage on digital video tape and other devices.

Both the Transport Stream and Program Stream are built on a common Packetized Elementary Stream packet structure, facilitating common video and audio decoder implementations and stream type conversions.

In addition to interlaced video, another key feature of MPEG-2 are the scalable extensions, which permit the division of a continuous video signal into two or more coded bit streams representing the video at different resolutions, picture quality (i.e. SNR), or picture rates.

Another important feature of MPEG-2 Video standard is its subdivision into 4 *profiles* and 4 *levels*, which limit the standard extent to meet the need for efficiency and feasibility of implementations for common classes of applications. “Profiles” define a syntax subset, thus they restrict the set of possible algorithms that are applicable to the coded stream. “Levels” limit coding parameters (sample rates, frame dimensions, coded bit rates, etc.) and therefore define a minimum level of performances required to a decoder to handle a particular bit stream. The combination of a profile and a level produces an architecture, which defines the ability of a decoder to handle a particular bit stream. In this way coded streams are grouped into a set of application classes.

Several other subtle enhancements have been introduced in Video layer, which have a noticeable improvement on coding efficiency, even for progressive (non-interlaced) video. The most important are listed below.

1. The syntax has been extended to specify wider ranges of some video parameters as aspect ratio and picture size. Signal information to assist post processing, as the video source type has been added.

2. Some non-linear quantization steps for the encoding of block samples have been introduced. It allows a larger range of values.

3. Encoding of the average value of the block with up to 10 bits of precision.

4. Some syntax constraints have been added to reduce the complexity of implementations, and facilitate the stream parsing and parallelization of the decoding process. Each macroblock row of a frame must start and end with a slice. In MPEG-1 slices have no position constraints: an entire frame can consist of a single, big slice. Macroblock stuffing bits, allowed in MPEG-1, are illegal.

5. Motion vectors are now always represented along a half-pel grid. With MPEG-1 optional integer-pel grid was possible. Half-pel accuracy encourages half-pel interpolation, that offers significant coding gain.

6. New prediction modes, especially suited for encoding of interlaced frame macroblocks, have been added. These compression types noticeably contribute to the improvement of compression efficiency with respect of MPEG-1.

7. The concept of the Group Of Pictures layer does not exist in MPEG-2, it is only an optional header, while in MPEG-1 the Group of Pictures header is mandatory, and must follow a sequence header.

8. In the 4:2:0 chroma format, the position of Cb,Cr samples has been shifted to be horizontally aligned to the same position with the luminance sample grid. By this way sample positions are consistent with the subsampled chrominance positions of 4:2:2 television signals, where only the horizontal direction is subsampled by a factor 2 for chrominance.

The MPEG-2 Audio Standard is currently under development for low bit rate coding of multichannel audio. MPEG-2 Audio coding will supply up to five full bandwidth channels (left, right, center, and two surround channels), plus an additional low frequency enhancement channel, and up to seven commentary or multilingual channels. The MPEG-2 Audio Standard will also extend the stereo and mono coding of the MPEG-1 Audio Standard to half sampling-rates (16 kHz, 22.05 kHz, and 24 kHz), for improved quality for bit rates at or below 64 kbits/s, per channel.

MPEG-2 Audio attempts to maintain as much compatibility with MPEG-1 audio syntax as possible, while adding surround-sound channels to the original MPEG-1 limit of 2 channels. The main channels Left and Right in MPEG-2 audio remain backwards compatible, whereas new coding methods and syntax will be used for the surround channels.

As we mentioned, the seventh part of the MPEG-2 documentation describes non-backwards compatible multichannel coding Standard. It is not yet decided whether it will be included in the final document. At present, some compression schemes are being considered as an amendment to the MPEG-2 audio standard. One such popular system is Dolby AC-3.

3.3 MPEG-1 Audio

This section describes in detail Layer I and II of MPEG-1 Audio part. First a general description of MPEG-1 Audio layers and their applications is given. Thereafter the formalism utilized to present the standard specifications is described. Then the complete syntax and semantics of Layer I and II is presented by means of this formalism. The last part of this section describes the algorithms related to the decoding process.

3.3.1 Introduction

The MPEG-1 Audio format is organized in three classes of compression algorithms, called audio layers. From Layer I to Layer III the encoding complexity is increasing as well as the coding efficiency, especially at lower bit rates. By definition, a Layer III decoder which is MPEG-1 compliant must be able to decode bit stream data which has been encoded also in Layer II and I. Similarly, a Layer II decoder must be able to decode Layer I bit streams. There are 4 possible encoding *modes* for the audio channels, which are orthogonal to the layer: *Single channel*, *Dual channel*, *Stereo* and *Joint stereo*. In dual channel mode two audio channels carrying independent audio program contents (e.g. bilingual) are encoded within one bit stream. The coding process is the same as for stereo mode. The coding algorithm used for Joint stereo mode exploits stereophonic redundancy [28].

Layer I maps audio input into 32 frequency subbands (20 Hz to 20 kHz). Data are segmented into fixed blocks of 32 samples, and for each subband sample a psychoacoustic model adaptively determines the number of allocation bits and quantization level. The psychoacoustic model is based on several characteristics of the human ear perception. Layer I has similarities in terms of complexity, efficiency, and syntax to the Sony MiniDisc and the Philips Digital Compact Cassette (DCC). The latter achieves an optimal bit rate (in terms of compression and sound quality tradeoff) of 384 kbits/sec.

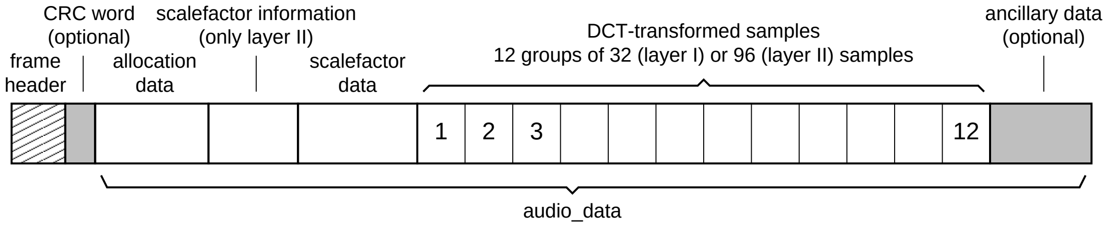


Figure 3.6: Structure of the audio frame: frame header, optional CRC word, allocation data, scale factor selection information (Layer II), scale factors, sample data, optional ancillary data.

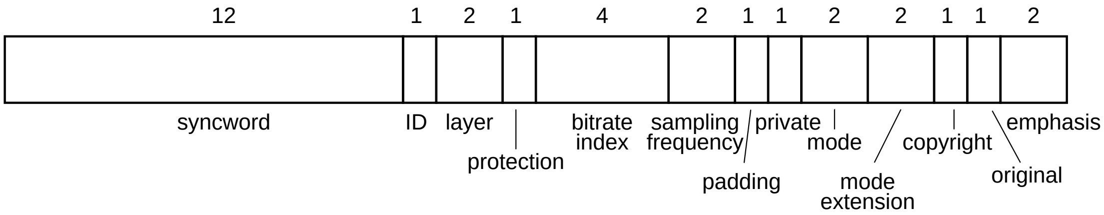


Figure 3.7: Frame header syntax.

Layer II provides additional coding for bit allocation, scale factors and samples. Data is segmented into blocks of 96 samples, corresponding to three blocks of 32 samples interleaved in a subband-wise fashion. Layer II is the industrial favorite, and is applied almost exclusively in satellite broadcasting (Hughes DSS) and compact disc video (White Book), which achieve an optimal bit rate of 224 kbits/sec.

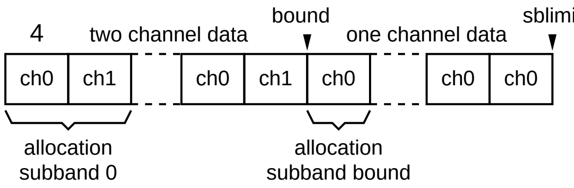
Layer III, that will not be covered in further detail in this report, introduces increased frequency resolution based on a hybrid filter bank. It adds non-uniform quantization, adaptive segmentation of samples and entropy encoding of the quantized values. Layer III has found use in ISDN, satellite, and Internet audio applications. The optimal bit rate with ISDN and Internet application is typically 128 kbits/sec.

3.3.2 MPEG-1 Audio syntax and semantics

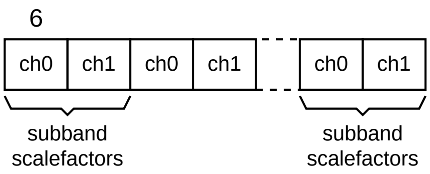
An audio MPEG stream is divided into frames, whose structure is depicted in figure 3.6. For Layer I and II the frame is defined as the smallest part of the audio bit stream which contains all the information needed for its decoding. A Layer I frame contains 384 time samples encoded and subdivided in 12 blocks of 32 subband samples, a Layer II frame contains 1152 time samples in 12 groups of 96 subband samples.

Each frame is introduced by a syncword, the 12 bit constant ‘1111 1111 1111’. After a syncword has been detected, the frame can be parsed. Each frame consists of a frame header, an optional word for error checking, a main body called audio_data and a block of optional, user definable data. The frame is also subdivided in an integer number of elementary units called slots. In Layer I a slot corresponds to 4 bytes, in Layers II and III it corresponds to 1 byte.

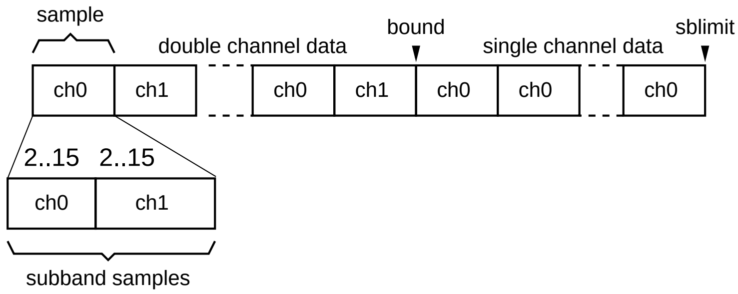
In figure 3.7 a scheme of the frame header is shown. ID is a flag bit that indicates if the incoming stream is a ISO/IEC 11172-3 compliant stream or has some reserved format. Protection bit indicates if the parity-check word is present after the frame header. Bit rate index refers to a table of values that specifies in kbytes/s the bit rate value for the stream. The sampling frequency can be one in 32, 44.1 and 48 kHz. Padding bit specifies if the frame contains an additional slot to adjust the value of the bit rate. Padding is in fact necessary with 44.1 kHz



(a) Layout of allocation data for a frame coded in Joint stereo mode.



(b) Layout of scalefactor data for a Layer I two-channels frame.



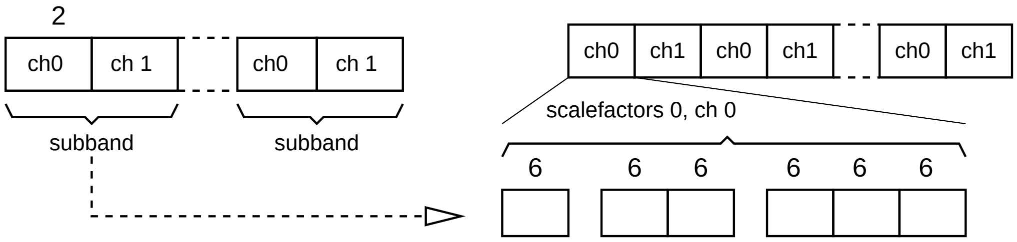
(c) Layout of sample data block for a Layer I frame coded in Joint stereo mode.

Figure 3.8: Layout of Layout of the allocation data segment (a) for Layers I and II. Layout of the scale factor data block (b) and sample data block (b) for a Layer I Joint stereo mode frame. The number above data slots represent their bit size.

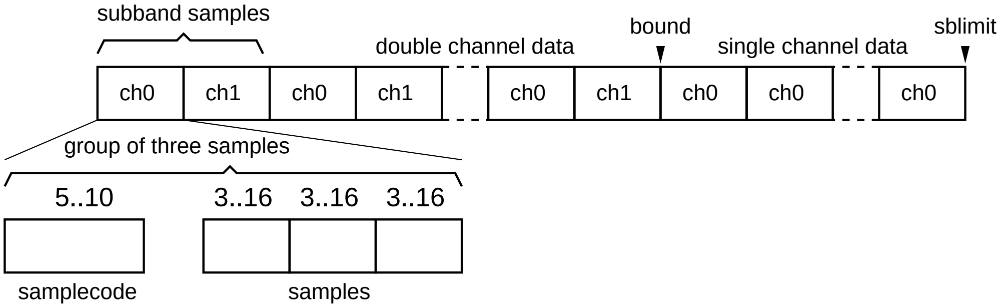
sampling frequency, as it corresponds to a bit rate with a fractional number of slots. The mode bits indicate which mode is used for the channel encoding. Mode extension is used only for joint stereo mode and it specifies the value of *bound*, the number of subbands coded in intensity stereo mode, i.e. coded in a single channel exploiting the redundancy between the stereo pair. The private bit is not defined by the standard, and it can be freely assigned by the implementor. The emphasis bit determines the deemphasis algorithm that must be used to reconstruct the original samples. Four modes are possible: 50/15 μs, reserved, CCITT J.17, and no emphasis. crc_check is a 16 bit word containing a parity-check used for error detection. In Layer I the CRC check word protects the last 16 bits of the header and the allocation data. In Layer II the protection is extended to the scale factor selection information.

The number of ancillary bits equals to the available number of bits in an audio frame minus the number of bits actually used for the header, error check and audio data. In Layer I and II this number corresponds to the distance between the end of the audio data and the beginning of the next header.

The audio_data block is the main part of the audio frame. Its structure can be subdivided in three sub-parts: allocation data, scale factors related data, and sample data. The format of allocation data is the same in Layer I and II; it consists of a number of allocation slots equal to *sblimit*, the number of subbands encoded for the signal. In Layer I sblimit is fixed to 32, while in Layer II its value depends on the bit rate index and the sampling frequency. In Single channel mode, each allocation slot refers to a subband and specifies the number of bits used to encode every sample of that subband. In Dual channel and Stereo modes, the allocation slots of first and second channel are interleaved for each subband. As shown in figure 3.8.a, in Joint stereo mode only the first *bound* subbands have allocation slots for both channels; for the remaining subbands allocation slots are coded as for Single channel mode.



(a) Layout of scale factor related data for a Layer II frame, consisting of scale factor selection information and scalefactor data.



(b) Layout of sample data for a Layer II frame coded in Joint stereo mode.

Figure 3.9: Layout of scale factor related data (a) and sample data (b) for a Layer II Joint stereo mode frame. The number above data slots represent their bit size.

All the information following the allocation data block is encoded only for those subbands whose allocation vale is not zero. The structure of scale factor related data block depends on the layer. In Layer I frames, it consists of a sequence of scale factors; each scale factor refers to a subband and it is applied to all its samples. In Single channel mode one scale factor is encoded for each subband. In Dual channel, Stereo and Joint stereo, as shown in figure 3.8.b, one value is encoded for each channel.

In Layer II, the scale factor related data consists of two parts: a scale factor selection information block and a scale factor data block. The former is a sequence of scale factor selection information fields. Each field applies to a single channel of a subband and determines how many scale factors are encoded for it. The latter consists of a sequence of scale factors as in Layer I, except for each channel of a subband 1, 2 or 3 scale factors may be encoded, depending on the value of the selection information. Figure 3.9.a shows the layout of scale factor selection information and scale factor data blocks.

The sample data block consists of 12 groups of samples. In Layer I, each group contains 32 samples per channel. In Joint stereo mode, as shown in figure 3.8.c, only the first *bound* subbands have a sample for each channel, the remaining subbands are encoded as in Single channel mode, i.e. only a sample is coded for each subband. In Layer I a sample can be 2 to 15 bits long, depending on the allocation value specified for its subband.

In layer II each group contains information for 96 samples. As shown in figure 3.9.b, three consecutive samples are encoded for each subband. Depending on the allocation value of their subband, three samples can be encoded separately with a number of bits ranging from 3 to 16, or in a single samplecode, 5 to 10 bit long.

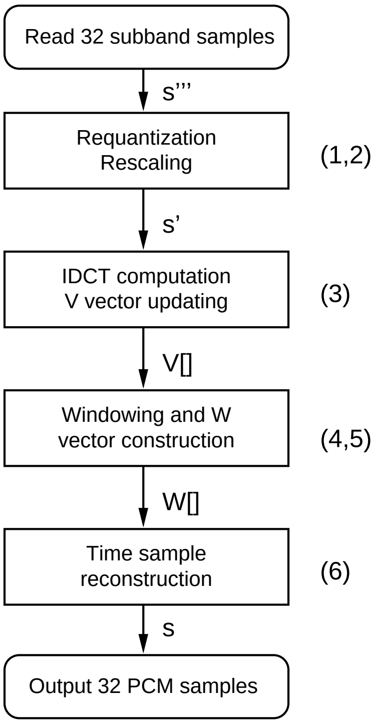


Figure 3.10: Flow chart of synthesis subband filter. The numbers on the right side of each block refer to their algorithms and are described in subsection 3.3.3.

3.3.3 MPEG decoding algorithm

After synchronization and parsing of the allocation and scale factor data, the 12 groups of samples are parsed and processed, one by one. For each group the operations shown in figure 3.10 are performed.

The sample codes retrieved from the stream represent fixed point fractional values in the range $[-1, +1]$, encoded with a precision ranging from 2 to 15 bit for Layer I and 3 to 16 bits for Layer II. As it is shown in Layer II syntax, it is possible to encode three consecutive samples in a single code word. The codeword is treated as an integer. This scheme allows a finer control of the quantization step: samples can be coded with a number of levels different from powers of two. This corresponds to a fractional bit precision. The inverse operation, to retrieve the original sample values, is called *degrouping* and is shown below.

```
for (i=0; i<3; i++) {
    s[i]=c%nlevels
    c=c DIV nlevels
}
```

Where *nlevels* is the number of quantization levels specified in the allocation table, *c* is the codeword and DIV is the integer division.

After left justification to the data word size of the application these samples are applied to the so called synthesis subband filter, which transforms them into 32 samples of the original audio signal. Figure 3.10 shows the flow diagram of the operations performed by the subband synthesis filter. First, samples are re-quantized using the linear formula (3.1), where s''' is the input sample after justification, s'' is the re-quantized sample and *nb* is the number of bits of the original sample value retrieved from the stream, and depends on the allocation value for the subband of the sample. If a subband has no bits allocated to it, the samples in that subband are set to zero.

$$s'' = \frac{2^{nb}}{2^{nb} - 1} * (s''' + 2^{-nb+1}) \tag{3.1}$$

Re-quantized samples are scaled by a coefficient specified by the scale factor index relative to the subband of the sample, using 3.2, where s' is the re-quantized and re-scaled sample.

$$s' = factor * s'' \tag{3.2}$$

After these preliminary steps, 32 reconstructed subband samples are available as fixed point values with the bit precision needed by the application. First samples are transformed back to 64 time samples by IDCT and are stored in a vector V containing the last 1024 computed time samples. The IDCT operation is shown in (3.3) and is performed for index i values ranging from 0 to 63.

$$V[i] = \sum_{k=0}^{31} \left[\cos(16 + i)(2k + 1)\frac{\pi}{64} \right] s[k] \tag{3.3}$$

As the frequency spectrum of the signal is sampled, these values may well present discontinuities across the borders of 64 sample blocks. These discontinuities represent spurious high frequencies not present in the original signal. To avoid this effect, the elements of the V vector are filtered to smoothen the reconstructed signal. Before encoding it must therefore be pre-processed to compensate this effect. The filtering algorithm is ordered in three steps. A half sized vector U is first derived out of the V vector by algorithm (3.4):

```
for (i=0; i<7; i++) {
      for (j=0; j<31; j++)
      U[i*64+j]=V[i*128+j]
      U[i*64+32j]=V[i*128+96+j]
}
```
(3.4)

Each element of the U vector is then multiplied by a window coefficient, as shown in (3.5). The operation is performed for index i values ranging from 0 to 511. The value of window coefficients D is part of the MPEG standard. The result of the windowing operation is stored in a vector W.

$$W[i] = U[i] * D[i] \tag{3.5}$$

To reconstruct the final output samples, the elements of W corresponding to samples in the same position within the group are summed as shown in (3.6). The operation is performed for index i values ranging from 0 to 31. The result values are the filtered output samples $s[i]$.

$$s[i] = \sum_{k=0}^{15} W[k * 32 + i] \tag{3.6}$$

This process is repeated for each of the 12 blocks in a frame. The resulting output is a sequence of 12×32 time samples for Layer I and 36×32 samples for Layer II.

The operations here described differ from any real implementation of MPEG Audio decoder. Some straightforward optimizations are common to every implementation. For example, the elements of the V vector are never shifted by one block before each execution of the IDCT transformation; the V reduction to the U vector is usually avoided, as all the computations starting from (3.4) to the output of sample reconstruction are usually performed ‘in one shot’ for each sample, to reduce the memory demand. The choice of the best algorithms depends also on the hardware architecture of the application. Parallel reconstruction of output sample is also common in many realizations and strongly influences the way algorithms are implemented. In section 4.3 the optimizations adopted for an implementation based on custom ASP, are shown in detail.

SP8881 MPEG Audio decoder 4

This chapter introduces the hand-crafted implementation of the MPEG Audio decoder designed by SPaSE B.V. The decoder is based on a custom Application Specific Instruction set Processor (ASIP) for digital audio processing. Section 4.1 describes the hardware architecture of the audio processor, which is part of a multimedia processor for CD-i decoding, called *SP8881*. Its function units, instruction set and machine description file are presented in detail. Section 4.2 describes the software framework used for the application and the design trajectory, which comprises assembling and simulation steps. A description of the assembly format recognized by the assembler and the input/output data files involved in the software trajectory completes this section. In section 4.3 specific details of this MPEG audio decoder implementation, like major optimizations applied to the critical parts of the decoding algorithm are given. Section 4.4 begins describing the extensions introduced in the analysis data set produced by the simulation. These extensions were necessary to have comparable data sets from the simulation of SPaSE's decoder and the version designed by means of the MOVE framework. The analysis of simulation results and their discussion conclude this section. A summary of the SPaSE implementation characteristics and the information obtained from simulation concludes this chapter.

4.1 Architecture of the SP8881 audio processor

This section describes the processor architecture of the MPEG Audio decoder, whose algorithm is detailed in chapter 3. First, a brief history of the company and its current projects are given. The whole SP8881 project, which the decoder is part of, is shortly presented here. The rest of this section treats the hardware configuration of the SP8881 audio module (referred to also as audio processor) and it is articulated in two parts: the former gives a general description of the architecture and its components, the latter presents the machine description file of the architecture.

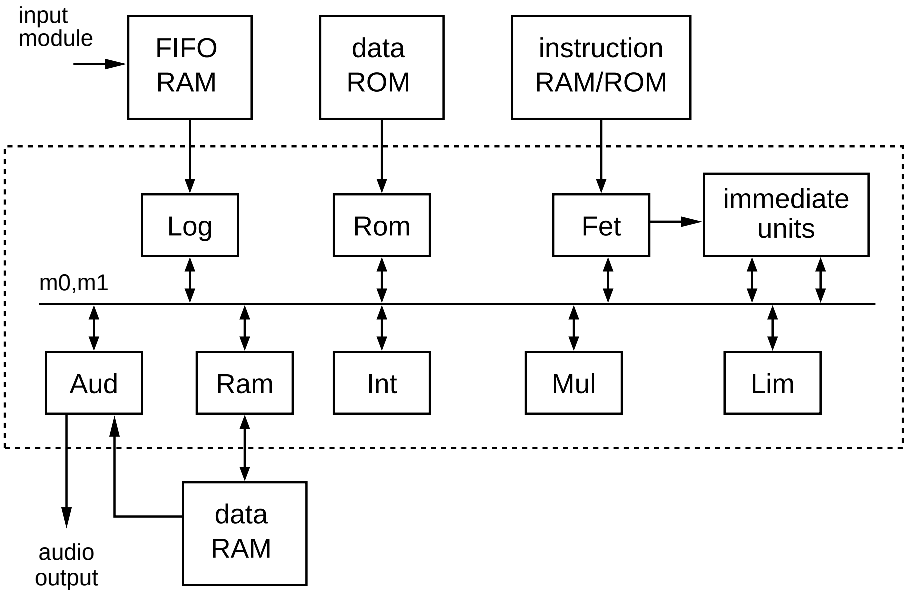


Figure 4.1: The structure of audio part of SP8881 and its data path connections with the memory banks. Note that RAM bank contains a shared buffer with external modules used for output data. The log unit is connected to a special RAM bank dedicated to the two input FIFO buffers.

SPaSE B.V. is a Dutch company founded in 1989 and located in Nijmegen. Its main field of activity is the design of integrated circuits for multimedia applications for the consumer market.

Several projects developed or under development are based on the TTA concept, that proved to be suitable for highly optimized and efficient embedded system applications based on low-end hardware configurations. One of such projects, currently under development, is SP8881, a single-chip decoder for CD-i audio and video. This processor will be included in a *set-top* box for satellite signal decoding, computer interfacing and video-on-demand. The CD-i audio decoder and mixer module of SP8881 are implemented by a small TTA configuration. It is this module that was chosen as test ASP and application to compare a code compiled by the MOVE software subsystem with.

Currently, no TTA synthesis framework is available for projects like this at SPaSE. Every aspect of their project is designed without the aid of any automatic tool for processor and code generation. This approach allows to obtain optimal hardware configurations, possibly not achievable with a templated architecture, but it requires a long design time. Nevertheless, the major drawback of the lack of a fixed framework is probably that a large part of the design efforts are not re-usable, as every architecture is designed independently.

4.1.1 Description of the hardware configuration

The purpose of SP8881 audio module is to concurrently read an ADPCM and an MPEG Audio stream, to decode them and to mix the reconstructed audio signals into a single PCM signal, in real time. The two input streams come from the same CD-i storage unit and are buffered by an external module into a dedicated RAM bank. The output signal is buffered into a shared area of data RAM and is read by the audio output module, which is internal to the audio processor. Apart from the proper decoding operations, the audio processor must communicate and adapt its decoding rate to input/output buffers. Two external modules write their input buffers and one module reads its output buffer. For this purpose a set of interrupts and RAM mapped command

FU	Instruction set description	Instructions
Mul	Integer and fixed point multiplications.	`mul`, `mulfx`
Int	Arithmetic operations.	`add`, `sub`, `subeq`, `subne`, `subgt`, `suble`
Lim	Accumulator based arithmetic operations with result limitation.	`addl`, `subl`, `accl`, `nacl`
Log	Boolean and shift operations. Input stream and crc check operations.	`and`, `or`, `xor`, `shl`, `shr`, `inrj0`, `cur`, `cnt`, `crc`
Rom	Rom bank load operations based on auto-incremented pointers and wrap-around control.	`r0rom`, `r1rom`, `r0romi`, `r1romi`, `rom`
Ram	Ram bank load/store operations based on auto-incremented pointers. Operations for output buffer management.	`r0ram`, `r1ram`, `r2ram`, `r3ram`, `r0rami`, `r1rami`, `r2rami`, `r3rami`, `rgram`, `w0ram`, `w1ram`, `w2ram`, `w3ram`, `w0rami`, `w1rami`, `w2rami`, `w3rami`, `wgram`, `rpram`, `wpram`
Fet	Jump and loop operations.	`jump`, `jumps`, `loop`

Table 4.1: Instruction set of the audio processor. Only instructions used for MPEG Audio decoding are shown.

registers are provided, to signal underflow and overflow of buffers and to make the processor take the necessary actions.

Figure 4.1 presents a simplified view of the audio processor architecture. The processor consists of 8 function units, connected by 2 move buses: m0 and m1. Two immediate units are also present, one connected to bus m0 and one to bus m1. No general purpose register unit is present: all operations are performed using function unit registers and memory as storage for temporary values. A one-bit boolean register is present. All FU and bus data paths are 20 bits wide, except for the Lim FU, which has an internal data path of 25 bits. The processor is connected to four memory banks: instruction RAM and data RAM are made of 2048 words each, data ROM and FIFO RAM contain 1024 words. Every memory space is thus addressable by 11 bits.

Three of these function units, Mul, Int and Lim, have the standard structure of a MOVE TTA FU, consisting of two read sockets for operand and trigger registers and one write socket for the result, all fully connected with both buses.

Figure 4.2 shows the internal structure and the socket connectivity of the last four function units. Table 4.1 presents the instruction set of the audio processor used for MPEG decoding and how it is subdivided among its FUs. Below a description of each FU is given.

Mul provides integer and fixed point multiply operations `mul` and `mulfx`, respectively. For fixed point multiplication it performs convergent rounding on read.

Int provides integer arithmetic operations. Four versions of `sub` operation set the boolean register with a specific condition result as a side effect of the operation.

Lim (Limited) provides only add-subtract operations, both in normal and accumulate mode. The operand and the result registers are 25 bits, in order to maintain accumulated results bigger than the normal operand size. The result is saturation limited on read to 20 bits.

Aud is the output module of the audio processor. It is not used in the MPEG decoding algorithm.

Log provides boolean and shift operations. In addition, four special instructions for bit field parsing and input stream control and one instruction for CRC check (see section 3.3 are present. As shown in figure 4.2, the Log FU contains 5 physical registers: oper, trig and res are the usual tree address operations; syn keeps the current value of the CRC check word, and cnt counts the number of words of the current audio frame to read from the MPEG input stream.

Rom provides instructions to access the ROM bank through two pointers, (ptr0 and ptr1) both in normal and auto-increment modes. A wrap-around feature modulo a memory page of 128 words is also present. The Rom FU contains three physical registers: ptr0 and ptr1 are the current memory addresses; out contains the value of the last word read from ROM. The location addressed by the last incremented pointer is pre-fetched into the out register.

Ram provides the instructions to access the RAM bank through five pointers and to control the output PCM buffer. The registers ptr0, ptr1, ptr2, ptr3, and gad contain the current memory addresses; the first four can be used both in normal and auto-increment modes. The register gdat contains the last value written into memory; out contains the last read value. The last two registers, pad and pst, are used to control the output buffer: pad points to the last element of the buffer that has been read by the output module; pst points to the last value of the buffer that the output audio module is allowed to read without incurring in a buffer underflow exception.

Fet (fetch) corresponds to the Instruction Unit of the MOVE Framework and provides the control flow instructions. In addition, three instructions present in the code for simulation only, for ease of debugging, are formally mapped into the Fet FU. As shown in figure 4.2, the Fet FU contains 8 registers. The registers lpno, lpsa and lpea are dedicated to the loop instruction: the former contains the number of iterations to be performed, lpsa points to the first instruction of the loop, and lpea points to the last instruction of the loop. The register pc is the program counter; ret contains the pc value incremented by one and it is used to save the return address. The registers brkt, brka, sig are used for trap and interrupt management.

4.1.2 Machine Description File

One of the data files taken as input is the machine description file "mach.mac", which specifies the architecture configuration and the assembling process. The machine description file developed for the audio processor consists of 8 specification blocks. This file, as for the MOVE framework, fully specifies the TTA hardware configuration. Below are listed the specification blocks contained in mach.mac:

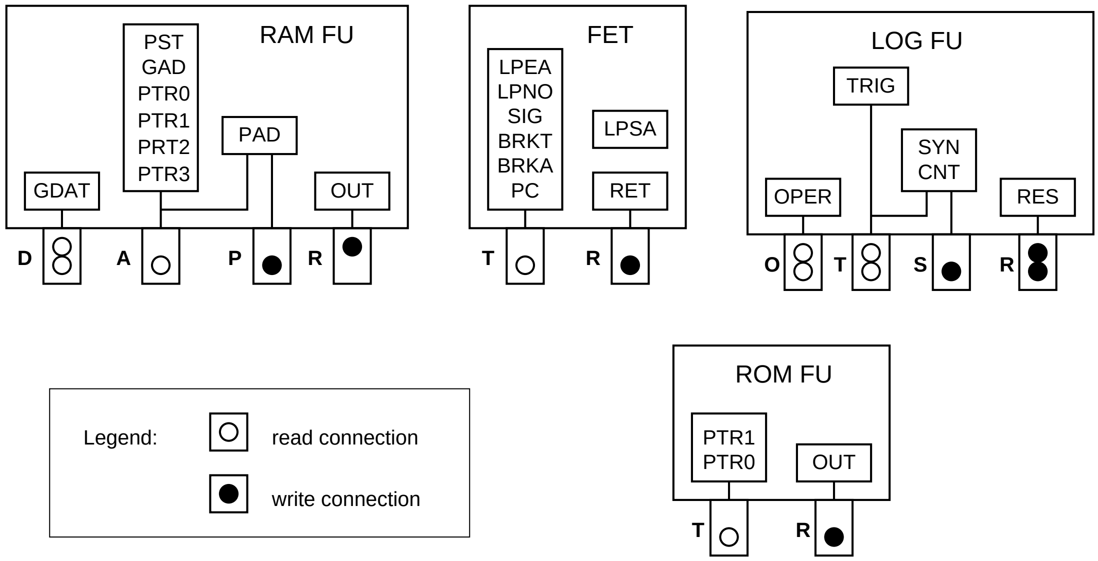


Figure 4.2: Internal registers and connections to sockets for the four FUs not in compliance with the MOVE FUs templated structure.

1. Operations
2. MoveBusses
3. Sockets
4. FunctionUnits
5. Source Map
6. Destination Map
7. Memory
8. Format

Three of these blocks are also present in the machine description file of the MOVE TTA framework: MoveBusses, Sockets and FunctionUnits. More specifically, MoveBusses block is syntactically identical in the two machine description files, while the FunctionUnits block shows a different syntax and different semantics, and the Sockets block has a reduced syntax. Figure 4.3 shows a part of the audio processor machine description file.

The Operation description block specifies the name of every instruction recognized by the assembler. Each instruction can have several name specifications for its operand, trigger and result registers. These names address different physical registers of the FUs; they may be considered as names of logical registers, resembling the usage that is made of their move. For example, `jump.ad` and `jump.ra`, indicate an address move destination and the return address move source respectively. Similarly, `op`, `tr` and `rs` correspond to the standard MOVE FU registers for operand, trigger and result moves.

The MoveBusses description block specifies name, width, maximum immediate size and signedness for each bus present. The Sockets description block specifies name and access type of each socket present. Unlike in the MOVE machine description file, the connections between sockets and buses are not explicitly specified in the sockets block. They are determined by the mapping specification blocks described below, instead.

The FunctionUnits description block specifies name, latching discipline and internal registers of the audio processor FUs. As shown in the example, the latching discipline used for the

```
Operations {
//     Oper   Trig   Result
jump    {},   {ad},   {ra};  // jump & exec next instruction
add   {op},   {tr},   {rs};  // add
r0rom   {},   {ad},   {dt};  // read using rom pointer 0
}
MoveBusses {
m0  20,  8,  signed;
m1  20,  8,  signed;
}
Sockets {
sint_o input;
sint_t input;
sint_r output;
}
FunctionUnits {
//   Type     Lat   Phys-registers
int  always,  1,  { oper, trig, res };
log  always,  1,  { oper, trig, syn, cnt, res };
rom  always,  1,  { ptr0, ptr1, out };
}
Source Map {
//           Src-id   Transport path     Occ
r0romi.dt  { 0x010 = m0.srom_r.rom.out : -1 };
add.rs     { 0x019 =  *.sint_r.int.res :  0 };
jump.ra    { 0x01E = m1.sfet_r.fet.ret :  0 };
}
Destination Map {
//           Dst-id   Transport path     Occ
r0romi.ad  { 0x024 = m0.srom_t.rom.ptr0 : 0 };
add.tr     { 0x010 =  *.sint_t.int.trig : 0 };
jumps.ad   { 0x03D = m1.sfet_t.fet.pc   : 0 };
}
Memory {
//   Type         Access  Fill      Wid Size
iram instruction, bidir,  upwards,  24, 2048;
dram data,        bidir,  upwards,  20, 2048;
drom data,        output, upwards,  20, 1024;
}
Format {
//    Bus Type         Pos Wid Nop
field m0, guard,       23, 1,  0x0
field m0, source,      18, 5,  0x00;
field m0, destination, 12, 6,  0x3F;
//        Bus Map-type     Src-id Wid  Range
immediate  *, source,      0x00,  5,   0..0xF;
immediate m0, move,        0x1E,  12,  0..0xFFF;
immediate  *, instruction, 0x1F,  24,  -0x80000..0x7FFFF;
}
```

Figure 4.3: Example of machine description file. The lines of each block are taken from the description of the audio processor.

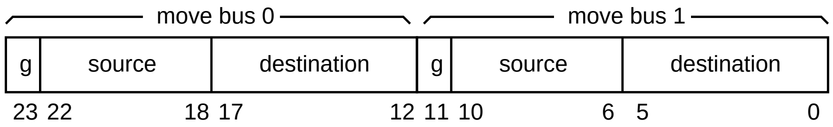
FUs is ‘always latching’. This means that both operand and trigger moves can change the FU result, and it corresponds to the True Virtual Time Latching Pipeline described in [4]. The result value is available until it is overwritten by the result of the subsequent operation. The latency field is only relevant for jump instructions: it specifies the number of instructions that will be executed before the jump has actually been completed. This piece of information is therefore not complete: different operations of the same FU can have different latencies; their value is not specified in the mach file. For each FU the entire set of physical registers present in the unit must be specified. The result registers are not actually present in hardware and their name as indicated in the register list is just a reference for socket mapping. The connections between physical registers and sockets are specified by the source and destination map blocks. There are no restrictions for FU register and socket specifications. The MOVE framework version of function units specification block contains as further parameter of each FU the set of operations which can be performed by the FU. This piece of information is also implicitly established in the source and destination map blocks.

The Source Map specification block, together with the Destination Map, specify the mapping process from assembly mnemonics to binary vectors of instructions. Each line specifies the transport path of a move, i.e. the move bus, socket, FU and physical register involved in the data transport. Source and destination IDs are assigned to transport paths. IDs are hexadecimal values found in the source and destination fields of binary code move slots. Used source IDs range from 0x00 to 0x1F; IDs from 0x00 to 0x0F represent immediate source values. It is important to remember that the same source ID can refer to different operations when mapped in different buses, thus the meaning of a source ID depends on the move slot. For example, the source ID 0x1E specifies short immediates (or move immediates) when it appears in the first slot (bus m0), while it specifies the return address register if it is found in the second slot (bus m1). This scheme implicitly defines a reduced connectivity network and allows a better exploitation of the ID space. If both buses are available for a move an asterisk `*` replaces the bus field in the specification line. Many different operations share the same source ID for their result move, as this transport path itself is independent from the operation type that was formerly triggered. Used destination IDs range from 0x00 to 0x29 and from 0x3A to 0x3F. It is worth noticing that many different operations share also the same destination ID for their operand move, as their transport path does not imply the operation that is going to be triggered by the subsequent (or concurrent) trigger move. However, operand moves do change the FU result: the operation executed depends on the opcode specified by the last trigger move on the FU.

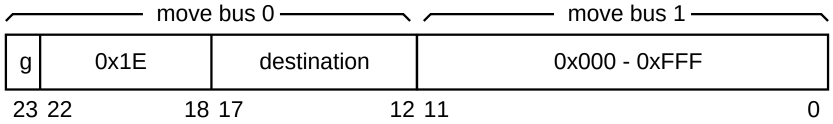
The Memory block specifies how memory is partitioned into multiple banks and which are their characteristics. The name of the memory bank is used as a storage qualifier in assembly code to specify which bank the code is to be loaded into. For example, functions will be loaded in instruction RAM, constant data like coefficient tables will be loaded in a ROM and variable data will be loaded in a RAM. The Memory block in figure 4.3 shows the specifications for the memory banks used by the audio processor. Note that the FIFO RAM is not declared, as it is not directly accessible.

The Format block specifies the instruction word format. The keyword `field` introduces an instruction field, and it can be one of: move source, move destination, guard. For each field the starting bit position and its size within the instruction word are specified. The value which represents a null operation is also specified. Figure 4.3 shows the specifications for the first move slot of audio processor instructions.

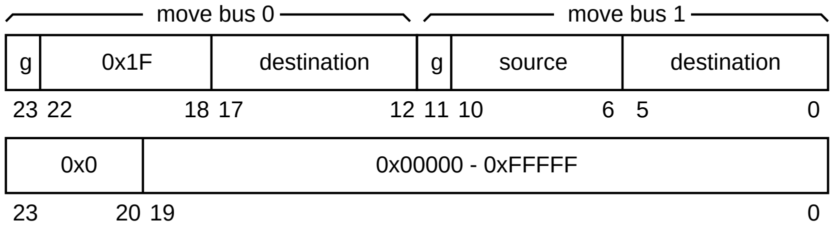
The keyword `immediate` introduces an immediate field description. A single immediate field can be as long as one move slot or an entire instruction. The bus field specifies the bus used for the transport of the immediate value.



(a) Instruction word with normal IDs for registers and source immediates.



(b) Instruction word for short immediate.



(c) Instruction words for long immediate.

Figure 4.4: Instruction formats and immediate sizes supported by the audio processor: (a) source immediate, (b) short immediate, (c) long immediate.

Figure 4.4 shows the instruction formats of the audio processor for immediate values. A source immediate value takes the source field of its move, as shown in figure 4.4.a. A short immediate value takes the entire move slot of m1 bus, therefore only one move with short immediate can be specified per cycle. The bus taken for its transport is m0, that is the one carrying the source ID for short immediates (0x1E); this means that only bus m0 can be used for short immediates and that bus m1 is not utilized in that cycle. It is possible to specify only one long immediate field per instruction, but unlike the short immediate format, both slots can be specified, by means of the source ID 0x1F. The other move slot can therefore contain a source immediate or specify a FU register. The next instruction word of the instruction stream is taken as value of the long immediate.

4.2 The software design framework

To design the SP8881's audio module software a set of tools was developed, to facilitate assembling, testing and debugging of the application. It consists of a move assembler, strongly parameterized on the hardware configuration specified in mach.mac, and a C module for software simulation. An assembly instruction format, called *Mas* code, was also defined to represent audio processor programs. In this section first the assembler and its input and output data files are presented, then a brief description of the assembly code format is given and finally the simulation process is treated.

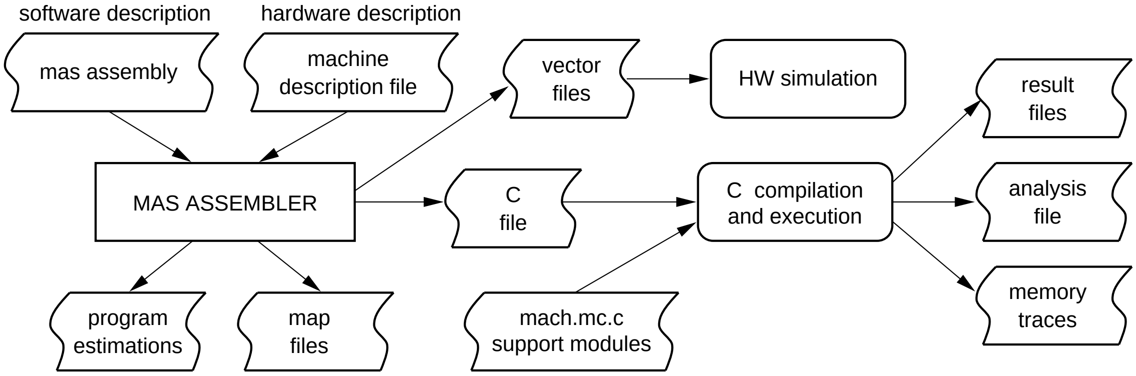


Figure 4.5: The software trajectory. Assembling of Mas source files and simulation of the resulting C code.

4.2.1 The Mas assembler

As shown in figure 4.5, the Mas assembler takes as input a Mas assembly source file and a machine description file called "mach.mac". The output produced by this tool is fully parameterized on the machine configuration described in the latter. The assembler has two functioning modes: it can assemble Mas source files into machine code or it can translate them into a C source file.

In the first case, it produces three output files, one per each memory bank, containing the binary vectors to be loaded into instruction ROM, data RAM and data ROM for the real program execution. These files are also used for low level simulation of the hardware description, but this remains out of the software trajectory. In the second mode the move assembler produces a C source file used for simulation and result analysis.

Several files containing additional information are generated by executing the move assembler on a Mas source file:

1. Program estimations based on static profiling.
2. Map files of memory banks.

During assembling, the Mas assembler estimates several program characteristics and requirements. These estimations comprise function code size and rough estimations of function trace length, bus load, socket load, instruction and FU usage. All these estimations are based on an optional field that can be present in every block of assembly code declaration, which specifies the expected number of times the block is entered per each call of its function.

After the proper assembling step, in addition to the three vector files three more files, called map files, are generated. A Map file describes the allocation map of a memory bank: for each object allocated its name, starting address and length is specified. An object is an instruction block for the instruction RAM/ROM, a variable or a matrix of any dimension for the data banks. The map file of data RAM includes the locations of the static stack area assigned by the assembler to the local variables of each function.

4.2.2 The assembly code format

The assembly code defined for the SP8881's audio processor is structured in a nested hierarchy of source modules, functions, instruction blocks, instruction slots and move slots.

Structure of the assembly code

A Mas function is declared in a way similar to C functions. The first field specifies the memory bank into which the machine code will be loaded. This field can be considered as a storage class specifier of C/C++ languages. The second field specifies the return type. Only void is used. The third field is the name of the function. Here is an example of function header:

```
iram void Parse_LI()
```

Each function is structured in *blocks*, introduced by a declaration with the format:

block_name [`frequency` freq]

The optional field, introduced by the `frequency` keyword, gives an estimation of the number of times the block is entered for each function call, and is used to compute the estimations mentioned in section 4.2.1. Blocks are not necessarily basic blocks: they may include one or more jumps. The main purpose of blocks is to provide a label referencing to the first instruction; in fact, jump instructions can have only block names as destination. A Mas function can be freely split into a number of blocks higher than what strictly necessary.

Each Mas instruction is made of one or two moves and is terminated by a semicolon. A comma separates the move slots, if they are both present. A single move instruction refers to the bus 0, while a single instruction preceded by an empty move slot, denoted by '`--`', refers to the bus 1. FU registers for move sources and destinations consist of an instruction mnemonic followed by a period and the name of the logical register. Immediates can be represented by expressions, including constant symbols. Data memory addresses and function addresses are indicated by their name preceded by '`&`'. Some examples of Mas instructions are shown below.

```
MD_SNGL -> sub.op,                  rgram.dt -> subeq.tr;
&ad_bit_rate -> rgram.ad;
0x00 -> addl.tr,                    rgram.dt -> addl.op;
--,                             bl: rgram.dt -> addl.tr;
```

4.2.3 The simulation of Mas code

There is no simulator in the sense of a tool reading and interpreting the Mas code. The simulation process consists of two steps: (1) compilation of the C code equivalent to the Mas source and link to the simulation module mach.mc.c; (2) execution of the executable file obtained.

The C source produced by the assembler is a straightforward translation from the Mas code and mainly consists of calls to functions that simulate the behavior of moves. The definition of all these functions is contained in a fixed C module called mach.mc.c, which represents a complete algorithmic description of the chip functioning. Unlike the code produced by the assembler, this module is not parameterized on the machine description file. It is worth to notice that this module contains details of the hardware model not present in the machine description file, like the different latency of single instructions. The 'C translated' code and mach.mc.c are compiled and linked to other two modules, debug.c and analysis.c, which are necessary to print-out tracing information and analysis data respectively. The executable file obtained comprises therefore the simulator and the code to be simulated in a single program. The best quality of this approach is that it allows rather fast simulations: there is no interpretation of Mas assembly, the code for the simulation is compiled and runs on the workstation architecture. Its main drawback is the lack of flexibility: changes in the Mas code or in the hardware configuration require a new compilation step. As shown in figure 4.5, each time the executable is run the following files are generated:

1. Result files: V vectors, S vectors and PCM files.
2. Analysis files: execution and parsing trace.
3. Memory trace files.

The result files are generated during simulation by the code stored in debug.c. The V vector file contains the printouts of the subband synthesis filter state after the parsing of each block. The S vector file contains the PCM output samples obtained after the subband filter. The PCM files contain the fixed point values of output PCM samples in a format suitable for generating plots.

The execution trace file contains information on the bit stream parsing process. It also contains cycle and operation counts at fixed positions in the program, called mark points, which correspond to debug moves. The information originally contained in this file is far from sufficient for a complete comparison of the two decoder programs.

The memory trace files contain the sequence (in execution order) of the memory accesses, and, along with map files, are a useful means for debugging.

4.3 The MPEG decoder implementation

The efficiency of the decoding algorithm developed at SPaSE heavily relies not only upon the available hardware functionality, but also on several software optimizations. For example, the subband synthesis filter described in subsection 3.3.3 requires 64×32 fixed point multiplications for each block of samples, while the optimized code reduces this figure to 32×16.

In this section the optimizations of the decoder implementation developed at SPaSE are described. Most of these algorithms have been directly copied into the MOVE version of the decoder. The MPEG format recognized by the decoder is fully CD-i compliant and represents a two channel subset of the MPEG-1 and MPEG-2 Audio format [1], described in section 3.3. First some specific format issues are briefly described, then the main optimizations applied to the critical functions of the subband synthesis filter algorithm are explained.

4.3.1 CD-i MPEG format

The MPEG-1 Audio bit streams which can be read by the decoder are incoming from a CD-i storage support. Such streams are compliant with the CD-i format recommendations for MPEG standard and present some limitations in the parameter ranges allowed with respect of the full standard detailed in section 3.3.

1. The sampling frequency value is limited to 44.1 kHz, the Compact Disc sampling frequency.

2. The value of the bit rate cannot be freely defined by the user, only values defined in the bit rates tables of Layer I and II are possible.

3. The deemphasis scheme CCITT J.17 need not to be applied, only $50/15\mu s$ and no deemphasis at all are allowed.

When reading MPEG data from a CD-i, the input module inserts "on the fly" a 32 bit word for each frame, just after the first 16 bits of the header. Therefore, though data on the storage support are fully MPEG compliant, the stream submitted to the decoder is not. This feature has been excluded for the testing. The 32 bit word contains a Decoding Time Stamp reporting the

[1]MPEG-2 Audio is supported limitedly to the backward compatible subset.

time the first sample of the frame should be played. This data must be dispatched directly to the output module along with the PCM decoded samples.

4.3.2 Matrix function optimization

The matrix function performs the IDCT computation reported in subsection 3.3.3. First of all, a table of 128 coefficients $N[\,]$ substitutes the cosine coefficient computation:

$$N[i,k] = \cos(i+16)(2k+1)\tfrac{\pi}{64} \quad (0 \le i \le 63, \quad 0 \le k \le 31) \tag{4.1}$$

The table N presents four symmetries that can be exploited to avoid part of the computations:

$$\begin{array}{rlll} N[i,k] = & N[i,31-k] & \text{if } 0 \le k \le 31 & i \text{ even} \\ N[i,k] = & -N[i,31-k] & \text{if } 0 \le k \le 31 & i \text{ odd} \\ N[i,k] = & -N[32-i,k] & \text{if } 0 \le i \le 31 & \\ N[i,k] = & N[96-i,k] & \text{if } 32 \le i \le 63 & \end{array} \tag{4.2}$$

The first two equations in (4.2) are exploited to reduce the number of expensive fixed point multiplications by half. For each couple of samples having the same N coefficient first their sum and their difference is computed and stored in a vector of 32 elements, called mirror vector. Then these elements are multiplied by their coefficients. This way, for each sample transformation 16 fixed point multiplications between mirror vector elements and cosine coefficients are sufficient instead of 32 between cosine coefficients and subband samples.

The last two equations in (4.2) imply that the blocks of the subband synthesis filter vector posses the same symmetries. This gives the possibility to reduce IDCT computations only to a part of the original V vector: only 34 elements per block out of 64 are unique, therefore 544 elements of the original V vector are sufficient to carry all the information of 1024 samples. Let us consider the first block of the V vector. Only its first 16 elements and the elements between $V[49]$ and $V[63]$ are computed. The value of $V[16]$ is always zero, while $V[63]$ equals to $-V[0]$. This further reduces the number of fixed point multiplications by half.

4.3.3 Windowing algorithm optimization

A first, straightforward optimization consists of avoiding to shift the V vector by one block at each execution of the subband filter algorithm. Instead, a pointer to the last block computed is updated and the values of the new block are written starting from the new position, overwriting the values of the oldest block. The reduction of V vector into a U vector described in section 3.3.3 is an expensive and unnecessary operation that can be avoided by accessing the V vector appropriately. To make the access more regular V is split into two vectors, called Vlo and Vhi, containing the even and the odd blocks of the V vector respectively. Figure 4.6 shows the subdivision of V into blocks and the rearrangement of these blocks into Vlo and Vhi. The blocks selected for the windowing algorithm are indicated by the shading. The regular path used to select blocks from Vlo and Vhi is shown explicitly. The access of Vlo and Vhi is split into two loops per each vector. In this way, much smaller and efficient loops can be coded. Note also that, due to the 3rd symmetry in (4.2), the computations for Vlo must be split in two parts, as the first part of the sum requires an add-accumulate, while the second requires a subtract-accumulate.

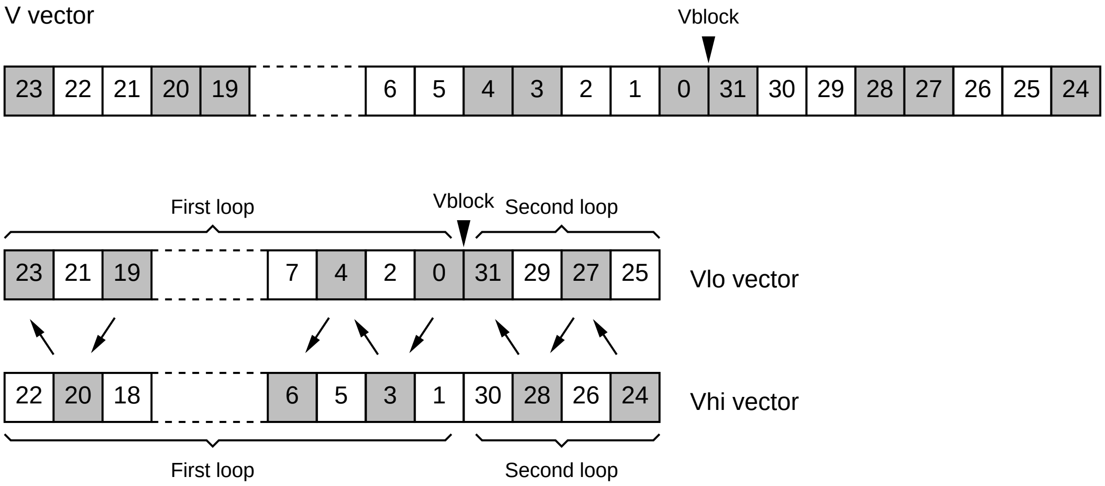


Figure 4.6: The V vector is subdivided into 16 blocks of 64 samples, which are further subdivided into parts *high* and *low*. The shaded blocks are selected for the windowing step. The number indicates the blocks order of computing: the higher, the older. The regular access path used for Vlo and Vhi element selection is split in two loops and is indicated by the arrows, which start from block 0.

4.4 Simulation trajectory of the decoder

In order to perform the comparison between the two implementations of MPEG Audio decoder we need (1) comparable hardware configurations, in terms of resources, (2) extended and comparable information, and (3) equal program flows. The first point is treated in section 5.1, with the hardware configuration design of the MOVE decoder. The modifications needed to achieve the two latter are described in the first part of this section. The Mas decoder was changed to adapt its execution flow to the decoder developed by means of the MOVE framework, presented in subsection 5.2.2. The simulation of the Mas decoder was then run on two files containing MPEG Audio streams, one for Layer I testing and one for Layer II. The simulation results, consisting of execution time spent in each function and operation counts, are presented and conclude this section.

4.4.1 Changes to the simulator

To perform the comparison, the same set of data were needed as result of the simulations: bus load, FU usage count, instruction count. All of these were not present in the original version of the Mas simulator. Several extensions were therefore added to the debug.c module and a new module called analysis.c was introduced to define the data structures for analysis information. The simulation environment was set up to extend the data set computed during simulation: execution time spent in each function and operation count have been added, as well as FU and bus utilization counts. None of these extensions presented particular problems except for the assignment of instruction counts. In fact, while in the MOVE framework a trigger move may be counted approximately as the operation unit, in the TTA architecture of SP8881 audio processor the trigger move and the result move of some instructions (cnt, crc, wpram) do not have the functional dependency of the MOVE FUs. These result moves have been counted as independent operations. For example, the cnt instruction mnemonic is used both to set the word counter for stream 0 and to read the current number of words yet to be read. Furthermore, some operations, like load from memory and read from input stream, are triggered by the result move. No trigger move, as defined in the MOVE framework, is needed, only a single, preliminary operand move to set the internal state of their FU. These result moves are also counted as operations.

4.4.2 Changes to the Mas source code

MPEG Audio decoding is just a subset of the operations performed by the Mas code developed for the audio processor. All the extra functionalities performed in addition to MPEG decoding had to be removed to permit comparable measurements of execution counts and resource usage. In fact, after the simplifications described below, the execution time of the Mas decoder dropped by roughly 50%. These are the changes introduced in the Mas code:

1. Disabling of ADPCM decoder and mixer routine.
2. Simplification of output buffer and disabling of its interrupt and execution lock.
3. Elimination of code for external command management.

Besides MPEG decoding, the application developed at SPaSE reads an Adaptive Pulse Code Modulation audio signal, decodes it and mixes the two audio sources. The mixer routine is parameterized by the value of a set of command registers controlled by external modules. The ADPCM decoding part of the mixer has been removed, the proper mixing routine has been furtherly simplified in rescaling the output signal amplitude.

A *buffer overflow* execution lock is thrown when the output buffer is full. The original simulation of the Mas code showed that almost half of the execution time was spent waiting for the simulated output module to read the decoded samples, as the decoding rate is roughly double than the rate required for a real time audio reproduction. The simplified mixer routine avoids overflow by manipulating the output buffer pointers.

The execution of the decoder application is controlled by means of registers set by external module. The values of these registers control the decoding mode (MPEG, ADPCM, or both), the mixing parameters (for example the volume of each channel and the balance of the two audio sources). The simplified version adapted for the comparison simply ignores these commands and always performs just MPEG decoding. The main module flow-chart has been changed to conform to the scheme shown in figure 5.2, consisting on a loop performing a fixed number of frame parsing and decoding.

4.4.3 Simulation results

The MPEG streams used for the simulation represent non-pre-emphasized stereo signals sampled at 44.1 kHz, without CRC protection.

Table 4.2 shows the sizes of the Mas functions which form the decoder, as they are printed in the file containing static estimations computed during the assembling phase. Most of these functions are present in the MOVE decoder and have exactly the same functionality. Another important piece of information provided during the assembling stage in the data RAM map file is the size of the static stack. The current decoder requires only 11 word static stack, indicating very low requirements for temporary values.

Table 4.3 shows the execution time spent in each function of the SP8881 MPEG decoder. It is remarkable that the functions related to the subband synthesis filter algorithm, namely Window(), Matrix() and Reconstruct(), take more than 88% of total execution time. It is also interesting to point out that the parsing routines are responsible of a minimal part of cycle count: 5.8% for Layer I test stream and 3.21% for Layer II test stream.

Function name	Description	Size
Parse_LII	Parsing layer II scale and samples	367
Parse_LI	Parsing layer I scale and samples	139
Window	Time sample reconstruction	122
Matrix	IDCT computation on subband samples	99
Deemph	Time sample deemphasis and rescaling	93
Parser	Parsing of allocation data	92
Reset	Global data initialization	80
Frame	Frame length computation	75
Sync	Audio frame synchronization	69
Reconstruct	Re-quantization and rescaling	55
main	Main frame loop, fixed to 3 frames	52
Mixer	Time sample writing to output buffer	47
Parse_hdr	Parsing of the frame header fields	34
Degroup	Degrouping of layer II granules	32
CRC_check	Parity check of allocation data	29
Frame_end	Discard ancillary and padding bits	26
Simulate	External environment setup	26
Read_hdr	Reading of frame header	20
Total		1431

Table 4.2: Sizes of the function of the Mas decoder, expressed in 24 bit instruction words.

Function	Layer I			Layer II		
	Cycle count	Cycle %	Calls	Cycle count	Cycle %	Calls
Window	288632	51.70	72	866792	52.50	216
Matrix	153792	27.55	72	461376	27.94	216
Reconstruct	49812	8.92	36	150012	9.09	108
Parse_LI	29841	5.35	3	0	0.00	0
Parse_LII	0	0.00	0	50154	3.04	3
Mixer	15480	2.77	36	46440	2.81	108
Deemph	15120	2.71	36	45360	2.75	108
Degroup	0	0.00	0	25344	1.53	528
Reset	2747	0.49	1	2747	0.17	1
Parser	2091	0.37	3	2160	0.13	3
Frame_end	239	0.04	3	239	0.01	3
Frame	150	0.03	3	156	0.01	3
Parse_hdr	138	0.02	3	138	0.01	3
main	82	0.01	1	85	0.01	1
Read_hdr	75	0.01	3	75	0.00	3
Sync	53	0.01	1	53	0.00	1
Simulate	31	0.01	1	31	0.00	1
Total	558283	100.00	–	1651162	100.00	–

Table 4.3: Execution time spent in each decoder function, percentage of the total cycle count and number of calls for three frames decoding. The first three columns are obtained decoding three Layer I frames. The last three are obtained decoding three Layer II frames.

Layer I		Layer II	
Operation	**Cycle count**	**Operation**	**Cycle count**
r0rami	102956	r0rami	305447
mulfx	79947	mulfx	239523
r0romi	76944	r0romi	232128
accl	64512	accl	193536
addl	24989	addl	78443
add	20341	add	59657
r1rami	19776	r1rami	58752
subeq	13283	rgram	38509
rgram	12115	subeq	37934
w2rami	10848	w2rami	32460
nacl	9216	nacl	27648
sub	7119	sub	19007
subgt	6144	subgt	17502
w1rami	4707	r2ram	14019
r0ram	4617	r0ram	13851
r2ram	4611	and	13252
loop	11246	loop	33808
jumps	5882	jumps	17828
jump	5848	jump	13084
Total	510956	Total	1519520

Table 4.4: Most executed operations in Layer I and Layer II simulation.

Table 4.4 shows the count of 20 most executed operations for Layer I and Layer II simulations. It is interesting to notice that the four most executed operations form the body of the critical loops for IDCT and windowing computation, consisting of a fixed point multiply and an accumulate operation. A consistent part of `r0rami` high execution count is due to the operations to load the subband samples to transform, in matrix(), and the V vector elements to which apply windowing, in window(). Similarly, `r0romi` execution count is mainly due to cosine and window coefficients loading. Most of the fixed point multiplications are also performed in these two functions. For three Layer I frames with stereo channels, the `mulfx` operations required in these functions are 73728, corresponding to 93.7% of all the `mulfx` operations executed. It is also interesting to notice that the `nacl` instruction is executed only within the Window() function. Its 9216 executions correspond to 16×8 operations per call, i.e. 8 sums of 16 values, necessary to reconstruct the second half of the 32 block elements, for which the symmetry $V[i] = -V[32 - i]$ holds. This consideration holds also for the `accl` instruction, which is executed $73728 - 9216 = 64512$ times. Those two instructions have been thus designed exclusively for the windowing operation. Their effect is not limited to avoid a bypass move between the result register and the operand register of the Lim FU: as explained in subsection 4.1.1, they automatically perform clipping of the accumulated value on read which would cause a significant increase of execution time, if performed by software. The execution counts of `inrj0` are very low: 1982 (0.39%) and 3304 (0.22%) for Layer I and II, respectively.

4.5 Conclusions

In this chapter we have presented the custom ASP-based decoder application used as reference implementation for our comparison study. The most interesting points discussed in this chapter are:

1. A set of features which are likely to be useful for a possibly vast set of ASP application is present in this custom TTA based ASP. Several of its instructions lack in the instruction set of a General Purpose Processor, neither they are in the current MOVE framework.

2. The audio processor architecture template implied by the machine description file is more flexible than the template of the MOVE framework. FU architecture, memory partitioning and size of the data-path are definable by template parameters.

3. The execution time spent on two routines of the subband synthesis filter algorithm, namely matrix() and window(), amounts to 88% of all execution time. This suggests that the attention should be focused on them. Analysis show also that execution time spent for the stream parsing constitute a minimal part of the total cycle count (5.8% for Layer I streams) and is therefore not worth a deep optimization.

4. Four fundamental instructions, present in the loop body of the two critical routines, are responsible for 58% of all the operations executed. No one of these instructions is present in the current MOVE framework.

Audio decoder implementation in C++ 5

This chapter presents the MPEG Audio decoder processor synthesized by means of the MOVE Framework. C++ has been chosen as source language of implementation for the application. The design of a test application for comparison requires a different approach from the standard ASP automated synthesis. In fact, instead of optimizing both software and hardware searching for the best cost to performance ratios, the selection of the hardware configuration is strictly bound to that of the reference ASP application. Only with an equivalent set of resources the quality of the generated code can be effectively compared to the manually written (and scheduled) code. This topic is discussed in detail in the first section, that describes the hardware configuration of the decoder, the choice of the special function units and its motivation. The second section describes the software design of the decoder. General structure of the program, execution flow, and main data structures of the decoder are presented. The motivations for the algorithms chosen for the implementation are also discussed. The last section presents the simulation results obtained. This section is organized in two parts. Firstly the code size analysis of the program and the algorithm requirements, indicated by the sequential code simulation, are presented. Then the results of the scheduled code simulation are discussed in detail. These include execution counts, TTA's specific optimization results, procedure execution times, software pipelining ratio and most frequently executed operation counts. A summary of the MOVE decoder characteristics and the main information obtained from the code simulation concludes this chapter.

5.1 The hardware configuration

The choice of the processor configuration was driven by the necessity to have an architecture comparable to the one designed at SPaSE. Two main criteria were considered. The first criterion was to make the two hardware configurations as similar as possible, to allow a one-to-one comparison of resource usage. The second criterion was to avoid as much as possible user-defined

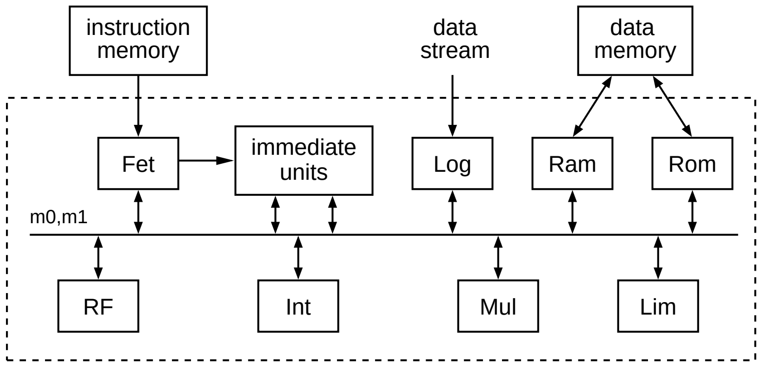


Figure 5.1: The templated hardware configuration of the MPEG Audio decoder obtained by means of the MOVE Framework. Note that the implementation details regarding the input data stream of the Log FU are not defined in the software trajectory.

FU	Instructions
Mul	`mul`, `mulfx`
Int	`add`, `sub`, `eq`, `gt`, `gtu`
Lim	`add`, `sub`
Log	`and`, `ior`, `xor`, `shl`, `shr`, `sxbh`, `sxbw`, `sxhw`, `inrj0`
Rom	`ld`, `ldb`, `ldh`
Ram	`ld`, `ldb`, `ldh`, `st`, `stb`, `sth`
Fet	`jump`, `call`, `trap`

Table 5.1: Instruction set of the MOVE decoder processor.

instruction, because we wanted to test the capabilities of the standard MOVE template. User-defined instruction are custom extensions of such template and affect the comparison results.

One of the most evident problems found when comparing the two ASPs, was the rather different hardware configuration due to the limitations of the architecture template of the MOVE Framework. The most significant are: single data width of 32 bits fully supported, single memory bank, and the limitations of the FU template. As described in chapter 6, SPaSE's architecture also contains several special functionalities implemented in hardware, which are not present in the current MOVE Framework and strongly influence code performance.

5.1.1 TTA template configuration

For all of the custom processor characteristics that can be expressed by some parameters of the MOVE template, like the number of buses or the number and the instruction set of the function units, the corresponding values were assigned to the MOVE processor configuration. By this way, we intended to test the scheduling process with a set of resources as close as possible to the set of the SP8881 audio processor.

Figure 5.1 shows a schematic view of the decoder hardware configuration. The processor consists of 6 standard function units, fully connected by two move busses, plus a fetch unit Fet, and two signed immediate units. An integer register file of 32 registers and one boolean register are also present. The Ram and Rom FUs are connected with a unique memory bank. Table 5.1 shows the instruction set of the 6 FUs. The functionality of the 6 units is as follows:

Mul has the same instruction set as the SP8881 mul FU, consisting of a special instruction for 20 bit fixed point multiplication, mulfx, and 32 bit integer multiplication mul.

Int groups the standard instruction sets of the MOVE Framework for addition, subtraction and comparison.

Lim provides only add and sub operations. This function unit is meant to take the place of the specialized Lim function unit of the SP8881, but it has no special features.

Log groups the standard instruction sets of the MOVE Framework for logic operations, shifting operations and sign extensions. In addition, the read-and-justify special instruction `inpj` is also present.

Rom is a standard load unit of the MOVE template.

Ram is a standard load-store unit of the MOVE template.

The choice of the FU latching discipline was not obvious, because neither hybrid pipelining nor (Semi) Virtual Time pipelining, the options available in the MOVE architecture template, match the behavior of True Virtual Time latching of SP8881 audio processor FUs, described in section 4.1. The VTLP latching discipline was chosen, as it shares more similarities.

One of the essential factors that affect the performance of the scheduled code is the FU latency. The MOVE Framework allows to specify one operation latency value per each function unit. All the instructions supported by a FU share this latency value. In the SPaSE architecture, different latencies are possible for instructions of the same function unit[1] For the load/store units a latency of two was chosen. For the log unit, containing both single cycle latency and two cycles latency operations, the latter was chosen, as a worst case. This implies that elementary boolean operations `and`, `ior`, `xor` have a latency of two cycles. The alternative, that is to include these operations in another FU, would have affected in a positive way the scheduling result, giving extra scheduling freedom not present in the custom decoder. The worst case solution was therefore chosen, also to point out this remarkable limitation of the current MOVE template.

5.1.2 Specializing the Log and Mul function units

When deciding the hardware configuration of the decoder, a crucial choice is to establish which of the special features supported by hardware in SPaSE audio processor were to be included in the MOVE configuration as user defined special FUs, and which ones were to be considered too specialized for a realistic and efficient templated synthesis tool and to be left out.

At first sight, to gain a fair comparison, no special features should be included in the standard framework, but it is unrealistic not to implement very critical and profitable special functions, as this is part of the standard design trajectory in the MOVE Framework. Only two special instructions of SPaSE's processor have been included in the MOVE decoder as user-defined instructions: the fixed point, multiply instruction and a limited version of the read and right-justify instruction `inrj0`, called `inpj`.

A fixed point multiply instruction appeared compulsory in order to avoid an expensive, unnecessary floating point hardware and the additional overhead caused by the conversion operations between float and integer data words. Its cost is slightly higher than for a standard

[1]The latencies of single operations can not be specified in the machine description file, but are implied by the simulation algorithms in the module mach.mc.c, described in section 4.2.

multiplier. A test showed that the cycle count increases by 40% using a function call and floating point arithmetics to simulate the fixed point operation. Another alternative, less expensive from the hardware point of view, is to implement the fixed point multiplication as a software function using integers. Since this operation is present in every critical loop of the program, the inlining or the calling of such a function, though small and simple, causes a cycle count increase even higher than the first alternative. The increased resource requirements would have made the comparison task more difficult and the results less significant. From another point of view, fixed point multiplication is a 'specialized' operation implied by the MPEG specifications, which state also its minimum bit precision. This makes it an obvious candidate for a user defined instruction.

The bit stream read-and-justify instruction was also implemented as a user defined instruction. In the actual hardware configuration of SPaSE's decoder, the input bit stream is implemented as a FIFO buffer of 1024 words located in a dedicated RAM bank. The read-and-justify instruction does not appear within any critical loop executed for each block of samples, therefore the program performance is not the chief reason for implementing it as a special function. Nevertheless, such an instruction is a valid general approach for those applications that need to parse of bit fields of variable size. First of all, it allows to simulate a bit stream using a file for its data contents, while hiding its management behind the user-defined interface of the instruction. The sole, highly inefficient alternative is to define a constant memory area representing the bit stream contents, and to link it to the program. The second reason is that in this way we avoid the explicit management of RAM buffer, that requires shifting and masking operations for bit field extraction and right justification. Such operations are efficiently scaled and implemented in hardware. Again, the performance comparison would have been more difficult and the resource requirements would have been affected.

Many other specific features of the audio processor for possible user-defined instructions were considered and discarded. The first reason for not implementing them is that it was considered interesting to test the effect of their absence on performances; secondly, they are too specific to be efficiently implemented in the user-defined FU template. For example, a consistent part of the functionalities of read-and justify instruction can not be implemented in the MOVE TTA version, because the template of MOVE FUs, based on a trigger input move, does not allow result moves to trigger another operation. The remaining part of `inpj` functionality was implemented in software within a class method.

5.2 Software implementation

In this section the decoder C++ sources are described. It is possible to compile conditionally the decoder for the TTA MOVE Framework or for a standard workstation architecture. The latter compilation mode has been used for a quick simulation and test of decoding results in the first stage of the design. The first part of this section gives a general description of the application and motivates the design choices that were taken for the implementation. The second part goes into the implementation details such as the main data structures and the general flow chart.

5.2.1 The implementation approach

When approaching the decoder design, two main freedom degrees were available: the language of implementation within the MOVE Framework and the choice of the decoding algorithms. Three high level languages are currently available to design applications: Fortran, C and C++.

The former is currently available as a preprocessor that translates it into C code. Therefore, C and C++ were the two sensible choices to be considered for a better compiled code. Many C applications have already been developed or simply recompiled for the MOVE Framework, both to test the back-end and to address practical applications. For this reason, the compilation results are already well known, as well as the strong and weak points of the gcc front-end.

On the other hand, it was interesting to acquire a more detailed idea of what is the code quality of C++ compiled programs. One of the purposes of this choice was therefore to verify the quality of a C++ compiled code, to establish if it may cause overhead with respect to an equivalent, yet less elegant, C implementation. Classes were used to encapsulate sets of global data, to give them a coherent and safe interface and to hide the implementation details of functions depending on the compilation target machine.

After the analysis of the compiled code, it seems evident that classes do introduce significant overhead in terms of memory accesses, address computations and function calls. Probably most of the features of C++ classes, ranging from automatic constructors and destructors to virtual members and class inheritance, are very unlikely to be needed in computation-intensive programs as the ones usually compiled by the MOVE Framework. What C++ and classes could eventually buy for the designer is data hiding and better structured and readable programs. This is probably not worth the above mentioned overhead.

The MPEG Audio decoding consists of rather complex algorithms, from the computational point of view. Different implementations are possible for them, usually trading algorithm complexity for efficiency. On the one hand, it is desirable to keep the algorithms as similar as possible to the specification pseudo-code, after introducing some obvious optimizations. In fact, those algorithms are simple, readable and they are the most intuitive way to perform the actions described in the specifications, therefore they require less design time. In principle, the compiler should be able to adapt the source code and to restructure loops in an optimal way. On the other hand, it is convenient to utilize optimized algorithms as similar as possible to the code of the SPaSE solution. In fact, this approach gives a more similar code and therefore a more reliable comparison, but results in a complex, sometimes obscure code, which is far from a direct implementation of the specifics presented in section 3.3.

The MPEG decoder algorithm contains several loops characterized by heavy computations. Usually the second approach has been taken for these critical loops and it proved to produce a more efficient MOVE code. This approach did not work for one of the most critical loops, the Inverse DCT computation. The algorithm chosen by SPaSE is strongly optimized for their specific hardware configuration and a direct translation into C++ gives a less efficient MOVE code. This can be seen as an indication of how different the two hardware configurations are and how it is difficult to perform a reliable comparison of their performances.

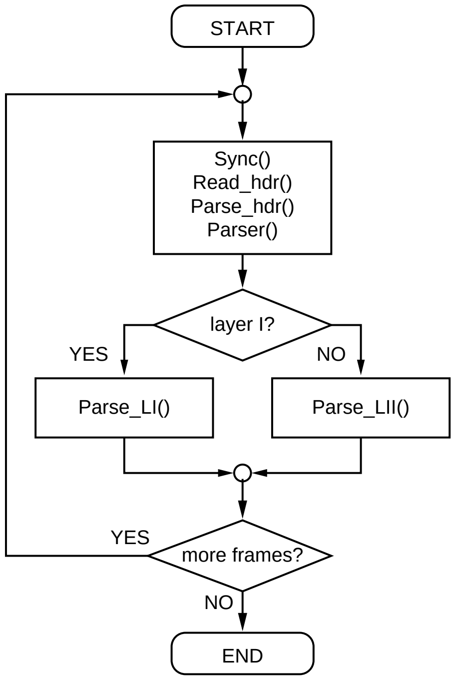


Figure 5.2: Flow-chart of the main module of MPEG Audio Decoder.

5.2.2 Decoder flow-charts

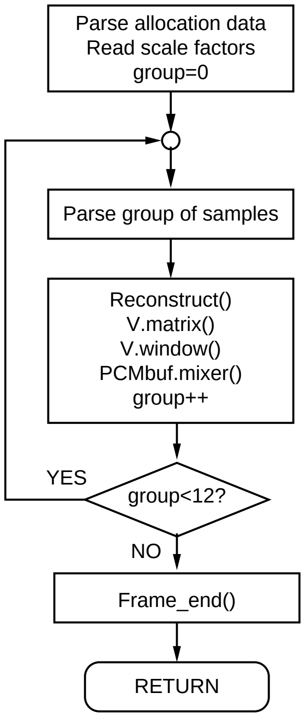
Figure 5.2 depicts the flow-chart of the main module, consisting of a loop executed for each frame. This loop calls the functions which synchronize the decoder to the next frame, parse its header and establish the frame layer. Then allocation data is parsed and, depending on the layer, a different function is called to parse the rest of the frame data and perform decoding.

The organization of the decoder into routines and their functionalities exactly replicates the structure of SPaSE code for SP8881. This choice was taken to allow a function-by-function analysis and comparison.

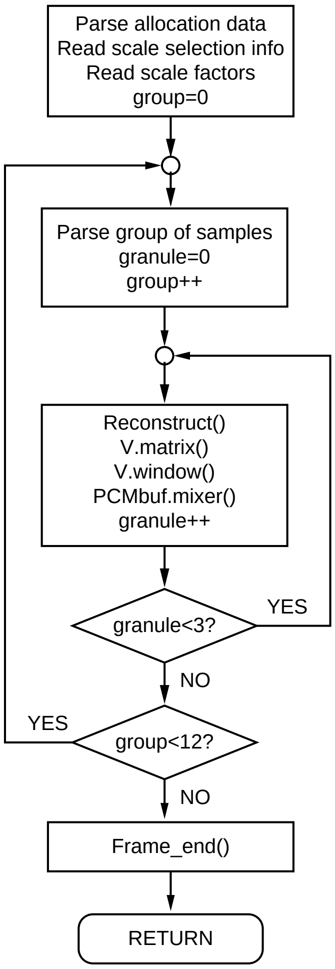
The most critical parts of the program are contained within the two decoding functions for layer I and II, whose flow-chart is depicted in figure 5.3. These functions first perform the parsing of scale factor data and, for each of the 12 groups of samples, the parsing and subband filter synthesis. All the algorithms of the critical core of the decoder reproduce the optimizations applied in the Mas version, except for the loops in Matrix() computing the IDCT.

5.2.3 Class descriptions

Three classes have been fully implemented: MPEGheader, SBfilter, Bit stream. Two more classes have been defined and partially designed for possible extensions of the program functionalities: CRCunit, to manage the parity check and DualBuffer, to enable the usage of a shared output buffer. These classes have currently minimal functionality and will not be treated in further detail.



(a) Flow chart Parse_LI()



(b) Flow chart Parse_LII()

Figure 5.3: Flow charts of Parse_LI() and Parse_LII(). Both functions are called once per frame. Their inner loop is the critical core of the program. It comprises Reconstruct(), V.matrix() and V.window(). The latter two correspond to Matrix() and Window() functions in the Mas decoder. The critical core is executed 12 times per channel for Layer I frames and 36 times for Layer II.

The class SBfilter

The class SBfilter implements the IDCT computation and windowing algorithms of the subband synthesis filter, that represents its critical parts. Below the class declaration is shown:

```
class SBfilter {
    word_i  Vlo_R[NOVVE]
    word_i  Vhi_R[NOVVE]
    word_i  Vlo_L[NOVVE]
    word_i  Vhi_L[NOVVE]
    int     pV
    int N(int i, int k)
 public:
    void matrix(word_i *samples, word_i chan)
    void window(word_i *samples, word_i chan)
    void initialize(void)
    void newblock(void)
}
```

The data members consist of 4 vectors Vlo_R, Vhi_R, Vlo_L, Vhi_L, two per channel, corresponding to the V vector of the subband synthesis filter described in section 3.3.3. Section 4.3.3 describes how they are computed from V. The pointer pV holds the address of the latest block of 64 samples of the vectors which have been computed. The description of the function members is shown below:

int N(int i, int k) returns the cosine coefficient used in IDCT computation corresponding to V vector element $V[i]$ and subband sample k of the current block.

void initialize(void) resets the elements of the vectors and the pointer pV.

void newblock (void) updates the value of the pV pointer to the next group of 64 elements.

void matrix(word_i *samples, word_i chan) computes the IDCT on the group of 32 subband samples pointed by samples, and stores the results in the position of the filter vectors pointed by pV.

void window(word_i *samples, word_i chan) performs in a single step, sample by sample, the operations described in section 3.3.3. The reduction of the subband filter vector into a half sized vector is simply implemented as the selection of its 16 elements needed to reconstruct the time sample considered. These elements are multiplied by the corresponding coefficient of the windowing vector and then they are summed. The result is the output time sample.

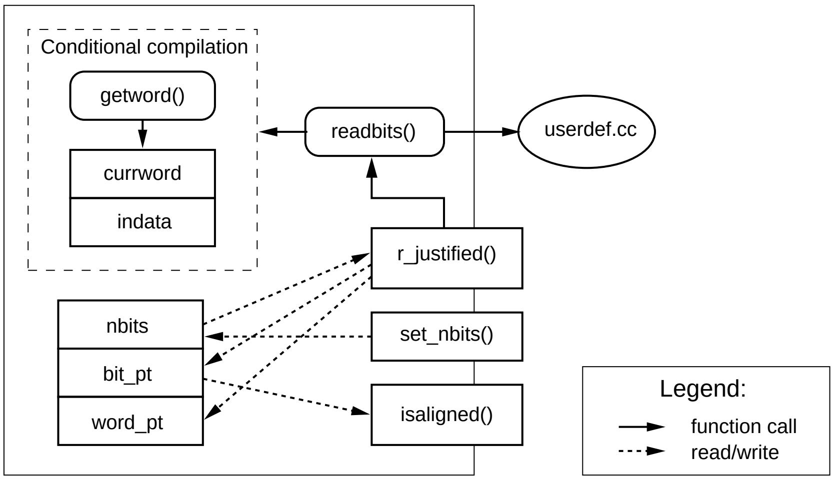


Figure 5.4: Structure of the class Bitstream.

The class Bitstream

The purpose of the Bitstream class is to simulate the input buffer used for the MPEG stream by means of a C++ file stream. This class provides the interface and the complete behavior of the read-and-justify instruction implemented in the SP8881. At the same time, the Bitstream class hides the interface of the user defined, limited version of the instruction implemented within the MOVE Framework. Figure 5.4 shows the structure of this class and the relations between its methods. In order to simulate both in MOVE TTA and the workstation architecture code, parts of this class are conditionally compiled.

```
class Bitstream {
#ifndef TTA_COMPILE
    ifstream indata
    word    currword
    word    getword(void)
#endif
    int     word_pt
    char    bit_pt
    char    nbits
    inline int  ReadStreamBits(int nb)
  public:
#ifndef TTA_COMPILE
    inline void open(char * filename)
    ~Bitstream()
#else
    inline void open(void)
#endif
    void set_nbits(int n)
    int  r_justified(void)
    int  is_byte_aligned(void)
 };
```

The data members of this class contain the state of the input buffer. Their descriptions are listed below:

word_pt contains the number of words currently read from the input buffer.

bit_pt contains the position of the next bit to be read in the current word.

nbits contains the number of bits to be read from the bit stream.

currword contains the value of the last word read from the bit stream in the non-TTA compilation mode.

indata is the input file stream that contains the data of the MPEG bit stream in normal compilation mode.

The function members allow to control the input stream parsing. Their descriptions are listed below.

void set_nbits(int n) sets the number of bits to be read by the next read-and-justify instructions.

int r_justified(void) reads from the bit stream a number of bits specified by the nbits data member and updates the word counter and the bit pointer. This method simulates the exact behavior of the read-and-justify result move of the SP8881. This allows a direct comparison of the overhead caused by the lack of an extended version of the instruction.

int is_byte_aligned(void) returns TRUE (1) if the current bit position in the bit stream is multiple of 8.

void open(char * filename) opens the binary file that contains the data used to simulate the bit stream in normal compilation mode, reads the first word from the stream and sets correspondingly the data members.

int ReadStreamBits(int nb) is the C++ counterpart of the actual read-and-justify instruction of the MOVE Framework, implemented as a user defined instruction.

The class MPEGheader

The MPEGheader class holds the information carried by the currently parsed frame header. For each field of the header only two operations are possible: reading of the current value and setting a new value. The implementation is thus straightforward. 8 data members correspond to the fields of the frame header depicted in figure 3.7. The latter contains the CRC check protection word.

```
char dlayer            char padding
char dmode             char demphasis
char dmode_ext         char dfrequency
char protection        word dCRC
char dbitrate
```

As explained in subsection 6.4.2, the access of data members of a global class causes inefficient code, which can be avoided by using global built-in types instead.

5.3 Simulation and analysis

In this section the MOVE decoder analysis and simulation are presented. Two MPEG Audio streams, one for each layer, have been used as test data set for these simulations. They are referred to as Layer I test stream and Layer II test stream, and they are the same streams used for simulation in section 4.4. Both streams encode a stereo signal sampled at 44.1 kHz, without

CRC protection or pre-emphasis. See section 3.3 for a complete specification of Layer I and Layer II encoding parameters.

A description of the changes introduced to the parallel simulator begins this section. These changes were needed to obtain a complete analysis data set, comparable with the set produced by simulating the Mas decoder. In the second part of this section are presented the general characteristics of the application and its resource requirements, collected from the simulated execution of the sequential code. Then the code size analysis of the scheduled code is discussed, with particular detail on the effect of the loop unrolling parameter. Finally, the simulation results of the scheduled code are presented for both scheduled versions. Results comprise execution cycle counts sorted by functions and operation counts.

5.3.1 The simulation approach

Before starting the comparison between MOVE and Mas decoder versions, some extensions were needed to the simulation results presented by the parallel compiler of the MOVE Framework. In fact, an important piece of information for the comparison is the execution count and percentage contributed by each instruction, as it gives an indication of what are the operations causing more overhead. Therefore, the execution count of each operation in the scheduled code has been added. Each trigger move has been counted as an operation. The jump, call and trap moves are also counted as single operations. These counts, surprisingly, were not present in neither simulations, though they are important to estimate the relevance of call overhead and of jumps.

The code size computed after scheduling does not take into account the instruction words that should be occupied by long immediates. In order to have an idea of what could be the size increase due to immediates, the count of immediate fields present in the program has been added. Similarly, to have a rough estimation of what is the effect of immediate field fetching on code speed, it has been added the execution count of immediate moves. Both counts are ordered by the minimum bit width required for the immediate values.

5.3.2 Static analysis

Table 5.2 presents the size of the decoder functions obtained with inter basic block and software pipelining scheduling scope and without loop unrolling and peeling. Profiling information, obtained by simulating Layer I or Layer II decoding algorithms, has a negligible effect on the scheduled code size (+8 instruction words in scheduled code profiled with the Layer II test stream). However, the scheduler does not schedule non-profiled functions in a profiled program, therefore both streams are necessary to simulate and profile non-common routines, namely Parse_LI(), Parse_LII() and Degroup().

The unrolling parameter greatly affects the code sizes of the two critical routines Window() and Matrix(), but it does not influence the rest of the program. Execution speed also increases, but by a smaller factor. When these factors are reported to the global program, the size increase is less relevant (in percentage), while the speedup gain remains stable at the values obtained for the transformed functions. This is due to the fact that, as shown in table 5.5, Matrix() and Window() are counting for the 80% of execution time. The unrolling factor has a range of optimal values. Some tests were performed on the Window() function to evaluate the effect of the unrolling aggressiveness parameter. These tests, performed using inter basic block scheduling scope and the Layer I stream, show that the performance to code size ratio rapidly decreases

Function name	**Description**	**Code size**
Matrix	IDCT computation on subband samples	211
Window	Time sample reconstruction	199
Parse_LI	Parsing layer I scale and samples	193
Parse_LII	Parsing layer II scale and samples	594
Parser	Parsing of allocation data	167
Deemphasis	Time sample deemphasis and rescaling	110
Reconstruct	Re-quantization and rescaling	101
main	Main frame loop, fixed to 3 frames	67
Read_hdr	Reading of frame header	50
Mixer	Time sample writing to output buffer	46
Degroup	Degrouping of layer II granules	39
Short_Sync	Audio frame synchronization	34
initialize	Initialization of V vector to zero	33
Parse_hdr	Parsing of the frame header fields	23
Frame_end	Discard ancillary and padding bits	23
r_justified	Reads variable bit field from stream	20
Reset	Global data initialization	14
total		1924

Table 5.2: Number of instructions of decoder functions. These figures are obtained by scheduling sequential MOVE code profiled by Layer I stream decoding simulation, except for Parse_LII() and Degroup(), which are specific functions of the Layer II decoding algorithm. For these function the Layer II test stream has been decoded to obtain profiling information.

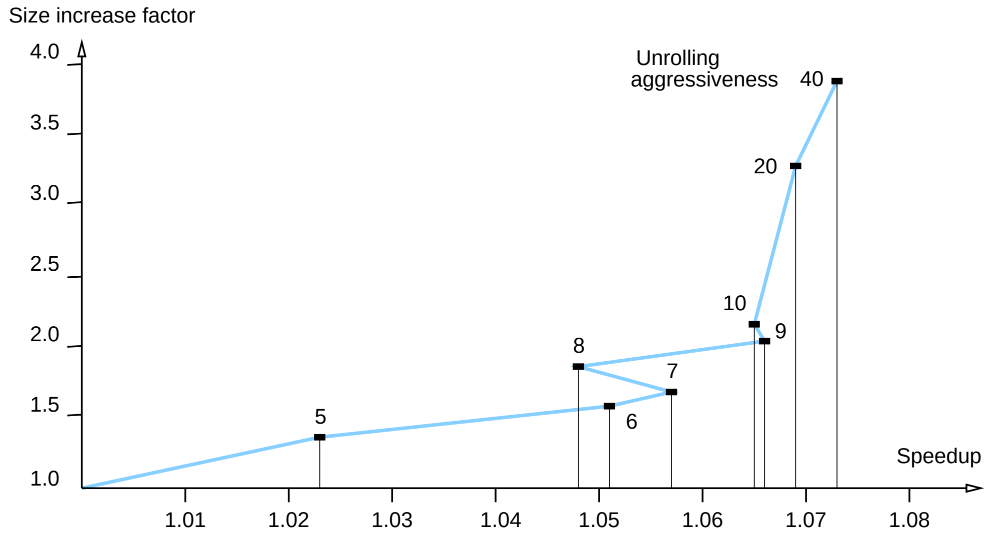


Figure 5.5: Effect of loop unrolling/peeling aggressiveness on code size and cycle count of Window(). The horizontal axis shows the speedup with respect to the version without loop unrolling. Vertically, the code size ratio, compared to the version without loop unrolling, is shown.

with the increase of unrolling aggressiveness factor, making it less and less effective. See figure 5.5.

The same sequential code has been scheduled with unrolling, to test its effect on code size. By specifying a loop unrolling aggressiveness value 10 the code size of Matrix() raises to 524 words (+148%) and Window() size rises to 418 words (+110%). The total size of the program rises to 2456 words (+27.7%). Also for Layer II only Matrix() and Window() are affected by loop unrolling. The sizes are 524 (+148%) and 419 (+108%) words, respectively. The total size of the program rises to 2465 words (+27.6%).

5.3.3 Sequential code simulation

The preliminary simulation of sequential MOVE code gives (1) profiling information based on execution counts for better scheduling and (2) application characteristics useful to estimate the resource requirements. The latter consist of average operations per basic block, average number of live GPRs and operation counts. The average number of live GPRs gives an indication of the register file size required to avoid register spilling. Table 5.3 shows the simulation results of the sequential MOVE code. The operation counts have been grouped into classes of instructions with similar functionalities. It is interesting that the load and store operation class is responsible for the largest contribution to the execution time. This suggests that a memory access optimization is fundamental to improve the application performance. The percentage of shift and sign extension operations is unexpectedly high for the MPEG decoding algorithm as described in sections 3.3 and 4.3, especially when compared to the percentage of multiply operations, which are fundamental for the above mentioned algorithm. Part of these operations is due to inefficient MOVE code generation.

Characteristic	Values	
	Layer I	Layer II
Executed operations/frame	406405	1178749
Average operations per basic block	6.03	6.15
Percentage of load/store operations	17.1%	16.7%
Percentage of shift/sign operations	7.2%	7.1%
Percentage of compare operations	8.7%	8.6%
Percentage of logic operations	3.5%	3.6%
Percentage of multiply operations	6.7%	6.9%
Parallelism upper bounds	430	2644
Average number of live GPRs	19.752	21.032

Table 5.3: Characteristics of the decoding algorithm and results of sequential simulation. The first column is obtained using the Layer I test stream as input data, the second one using the Layer II.

	Layer I		Layer II	
	No unrolling	Unrolling 10	No unrolling	Unrolling 10
Moves	2594309	2470889	7585450	7226352
Cycles	1402738	1306614	4112760	3816751
GPR reads	983044	901724	2875537	2633630
GPR writes	564807	483363	1649551	1399539
Operations	1182059	1172267	3445582	3409027
Bypass ratio	0.273	0.326	0.272	0.326
Dead result ratio	0.367	0.441	0.366	0.443
Common operand ratio	0.034	0.058	0.034	0.051
SW pipelining ratio	0.020	0.021	0.019	0.020

Table 5.4: Execution results for the scheduled code. The second and the third columns come from the profiling information obtained decoding a Layer I test stream, the fourth and the fifth ones refer to the code profiled with a Layer II test stream. The third and the fifth columns are obtained specifying an unrolling aggressiveness value equal to 10.

5.3.4 Scheduled code simulation results

Many scheduling parameters are available for adjusting the scheduling process. The chosen scheduling scope was inter basic block with software pipelining, which is the largest scope available. For the other parameters, the default values have been maintained.

Table 5.4 shows the results of the MOVE decoder simulation. The decoder has been tested with three audio frames of Layer I and Layer II test streams. Note that the Layer II audio frames contain three more times samples. Therefore the critical functions implementing the subband synthesis filter are executed three times more for Layer II.

The second and the fourth columns of the table show the results of the same simulation when loop unrolling is applied during scheduling. The unrolling aggressiveness parameter allows to trade code size for execution speed. For the performance comparison it is thus necessary to take always into account that with an automated framework such as MOVE it is possible to have a range of solutions.

Procedure	Layer I			Layer II		
	Cycle count	**Percentage**	**Calls**	**Cycle count**	**Percentage**	**Calls**
window	577440	41.15	72	1550556	40.58	216
matrix	549612	39.16	72	1540252	40.31	216
Reconstruct	115574	8.24	36	342648	8.97	108
Parse_LI	72354	5.16	3	–	3.68	0
mixer	46800	3.33	36	140648	2.25	108
Deemphasis	28620	2.04	36	85860	2.22	108
Parse_LII	–	0.00	0	84987	1.25	3
r_justified	4974	0.35	350	47808	0.54	3292
Degroup	–	0.00	0	20592	0.10	528
Parser	3732	0.27	3	3783	0.09	3
initialize	3542	0.25	1	3542	0.00	1
Frame_end	252	0.02	3	191	0.00	3
main	147	0.01	1	147	0.00	1
Short_Sync	116	0.01	3	136	0.00	3
Read_hdr	114	0.01	3	114	0.00	3
Parse_hdr	69	0.00	3	69	0.00	3
Reset	14	0.00	1	14	0.00	1

Table 5.5: Execution time spent in each decoder function.

For Layer I version loop unrolling reduces execution time by -6.8% For Layer II unrolling performs slightly better: execution time is reduced by -7.1%, while the code size increase remains the same. This is due to the fact that the call counts of unrolled functions is three times higher per each frame.

Table 5.5 shows the execution time spent in each function of the MOVE decoder. It is worth to notice that more than 80% of the cycle count is due to Window() and Matrix(). Including Reconstruct() the three functions, representing the subband synthesis filter algorithm, take more than 88% of cycle time. This datum shows also that parsing is not a critical part of the program. The execution time spent on r_justified() represents in fact only 0.35% of execution time for Layer I and 1.16% for Layer II. The routines that parse the input stream take only 5.8% and 3.3% of the total cycle count, respectively.

The average number of moves per instruction word is also computed during analysis of function executions (not shown in this table). The average value is 1.77. Not counting the immediate moves, this value gives and indication of what bus load should be expected. These values correspond to 88% of bus utilization.

Table 5.6 shows the count of the 16 most frequently executed operations for Layer I and Layer II simulations. Note that the Lim function unit is responsible for more than 78% of the executed add operations. Since it is the most specialized, its declaration precedes the Int FU declaration in the machine description file. The same holds for the Rom FU, that is assigned to 98% of load instructions, even though most of the memory accesses are coupled and independent, and therefore they could be performed using the Ram and the Rom units in parallel. This would save a GPR load and a GPR write per each couple of memory accesses and would have a significant impact on performance. A detailed analysis of this problem is presented in 6.1. It is also interesting to notice that the `inpj` operations represent only a minimal part of the

Layer I		Layer II	
Operation	**Count**	**Operation**	**Count**
lim.add	325737	lim.add	950151
rom.ld	177805	rom.ld	527546
int.gt	99436	int.gt	288664
int.add	87741	int.add	263825
log.shl	82731	mul.mulfx	242976
mul.mulfx	81072	log.shl	240653
log.and	40675	log.and	122040
mul.mul	36900	mul.mul	111756
lim.sub	22455	lim.sub	63390
ram.st	21289	ram.st	60652
rom.ldh	7650	int.eq	18108
int.eq	7075	rom.ldh	16491
rom.ldb	4297	ram.ld	10372
ram.stb	3567	rom.ldb	7909
ram.ldh	2379	ram.stb	4651
log.inpj	1934	log.xor	3888
jump	23927	jump	70738
call	622	call	4596
total	1182059	total	3445582

Table 5.6: Most executed operations.

execution count (less than 0.2%). These results confirm that the parsing routines do not represent a critical part of the program. Therefore, the partial software implementation of the Mas read-and-justified instruction does not significantly affect the decoder execution time.

5.4 Conclusions

In this chapter we have presented the MOVE decoder and the simulation results. Some significant points come out from the analysis of these results:

1. Most of the execution time is due to Matrix() and Window(), the main routines forming the subband synthesis filter algorithm. These two routines are therefore the part of the program to be analyze in detail. The execution time spent for stream parsing is a negligible percentage of the total cycle count.

2. Table 5.6 gives some hint of inefficiencies: for example, the high number of comparison operations (gt) necessary to check the exit condition of critical loops and of the shift left operations, are not directly deducible from the algorithms of MPEG specifications. The causes of such high counts are discussed in chapter 6.

3. Table 5.6 also reveals that FU assignment is a relevant cause of overhead and generates unnecessary register usage, as explained in subsection 6.1.4.

Comparison of MOVE and Mas decoders 6

To enhance the performance of an ASP generated by an automated framework such as MOVE, the comparison of a custom design and what can be achieved by means of the framework for the same application is a valuable way to understand which are the causes of inefficiency and the most restricting limits.

This chapter presents the results of the comparison between the custom designed decoder developed at SPaSE and the MOVE decoder. The comparison is articulated in three parts. In section 6.1 the main differences between the two architectures are analyzed, their effect on code size, cycle count and resource utilization of the ASP application are discussed. The possible approaches to implement them in the MOVE framework too are discussed. Section 6.2 compares the execution statistics of the two decoders. This section is introduced by a description of the limits of this simulation and their extent. The comparison of the code size and the execution times for each function are then presented. The comparison of the most frequently executed operation concludes this section and gives a valuable indication of what are the major sources of overhead in the MOVE decoder code. In section 6.3 the code and the performance of the two critical functions, Matrix() and Window(), are compared. Their most time-consuming loops are analyzed instruction by instruction. This gives the opportunity to estimate the effect of each instruction not present in the standard MOVE instruction set. In section 6.4 a list of recommendations for possible extensions of the move framework, each related to a specific problem found in the MOVE code, is discussed. For each proposal it has been discussed what are the expected effects on scheduled code and which are the requirements needed to implement them.

6.1 Architecture differences

The TTA configuration of the SP8881 audio module is not achievable by instantiating any set of parameter of the MOVE template. In fact, several fundamental characteristics of its templated architecture, which can have a remarkable impact on processor performance, cannot be freely defined by the user in the current MOVE framework. These characteristics are listed below and discussed in the following subsections:

1. Size of data-path.
2. Memory addressing and address space size.
3. Memory partitioning.
4. Temporary data: register file versus memory.
5. Function unit template and True Virtual Time latching discipline.

6.1.1 Size of data-path

The width of the data word is typically a design choice for custom ASPs and it is related to some application requirements such as bit precision, speed. One of the biggest limitations of the current MOVE framework template is that only a single, fixed size of the integer data-path is fully supported for all the building blocks of the architecture: FUs, Register files, and buses. Chapter 8 describes possible approaches to overcome this limitation and make it possible to specify the data path size of each component of the architecture independently. In particular, a subproblem is studied in detail: how to provide a formalism to map operations on FUs with multiple sizes smaller than 32 bits.

6.1.2 Memory partition

ASPs often partition their data and memory space into multiple memory banks. This approach alleviates one of the most remarkable bottlenecks of processors, i.e. memory access. The main benefits of memory partitioning into smaller banks are:

1. Each memory bank requires less read/write ports and can be smaller. This allows faster or cheaper memory banks.

2. Since accesses to different memory banks are always independent, the task of the memory disambiguator is alleviated. Due to limitations of real disambiguators, less write backs to memory are needed, and more parallel read/write accesses are possible.

Partitioning memory in multiple banks requires the introduction of a storage class qualifier for data, so that the software designer or the compiler are able to specify the data partitioning in the source code. This storage qualifier should pass information to the back-end through annotation.

6.1.3 Memory Addressing and address space

The memory addresses of MOVE code generated by the current framework are fixed to 32 bits. The memory requirements of virtually any ASPs, especially for embedded systems, are far from the limit of 4 GigaBytes allowed by 32 bit addresses. Along with the memory partitioning, the possibility to specify smaller address spaces, not present in the current MOVE framework, could limit the number of bits required to specify memory addresses. For example, an ASP requiring 16k of memory (14 bit addresses) could be partitioned into four banks of 4k, each requiring only 12 bit addresses.

Though the code generated by the current MOVE framework is for 32 bit architectures, memory is accessed byte-wise. This implies that adjacent data words are accessed by multiples of four, causing overhead when computing memory addresses. For example, the loops containing accesses to an array by means of a function of the loop index, either produce shift left by two (multiply by four) operations for each address computation, or cause an induction variable to be maintained for the memory address and update it correspondingly to the array index, but times four. Such a problem can be alleviated by explicit use of pointers instead of array indexes, but it gives an unreadable, difficult to maintain source code.

The mirroring loop of the Matrix() function shows an example of both problems. It was first implemented by the following code:

```
for ( int i=0, k=0; k<16; i+=st, k++)
{
        t0 = s[i]; t1 = s[31*st-i];
        M[k]    = t0 + t1;
        M[k+16] = t0 - t1;
}
```

This code produces a loop body that requires 23 cycles to be executed. Two shift left operations are present, to compute at each iteration the effective address of the two samples `s[i]` and `s[31*st-i]`. Besides two induction variables for the addresses of `M[k]` and `M[16+k]`, one more variable, although not necessary, is maintained for the actual value of the index loop `k`, used to check the exit condition.

The following piece of code presents the re-designed C++ version of the mirroring loop, using pointers. In this case one of the address values and the loop index are mapped into the same register. The effect on the MOVE code produced by the compiler is dramatic: only 16 cycles are required to execute a loop iteration; a saving of 9 moves and 3 registers is achieved. The total execution time spent in Matrix() is reduced by 7.3%. This relatively meager improvement is caused by the fact that the mirroring loop contributes only minimally to the cycle count of Matrix(), as shown in section 6.3.

```
s0 = s; s1 = s+31*st;
for ( word_i * m=M; m<M+16; m++ )
{
        t0 = *s0; t1 = *s1;
        *m       = t0 + t1;
        *(m+16) = t0 - t1;
        s0 += st; s1 -= st;
}
```

6.1.4 Register file

The TTA processor developed at SPaSE does not contain register files: every move source and destination is a FU register. The motivation for this choice is that register files are expensive, they take a large part of the chip area and of the power consumption. This design choice may be sensible for ASPs in which area is a critical factor and whose application does not need many frequently accessed temporary variables. Perhaps the most important reason for avoiding register files is related to the instruction bit coding. Typically, a large part of the source and destination ID space is taken by the register file addresses. A longer instruction word is thus required, or the maximum size of source immediates must be reduced.

The elimination of register files is likely to raise new problems. The decoder simulation results reveal that the amount of moves involving a GPR as source or as destination is a consistent percentage of all the moves executed: more than 58%. Eliminating register files would cause register read and write moves to be substituted by a couple of load and store moves, and will introduce several cycles of latency. This has an unacceptable effect both on performance and code size.

Apparently, the peculiar features of TTAs, result bypassing, dead result elimination and operand sharing, alleviate this problem but they are not sufficient to allow any efficient configurations without register files. To minimize the memory access, a general approach that has been taken for the design of SP8881 audio module is to embed in specialized internal FU registers many long-living, frequently accessed variables. Auto-incremented memory pointers, hardware word counter and hardware loop index are examples of such 'special purpose' registers. A natural solution to lower register requirements and make acceptable the absence of registers seems therefore to implement these and analogous features in the MOVE framework. Three other partial remedies to reduce register pressure are:

1. Static stack allocation, discussed in subsection 6.4.2.
2. Better FU assignment algorithm.
3. Introduction of identity operations.

The first fit algorithm currently in use by the scheduler always assigns an instruction to the first FU available in the FU list specified by the machine description file [10]. When a move slot is available, a move is scheduled to write the result of the FU to a GPR. By this way the scheduler makes the FU available for next operations before its current result register becomes dead. However, when other FUs are available for the next operations that could be assigned to the former FU, a better scheduling could assign them and maintain the result of the former operation for bypassing. When a RF is not present in the machine configuration, it becomes essential to detect the situations described above, to avoid expensive load and store operations. The following piece of code, taken from the IDCT loop in Matrix(), illustrates the FU assignment problem. It loads two values from memory and multiplies them. Using only the Rom FU for loading, these operations require 5 moves and use register `r3` to store the temporary value of the first memory load. The Ram unit is not utilized within this fragment of code.

```
lim.add_r -> rom.ld_t;
[...] \\ one cycle latency
rom.ld_r -> r3;
sp -> rom.ld_t;
[...] \\ one cycle latency
r3 -> mul.mulfx_o, rom.ld_r -> mul.mulfx_t;
```

By concurrently assigning the two loads to the two FUs available, 3 cycles and 4 moves are sufficient to perform the same operation.

When bypassing is not possible, or not convenient, it may be possible to avoid memory access. If a FU with a smaller latency than the load/store unit is available, an operation performing the identity can be scheduled to maintain the result value. By this way we can have the value available again after a shorter time than using memory access. The example below illustrates this technique.

```
op1_r -> or_o, 0 -> or_t;
[...]
or_r -> op2_t;
```

The result of an operation *op1* should be passed to the operation *op2* trigger register, but both FUs are necessary for other operations in between. If a FU not required is assigned to an identity operation, as a logic OR with a null operand, the result is available from its result register. If such an operation can not be scheduled, a store and a load operations are necessary:

```
op1_r -> st_t, address -> st_o;
[...]
address -> ld_t;
// load latency
ld_r -> op2_t;
```

In the worst case, the load trigger can not be scheduled within the operation block between *op1* and *op2*, and the memory access causes two delay cycles.

6.1.5 Function Unit template

In the MOVE framework the FU register set and its connections to the buses are rather rigid: a maximum of 2 operand and one result register can be present. A trigger register must also be present. A trigger move is necessary to trigger the operation.

Such a move is not necessary for some operations of the SP8881 audio module: their result move can also trigger an operation, without the need of any input move. For example, the result move reading the memory location addressed by an auto-increment pointer causes the memory address to be incremented and the automatic pre-fetch of the new location content into the result register. By this way an entire operation is efficiently mapped into a single move.

All function units of the audio processor of SP8881 have no result register, except for those accessing the memory. This design allows single cycle latencies for non-pipelined FUs and avoids the so called "lost cycle", i.e. the additional cycle needed for bypassing the result of a FU [4]. The current hardware generation module of the MOVE framework includes a FU result register, which is an actual pipeline stage. To avoid a delay cycle due to this stage, an expensive bypass circuitry must be provided for each function unit, doubling the number of connection between buses and FU output ports. Eliminating the result register increases the scheduling algorithm complexity, as operand moves can trigger a new result value and have the same scheduling constraints of trigger moves. To avoid more complex scheduling, a shadow operand register could be provided to maintain the old operand value until a new trigger move is also executed. This solution is rather expensive, as each FU operand requires two registers.

For FUs of the SPaSE processor with a latency higher than one the result move can be locked until the result generated by the last trigger move is available. This requires more complex function units, since *valid bit* is necessary to signal when the new result is ready. FU locking is of course a mandatory choice for FUs whose operations have non-deterministic latencies, but is also useful to make the manual code writing easier. FU locking is not fundamental in an automated cosynthesis framework such as MOVE, where the scheduler is in charge for making all the latencies visible.

6.2 Comparison of the two implementations

This section presents the comparison of the MOVE and the Mas decoder. The comparison in subdivided in two parts: code size comparison and execution counts comparison.

6.2.1 Comparison limits

The accuracy of the comparison results presented in this section is limited by two factors:

1. Instruction size: 24 bits in SPaSE SP8881 audio module, undefined in the current MOVE framework. Immediate field formats are also undefined.

	MOVE code		Mas	MOVE code
	No unrolling	Unrolling 10	code	increase factor
Size w/o immediates	1924	2456	1431	1.35 – 1.72
Size with immediates	2374	2903	1431	1.66 – 2.03

Table 6.1: Size comparison between Layer I MOVE and SPaSE decoder. The first row shows the sizes of scheduled code, the second row shows the optimistic estimation of the code size, based on immediate count and empty slots available in scheduled code.

2. Absence of a MOVE binary format for the scheduled code: a comparison at machine code level is thus not possible. The cycle count of the MOVE simulator does not include long immediates fetch and the move scheduler does not assign the slots taken by short immediates.

These factors are due to the lack of a final stage from parallel assembly MOVE code to instruction vectors. The mapping of long immediates in MOVE instruction words is currently a research subject. The current MOVE software subsystem assumes that one or more fixed fields of the instruction word are dedicated to long immediates. This scheme limits the number of long immediates that can be transported per instruction to the number of dedicated slots and, especially for small processors, can lead to an unacceptable waste of instruction bandwidth. The main advantage of this solution is that the scheduling task is simplified. A better solution for small TTA configurations is to dedicate a source ID[1] to indicate the presence of a long immediate and to store it in another move or instruction slot, as described in section 4.1. To obtain an estimation of the code size increase due immediates, two immediate formats were considered: a short one, taking an entire move slot, and a long format, taking an entire instruction word.

6.2.2 Code size comparison

Table 6.1 shows the code size comparison between the MOVE decoder and the Mas decoder. The size ratio between the MOVE and the Mas version was considered 'too good' to be reliable: an estimation of the code size increase due to the presence of long immediate fields was therefore performed. The second line of the table shows its results. This estimation is based on a set of immediate formats identical to the audio module of SP8881, described in section 4.1.1:

1. Source immediates, up to 4 bit long, are mapped into the source field of their own move.
2. Short immediates, 5 to 12 bit long, are mapped into one move slot which takes half instruction.
3. Long immediates, longer than 12 bits, are mapped into an entire instruction word.

The result of immediate analysis shows that 381 short immediates and 488 long immediates are needed by the Layer I version of the MOVE decoder. A first pessimistic estimation would thus indicate a size increase of 679 instructions. A more realistic approximation should take into account the number of move slots not utilized and the possibility for the scheduler to use these fields to store immediate values. As mentioned in section 5.3, about 12% of instruction words and move slots is not utilized, therefore the equivalent of 230 empty instruction words are available for short and long immediates. Considering that most of the empty instructions

[1] Instead of an ID it is also possible to dedicate a specific bit of the move to signal a long immediate. This solution allows much faster detection of immediates during instruction decoding at the expense of a waste of ID space.

Function	MOVE decoder Code size	MOVE decoder Percentage	Mas decoder Code size	Mas decoder Percentage	Code size factor
Parse_LII	594	30.9	367	25.6	1.62
Parse_LI	193	10.0	139	9.7	1.39
Window	199	10.3	122	8.5	1.63
Matrix	211	11.0	99	6.9	2.13
Deemph	110	5.7	93	6.5	1.18
Parser	167	8.7	92	6.4	1.82
Reset	14	0.7	80	5.6	0.59
initialize	33	1.7	–	–	
Frame	–	–	75	5.2	0.24
Sync	34	1.8	69	4.8	
Reconstruct	101	5.2	55	3.8	1.84
main	67	3.5	52	3.6	1.29
Mixer	46	2.4	47	3.3	0.98
Parse_hdr	23	1.2	34	2.4	0.67
Degroup	39	2.0	32	2.2	1.22
Frame_end	23	1.2	26	1.8	0.88
Read_hdr	50	2.5	20	1.4	2.50

Table 6.2: Instruction size comparison between functions of MOVE decoder and Mas decoder. The sizes of the MOVE decoder refer to the version for Layer I test and do not include any correction to take into account long immediates. The size percentage is referred to the total size of the decoder. The size factor of each function is computed as the ratio between the MOVE version and the Mas version sizes.

are exactly due to jump and call delay slots, we can optimistically assume that the majority of the empty instructions could be filled by long immediate fields. An analogous assumption for short immediates and empty move slots is less evident, but allows to establish a lower bound to the size increase, that can be therefore estimated between 450 and 680 instruction words, corresponding to +23% and +35% respectively of the program size. The same estimation is performed on a version of the decoder scheduled with a loop unrolling aggressiveness 10. It is necessary to point out that these estimations do not consider the possibility for new empty moves or empty instructions to be introduced.

Table 6.2 shows the comparison of code size between the functions of the MOVE and the Mas decoders, without taking into account the immediates. Note that r_justified() is not shown in the table, because it has no equivalent function in the custom decoder. Its functionality can be compared to the `inrj0` instruction instead. Reset() and Initialize() are grouped together and represent the Reset() function of the SPaSE decoder. The Short_sync() function of the MOVE decoder comprises the execution of Frame(), which has been inlined. It is interesting to notice that some of the routines related to stream parsing show size ratios between MOVE and Mas version smaller than 1. This is because the routines of the MOVE framework take advantage of the register file presence, while the Mas decoder must resort to memory access. The Mas routines also do not exploit the special operations that the SPaSE audio processor executes more efficiently using hardware support.

Count	Data set	MOVE decoder: No unrolling	MOVE decoder: Unrolling 10	Mas decoder	Performance factor
Cycles	Layer I	1402738	1306614	558283	2.51 – 2.34
	Layer II	4112760	3816751	1651162	2.49 – 2.31
Operations	Layer I	1182059	1172267	510956	2.31 – 2.29
	Layer II	3445582	3409027	1519520	2.27 – 2.24

Table 6.3: Comparison of execution cycle counts and executed operations between the MOVE and the Mas decoders. The performance factor is computed as the ratio between the MOVE and the Mas decoder cycle counts.

Function	Layer I MOVE	Layer I Mas	Layer I Factor	Layer II MOVE	Layer II Mas	Layer II Factor
Window	577440	288632	2.0	1550556	866792	1.8
Matrix	549612	153792	3.6	1540252	461376	3.3
Reconstruct	115574	49812	2.3	342648	150012	2.3
Parse_LI	72354	29841	2.4	–	0	–
Parse_LII	–	0	–	84987	50154	1.7
Mixer	46800	15480	3.0	140648	46440	3.0
Deemph	28620	15120	1.9	85860	45360	1.9
Degroup	–	0	–	20592	25344	0.8

Table 6.4: Comparison of the time spent in each function between MOVE and Mas decoders. Only the most time-consuming functions are listed. The performance factor is computed as the ratio between the MOVE and the Mas cycle counts. The MOVE versions are scheduled without loop unrolling.

6.2.3 Cycle count comparison

Table 6.3 shows the comparison between the cycle counts and the operation counts obtained simulating the MOVE and the Mas decoders with Layer I and Layer II data. It can be observed that the operation count factor is lower than the cycle count factor. This indicates that the manual scheduling of the Mas version is better, but also that its code is "denser", as many operations consist of a single move. It is also important to point out that the actual difference between the two factors is expected to be higher, when long immediate fetches are taken into account. Estimations based on the assumptions of subsection 6.2.2 have been computed for the MOVE decoder scheduled without unrolling. The results of immediate usage counts indicate a cycle count 13% higher for the Layer I version of the MOVE decoder and 20% higher for the Layer II version.

Table 6.4 shows the comparison of the execution counts for the most critical functions. The Matrix() function shows the worst performance ratio and is also the second most critical routine, responsible of 39% and 27% of execution time in MOVE and Mas version respectively. This function and Window() are therefore analyzed in detail in section 6.3. It is also interesting to point out the very high performance ratio of Mixer(). The reason for this is that the MOVE version needs to explicitly perform the output sample clipping to 20 bits before the proper mixing in version. In the Mas decoder, result clipping is automatically performed by the add-accumulate and the subtract-accumulate operations described in section 4.1.1 during the IDCT computation and the output signal reconstruction[2]. However, Mixer() is responsible of a minimal part of the execution cycle count (less than 3% on average), therefore it has not been analyzed in detail.

Table 6.5 and 6.6 conclude the comparison giving a valuable hint of which operations are the major responsible of the overhead present in the MOVE decoder. Function unit utilization counts show that:

1. In the MOVE version the utilization of the couples of FUs Rom, Ram and Int, Lim are 'shifted' towards the most specialized units, which are firstly found by the first fit assignment algorithm. Part of the resources available, namely Ram and Int FUs, is therefore underutilized, while the high utilization of the remaining resources increases register pressure, as explained in subsection 6.1.4. Conversely, in the Mas code the utilization is more evenly distributed and it allows a higher number of operations in parallel.
2. The overall contribution of Lim and Int FUs is much higher in the MOVE version, indicating that these units perform the largest part of operations causing overhead.

The MOVE version operation counts show that:

1. The high **add** count is due to address computations for array indexes, stack allocated variables, class data member access and induction index variable update operations. These operations must be compared to the sum of addition and accumulator based operations of the Mas version and show that only 26% of add operations are necessary for proper decoding computations.
2. **load** operations of the MOVE version are 15% less than the equivalent operations of the Mas decoder. The absence of a register file in the Mas version is the main factor.
3. **gt** (compare greater than) operations are mainly due to loop exit condition check. They are about 16 times more frequent than the equivalent subtract and compare `subgt` operation of the Mas version, where most of them are avoided by means of the hardware loop instruction. It is not a coincidence that 16 is also the ratio between inner loops and outer loops iteration count in Window() and Matrix().
4. **shl** (shift left) operations, as explained in subsection 6.1.3, are almost entirely due to address computations. They represent 5.8% of the operations executed by the MOVE decoder.
5. **and, mul** operations are mainly (99.90%) due to address computations of the cosine table elements within the IDCT loops.
6. **store** operation counts of the MOVE version is 22% higher than the Mas version. This is mainly due to insufficient memory disambiguation. When a variable located in memory is repeatedly written within a loop and the memory disambiguator can not ensure that its location is not accessed between two write operations, a store is performed for each variable writing, although only the last store to update the final value is needed.

It is also remarkable that the operation counts for jump instructions (including the Mas `loop` instruction) of the MOVE and Mas versions are comparable, although a 16 times higher count could be expected in the MOVE version, for the same reason that explains `gt` higher operation count. The low jump operation count is due to the way inner loops are scheduled. All inner loops of the critical functions Matrix() and Window(), have a *while loop* structure, with a conditional exit branch within their header block and an unconditional jump at the end of the loop body. Therefore, the exit jumps are executed only once for the entire loop execution, exactly as the `loop` instruction of the Mas version. Moreover, in MOVE scheduled code unguarded jumps are not explicitly scheduled, and they do not affect the operation count.

[2] The result clipping is not necessary in the IDCT computation.

	Layer I		Layer II	
FU	**MOVE**	**Mas**	**MOVE**	**Mas**
Ram	2	34.4	2	34.1
Lim	25	20.7	25	21.2
Mul	8	16.4	9	16.6
Rom	14	16.3	13	16.5
Int	14	10.2	14	9.8
Log	9	2.0	9	1.7

Table 6.5: Comparison of FU utilization percentage between MOVE and Mas version of the decoder. The operations taken into account to compute these values do not include Fet unit operations, namely jump, call for the MOVE decoder and jump, jumps, loop for the Mas decoder. Note that the values of the MOVE FUs are more roughly approximated.

MOVE decoder			Mas decoder		
Operation	**Cycle count**	**Cycle %**	**Operation**	**Cycle count**	**Cycle %**
add	413478	34.97	read rami	122732	24.02
load	193719	16.39	mulfx	79947	15.65
gt	99436	8.41	read romi	76944	15.06
shl	82731	7.00	accl	64512	12.63
mulfx	81072	6.86	addl	24989	4.89
and	36900	3.12	read ram	24199	4.74
mul	36900	3.12	write rami	20626	4.04
store	25256	2.14	add	20341	3.98
sub	22455	1.90	subeq	13283	2.60
eq	7075	0.60	nacl	9216	1.80
jump	23927	2.02	loop	11246	2.20
call	622	0.05	jump	5882	1.15
			jumps	5848	1.14

Table 6.6: Comparison of most executed operations between MOVE and Mas version of the decoder. The operation counts of the MOVE version are grouped apart from the FU they are assigned. The Mas operations computed by memory pointers are grouped in two categories for each bank: auto-incremented and not auto-incremented. Other operations performed by different FUs are kept separated, as their functionality is slightly different. These values refer to the Layer I test stream.

Decoder	Cycle count percentage			Cycle count percentage	
	Non-loop	**Outer loop**	**Inner loop**	**Mirror loop**	**IDCT loop**
MOVE	1	4	95	4.8	95.2
SPASE	6	18	75	5.2	94.8

Table 6.7: Comparison of the execution time distribution of Matrix() functions between non-loop-code, outer loop and inner loop code. These values are approximated. Figures in column 4 and 5 show how the inner loop percentages of the third column are distributed between mirror loop and proper IDCT loops.

6.3 Analysis of critical routines

In this section the two most critical functions of both MOVE and Mas decoders are analyzed in detail. The MOVE code analyzed and shown in this section has been obtained with Layer I profiling, no loop unrolling and inter basic blocks without software pipeline scheduling scope. The reason to avoid software pipelining is that this transformation makes loops difficult to interpret. Since software pipelining affects only Matrix() by a meager 0.5% cycle count reduction[3], we can affirm that the results reported below are perfectly valid for a code scheduled with software pipelining. The Matrix() function, with its performance ratio above 3, is the worst performing piece of code of the MOVE version with respect to the Mas decoder. Also the 1.9 average performance ratio of Window() has a critical effect on the decoder, since this function represents more than 40% of the overall cycle count. The control flow structure is first described. The critical parts of the functions, i.e. the inner loops, are thus identified. A comparison instruction-by-instruction of their body is then described. For each feature of the Mas decoder exploited within the loops, its effect on the MOVE version of the loop is estimated. It is important to remember that the effect on the whole decoder is relevant, as roughly 80% of the cycle counts are due to these two routines.

6.3.1 Matrix

The Matrix() function performs the IDCT computation on a block of 32 subband samples. The algorithm is split in two parts: computation of the mirror vector and proper IDCT computation. The mirror algorithm is performed by a loop of 16 iterations. The IDCT computation, described in subsection 3.3.3 and section 4.3, is structured in an outer loop of 32 iterations and a set of inner loops of 16 iterations, which compute the transformed value of a sample for each outer loop iteration.

In the Mas version there are 4 inner loops, while in the MOVE version they are joint two-by-two to achieve better performance (-8% overall cycle count). Furthermore, the MOVE version computes explicitly the cosine table index at each loop iteration. This algorithm is highly inefficient if compared to the algorithm used in the Mas decoder, where auto-incremented pointers and wrap-around can be exploited, but, when implemented by the MOVE instruction set, it performs 15% better.

Table 6.7 shows that the inner loops are far the most critical part of the function. The percentage is especially high in the MOVE version, indicating that these loops contain considerable overhead with respect of the Mas version. The table also suggests that IDCT inner loops are far the most critical fragment of MOVE Matrix().

[3]The few resources available strongly limit the gain achievable by software pipelining.

Mirror loop

The Mas version of the mirror loop, shown below, consists of 3 instructions containing 5 moves. A loop iteration takes 5 cycles, as the memory load operations by `r1ram` and `r0rami` cause one cycle lock each. For a description of the Mas operations presented below, see subsection 4.1.1.

```
--,                                r1ram.dt -> addl.tr;
addl.rs -> w2rami.dt,              r1rami.dt -> subl.tr;
subl.rs -> w3rami.dt,              r0rami.dt -> addl.op;
```

The MOVE version of the loop, not shown, takes 24 cycles per iteration and consists of 44 moves. The ‘core’ of operations really necessary is in fact the loop body of the Mas version, consisting of two loads with auto-increment from memory, one add-accumulate and one sub-accumulate operation, and two stores with address auto-increment, with all possible result bypasses exploited. Considering the MOVE instruction set, three more moves are necessary for each explicit address increment, therefore a total of 17 moves is the core of operations strictly necessary for the MOVE code[4].

Below are listed the operations of the MOVE version and their purpose. Move slots indicated by `[...]` represent scheduled moves which are not relevant for the operation discussed.

1. **Compare and backward branch.** This operation takes 4 moves and requires 4 cycles. For loops with a fixed number of iterations, as the mirror loop, this operation can be avoided using a hardware loop instruction.

```
r17 -> int.gt_o,          15 -> int.gt_t;
[...],                    int.gt_r -> b0;
[...],                    !b0:6968=5 -> jump;
\\ jump delay slot
```

2. **Loop index increment.** This operation takes 3 moves and requires 2 cycles. It also requires one GPR. This operation can be avoided using a loop instruction, if the index is used only by the compare operation.

```
1 -> int.add_o,           r17 -> int.add_t;
[...],                    int.add_r -> r17;
```

3. **Address computations.** These operations take 14 moves and require 9 cycles. Four of these moves are shift left operations caused by the byte-addressing restriction explained in subsection 6.1.3; they can be avoided by explicit use of pointers instead of array indexes. The first 6 moves, shown below, compute the address corresponding to the array index `s[31*st-i]`; the following 6 moves compute the address corresponding to the array index `s[i]` and increment it. The last two moves compute the address corresponding to the array index `M[k]`. They could be avoided by a better algorithm for induction variable detection.

```
r11 -> lim.sub_o,          r16 -> lim.sub_t;
lim.sub_r -> log.shl_o,    2 -> log.shl_t;
\\ shl one cycle latency
r4 -> lim.add_o,           log.shl_r -> lim.add_t;
r10 -> log.shl_o,          2 -> log.shl_t;
\\ shl one cycle latency
```

[4]Considering all possible bypassing, the operations necessary are reduced to 13. However, for configurations with limited resources, like this one, bypassing is not extensively applicable.

```
r14 -> int.add_o,          log.shl_r -> r3;
int.add_r -> r14;
[...],                     r3 -> int.add_t;
-128 -> lim.add_o,         r12 -> lim.add_t;
```

4. **Memory address increments.** These operations take 6 moves and require 4 cycles. They correspond to the array index increment `k++` and are propagated to the two induction variables representing the array pointers. They can be avoided using auto-increment pointers.

```
4 -> lim.add_o,            r13 -> lim.add_t;
lim.add_r -> r13,          [...];
4 -> lim.add_o,            r12 -> lim.add_t;
lim.add_r -> r12;
```

5. **Induction variable increment.** These operations take 3 moves and require 2 cycles. They correspond to the index variable increment `i+=st` and can be avoided using auto-increment pointers, but also by introducing in the source code an independent variable for each array index.

```
r16 -> lim.add_o,          r10 -> lim.add_t;
lim.add_r -> r16,          [...];
```

6. **Load trigger moves and store operand moves.** These operations take 4 moves and require 4 cycles. They can be avoided using auto-increment pointers.

```
lim.add_r -> rom.ld_t,     [...];
r14 -> rom.ld_t;
r13 -> ram.st_o,           [...];
lim.add_r -> ram.st_o,     [...];
```

7. **Core Operations.** A total of 8 moves are needed for the core operations mentioned above. Three moves are caused by failed bypassing of the load operation results, and could be avoided by assigning one of the load operations to the Ram unit, as explained in subsection 6.1.4.

```
rom.ld_r -> r9,            [...];
rom.ld_r -> r15;
r15 -> lim.add_o,          r9 -> lim.add_t;
lim.add_r -> r6,           r9 -> lim.sub_t;
[...],                     lim.sub_r -> ram.st_t;
[...],                     r6 -> ram.st_t;
```

In table 6.8 the results of mirror loop analysis are summarized. For each group of operation listed above we present the estimated contributions to code size, execution cycle count and register utilization.

IDCT inner loops

The Mas version of the IDCT inner loops consists of 3 moves. Each iteration takes 3 cycles, as the mulfx operation cause a one cycle lock. The load-and-increment-address operations do not cause execution locks, as their values are pre-fetched during the previous iteration. Below is shown the loop body of one of the IDCT loops.

Operation description	Count		
	Moves	Delay	GPRs
Compare and branch	4	4	1
Loop index increment	3	2	1
Address computations	14	9	7
Address increment	6	4	2
Induction var. increment	3	2	2
Load/store moves	4	4	2
Core operations	8	6	3
Total	42	24	11

Table 6.8: MOVE mirror loop analysis. For each operation, the second column shows the number of moves required; the third column shows the number of cycles required; the fourth column shows the number of GPRs used. Note that the figures in the third and the fourth columns are not additive, as the same cycle and the same GPR can be utilized for different operations.

```
r0romi.dt -> mulfx.op,      r0rami.dt -> mulfx.tr;
mulfx.rs -> accl.tr;
```

The Rom auto-increment operation `r0romi` has a wrap-around feature which consents to access the cosine table *modulo 128*. For this operation to work properly, the cosine table must be allocated starting from a 128-word boundary.

The MOVE version of the IDCT inner loops takes 26 cycles and requires 51 moves, but it performs at each iteration the equivalent of two iterations of the Mas loops. The 'core' of strictly necessary operations consists of 4 load operations with auto incremented offset, two fixed point multiply operations and two add-accumulate operations. Including also the explicit address computations for the cosine table and for the mirror vector, 6 additional operations are necessary. Two more operations are necessary for address increments.

The following list describes the operations of the MOVE version and their purpose.

1. **Compare and exit branch.** These operations take 4 moves. The conditional branch is an exit jump, therefore it is taken only once per loop. On the other hand, the unconditional backward jump necessary at the end of the loop body is omitted by the scheduler, therefore the jump operation is not performed at each loop iteration.

```
r16 -> int.gt_o,          15 -> int.gt_t;
[...], int.gt_r -> b0;
b0:9264=20 -> jump,       [...]; \\ exit jump
```

2. **Loop index increment.** This operation takes 3 moves and requires 2 cycles and one GPR (`r6`). The same considerations of the mirror loop index increment apply.

3. **Address computations.** Three addresses are computed at each iteration. The address of `M[k]` is computed by:

```
-128 -> lim.add_o,       !b0:r6 -> lim.add_t;
[...],                   lim.add_r -> r13;
```

This operation is redundant, since the induction variable stored in `r13` is incremented and maintained only for this operation. A better algorithm for induction variable optimization should move this addition before the loop start and get rid of one register use.

The address computations of the cosine table element are reduced to the following expression, consisting of 4 operations: $\mathrm{table\_base} + ((\mathrm{reg}_1 * \mathrm{reg}_2)\&127) \gg 2$ This expression, computed twice, requires 19 moves and 12 cycles, and uses 4 GPRs.

4. **Address increments.** These operations take 6 moves and require 4 cycles. They increment the current addresses of the M vector and can be avoided using auto-incremented pointers.

```
4 -> int.add_o,          r19 -> int.add_t;
[...],                   int.add_r -> r19;
4 -> lim.add_o,          r6 -> lim.add_t;
[...],                   lim.add_r -> r6;
```

5. **Induction variable increment.** The induction variable `(2*k+1)` is maintained, to avoid the expensive mul operation. It is necessary for the cosine table addressing.

```
2 -> lim.add_o, r11 -> lim.add_t;
lim.add_r -> r11, [...];
```

6. **Load trigger moves.** They take 4 moves and require 4 cycles. Only two of them, corresponding to the M vector load operations, can be avoided using auto-incremented pointers, as their result move triggers the next memory access.

```
[...],                   r13 -> rom.ld_t;
lim.add_r -> rom.ld_t,   [...];
lim.add_r -> rom.ld_t,   [...];
[...],                   r19 -> rom.ld_t;
```

Note that the Rom FU is always assigned, despite the Ram unit is available. By assigning the Ram unit for two of the operations two moves and two register uses can be avoided.

7. **Core operations.** These operations require 10 moves and take 6 cycles. Three GPRs are used. Six moves could be substituted by 2 add-accumulate trigger moves, saving the use of two registers. However, it is not profitable to perform such code substitution, because the two add units needed as accumulators are also required for several other operations within the loop body, like address computations and index increments. Splitting the loop back in two loops as for the Mas version could improve the situation, but the ultimate solution is to decrease the number of operations inside the loop, eliminating explicit address and index increments.

```
rom.ld_r -> mul.mulfx_o, r13 -> mul.mulfx_t;
rom.ld_r -> mul.mulfx_o, [...];
r18 -> lim.add_o,        mul.mulfx_r -> lim.add_t;
lim.add_r -> r18,        rom.ld_r -> mul.mulfx_t;
r17 -> lim.add_o,        mul.mulfx_r -> lim.add_t;
lim.add_r -> r17,        [...];
```

Table 6.9 summarizes the results of IDCT inner loops analysis. For each group of operation listed above the estimated contributions to code size, execution cycle count and register utilization are shown.

Operation **description**	**Count**		
	Moves	**Delay**	**GPRs**
Compare and branch	4	4	1
Loop index increment	3	2	1
Address computations	19	12	4
Address increment	6	4	2
Induction var. increment	3	2	1
Load trigger moves	4	4	2
Core operations	10	6	3
Total	51	26	11

Table 6.9: MOVE IDCT loops analysis. For each operation, the second column shows the number of moves required; the third column shows the number of cycles required; the fourth column shows the number of GPRs used. Note that the figures in the third and the fourth columns are not additive, as the same cycle and the same GPR can be utilized for different operations. The figures shown refer to a single inner loop.

Decoder	**Cycle count percentage**		
	Main level	**Outer loop**	**Inner loop**
MOVE	1	4	95
SPASE	6	18	75

Table 6.10: Comparison of the execution time distribution in Window() functions between code out of loop, outer loop and inner loop. These values are approximated.

6.3.2 Window

The Window() function consists of one outer loop executed 32 times and six inner loops. At each iteration of the outer loop, an output sample is computed. The inner loop is first split in two parts, one accessing Vlo and one Vhi. As explained in subsection 4.3, to simplify the loop body, the access path of the two vectors is further split into two loops. The couple of loops accessing Vlo is duplicated, because half of its elements must be inverted before the sum, as explained in subsection 4.3.2. For each couple of loops a total of 8 iterations is performed, distributed between the two loops.

Table 6.10 shows the execution time distribution in the MOVE version and the Mas version. There is a strong difference between the two distributions: in the MOVE version the inner loops are far the most critical part of the function, while Mas inner loops share almost equally the execution time with the outer loop. This is a clear indication that the inner loops of the Mas version are highly optimized.

One of the six inner loops of the Mas version of Window() is shown below. It consists of 2 instruction slots containing 3 moves. A loop iteration takes 3 cycles. The memory load operations `r0rami` and `r0romi` are pre-fetched, but `mulfx` causes an execution lock.

```
r0romi.dt -> mulfx.op,          r0rami.dt -> mulfx.tr;
mulfx.rs -> accl.tr;
```

The MOVE version of the loop, not shown, takes 12 cycles per iteration and consists of 21 moves. The 'core' of operations really necessary is the loop body of the Mas version, consisting

Operation description	Count		
	Moves	Delay	GPRs
Compare and branch	4	4	1–2
Loop index increment	3	2	1
Address increment	6	4	2
Load trigger moves	2	2	2
Core operations	6	4	2
Total	12	21	5–6

Table 6.11: MOVE windowing loop analysis. For each operation, the second column shows the number of moves required; the third column shows the number of cycles required; the fourth column shows the number of GPRs used. Note that the figures in the third and the fourth columns are not additive, as the same cycle and the same GPR can be utilized for different operations.

of two load with auto-incremented from memory, one fixed point multiplication and one add-accumulate (or subtract-accumulate) operation. Here are listed the operations of the MOVE version and their causes.

1. **Compare and exit branch.** The same considerations of the IDCT loop apply. Note that in 3 out of 6 inner windowing loops the loop index is compared against a register value.

2. **Loop index increment.** This operation takes 3 moves and requires 2 cycles and one GPR (`r18`). The same considerations of the mirror loop index increment apply.

3. **Address increment.** These operations take 6 moves and require 4 cycles. They increment the current addresses of the window table D coefficients and of the V vector element. They can be avoided using auto-incremented pointers.

```
136 -> int.add_o,        r3 -> int.add_t;
[...],                   int.add_r -> r3;
256 -> lim.add_o,        r6 -> lim.add_t;
lim.add_r -> r6;
```

4. **Load trigger moves.** They take 2 moves and require 2 cycles. They can be avoided using auto-incremented pointers.

```
[...],                   r6 -> rom.ld_t;
[...],                   !b0:r3 -> rom.ld_t;
```

   Note that the Rom FU is assigned for both operation, despite the Ram unit is available. By assigning the Ram unit to one, a move and one register use can be avoided.

5. **Core operations.** A total of 6 moves and 4 cycles are needed for the core operations. One of these moves could be avoided by means of bypassing. Two more moves can be avoided using an add-accumulate instruction.

```
rom.ld_r -> r15;
rom.ld_r -> mul.mulfx_o, r15 -> mul.mulfx_t;
r16 -> lim.add_o,         mul.mulfx_r -> lim.add_t;
lim.add_r -> r16,         [...];
```

Table 6.11 summarizes the results of windowing inner loops analysis. For each group of operation listed above the estimated contributions to code size, execution cycle count and register utilization are shown.

Feature description	Mirror loop		IDCT loop		Window loop		
	Size	Cycle	Size	Cycle	Size	Cycle	Impl.
Auto-incremented pointers	6.5	2.3	7.1	14.3	9.0	19.2	+/-
Wrap-around increment	–	–	10.6	25.0	–	–	+
Hardware loops	2.1	0.7	6.2	12.5	10.6	22.3	-
Accumulator instructions	–	–	6.2	12.5	4.5	9.6	-

Table 6.12: Estimated effect of each recommanded feature on critical loops: code size and cycle count decrease percentages. For each loop, the first column shows the size gain, i.e. the percentage of saved instructions with respect of the whole function size. The second column shows the cycle count gain percentage with respect of the whole function cycle count. The last column gives an expectation of the implementation complexity, easy '-' to difficult '+'.

6.4 Recommendations

In this section we propose extensions of the MOVE framework and present general recommendations for application writing in C++. In the first part, we discuss possible extensions of the instruction set, taken from the architecture configuration of SPaSE's audio processor. In the second part, recommendations for writing C++ applications based on the analysis of MOVE code are discussed.

6.4.1 Extensions of the MOVE template

The analysis carried out in section 6.3 demonstrates that a consistent part of the functionalities present in the SP8881 audio module, not included in the instruction set of the MOVE architecture template, give a fundamental contribution to the Mas decoder performance.

Table 6.12 presents the estimated contribution of each feature to the code size and the cycle count improvement of the critical loops analyzed: mirror loop, IDCT and windowing inner loops. The estimation for the whole program, considering only the most time consuming loops[5] indicates an overall size decrease of 6% and a cycle count decrease of 48%.

To be conveniently implemented into the MOVE framework, such functionalities must fulfill a set of requirements:

1. They are favorable and widely applicable for the big majority of the ASP applications.
2. They are not candidates for critical path: they do not stretch the processor cycle time.
3. They are cheap to implement in hardware in terms of area, power consumption and overall architectural complexity.

It is therefore interesting to consider their implementation and estimate what should be their effect on the code performance. The four features shown in table 6.12 are discussed in detail below.

Auto-incremented pointers

Read access of memory is well known to be one of the major bottlenecks of processor performances. An effective way to significantly reduce the latency of load operations is to pre-fetch the value that is expected to be accessed by the next load operation. In many ASP applications,

[5] Two more loops, one in initialize() and one in Reconstruct(), are considered.

especially those related to Digital Signal Processing (DSP), most of the memory accesses show a simple and regular pattern, like a constant increment or decrement of the address. These patterns are typically due to vector or array processing and represent the largest contribution to execution time of the MPEG Audio decoder algorithm, as shown in subsection 4.4. In the MOVE decoder, the operations necessary to compute the memory address give the main contribution to the high performance ratio between Mas and MOVE decoder.

A load instruction based on auto-incremented pointers can reduce the latency of load operations which access memory in a regular pattern. Such an instruction requires two preliminary operand moves to specify the starting address and its automatic increment. The operation itself is triggered by the result move, which reads the current address value, triggers the address increment and pre-fetches the value stored in the new address.

The explicit address increment and the load trigger move for each memory access are thus avoided. This saves 4 moves and one register usage for the address value. For load operations with two cycle latencies, 3 cycles can be saved, as shown in the example below:

```
reg -> add_o, increment -> add_t;
add_r -> reg, add_r -> ld_t;
\\ one cycle latency
ld_r -> op_t;
```

In this example, a register `reg` is used to maintain the current value of the address to be accessed. The result is moved to the trigger register of an operation `op`. The same operation with auto increment pointers requires only a single-move instruction, called `rrami` (Read RAM and Increment address):

```
rrami_r -> op_t;
```

If the previous `rrami` operation has been executed more than the latency time in advance, the result value is available and the result move requires no execution lock.

Auto incremented pointers can be implemented also for store operations, giving the same advantages (pre-fetch of course is not present). The main requirements for implementing auto-incremented pointers in the MOVE framework are listed below:

1. Data flow analysis for detecting possible auto-incremented pointers.
2. Introduction of a new scheduling model for the result move.
3. Modification to the simulator.

Detecting auto-incremented pointers requires a data flow analysis of loops. The compiler must select registers used as address of load or store operations; furthermore, the compiler must ensure that such registers are incremented by a fixed value at each loop iteration, and that they are not used or defined by any other operation within the loop. The introduction of a result move that triggers a new operation requires a new scheduling model, which differs from current trigger and result move. scheduling. For example, the FU result can not be bypassed more than once, unless all bypasses are performed in the same cycle[6]. The simulator must also take into account this new type of result moves, that can change the FU state.

Auto-incremented pointers also have disadvantages. An adder within the load/store unit is required, that may be considered as a resource not available for common add and subtract operations. On the other hand, if address increments contribute to a relevant part of the execution time, a specialized adder can be cost-effective, as it is cheaper, faster, and reduces the resource pressure and the scheduling complexity of the other operations.

[6]A better solution, adopted in SPaSE's audio processor, is to provide a 'non-increment' mode, i.e. a source opcode for the result move that does not trigger another address increment.

To implement auto-incremented pointers in the MOVE framework, the following requirements are necessary:

1. The FU template must allow operations triggered by result moves. Currently only input trigger moves can change the result register. The simulation and the scheduling must be adapted.
2. Data flow analysis must be employed to recognize regular memory access paths. The variable that maintains the address value must be defined only by a fixed increment and must be used only as source of load or store trigger moves.

In addition, if multiple pointers are available for each Load/Store unit, the scheduler must perform a pointer allocation. It is also interesting to point out that loop processing, by simplifying the access pattern of vectors, can increment the number of loop in which auto-incremented pointers are applicable.

Loop instruction

This instruction allows to specify loops with fixed iteration count and execute them without incurring in loop overhead, i.e. the necessary operations for index updating and conditional branch. In chapter 7 a detailed description of hardware loops and their possible implementation in the MOVE framework is given.

Wrap-around feature for auto-increment pointers

In some cases the memory access restricted to a memory area is characterized by regular increments with the area size modulo, i.e. the accesses wrap around the area boundaries. Load instructions based on auto-incremented pointers are not applicable in these cases: the memory offset and its modulo have to be separately computed, and then summed to the base address of the area to access. A solution to avoid these operations is to implement auto-incremented pointers with wrap-around capability. An operand move must be introduced for auto-increment pointer instructions, to toggle the state of the wrap-around feature and to possibly specify the modulo size.

A cheap hardware implementation requires the modulo value to be a power of two and the starting address of the memory area to be a multiple of such value. In this case the modulo operation simply requires the conditional addition of the carry value of the pointer bit corresponding to the modulo size. For example, an address increment modulo 2^n requires that the carry of bit n is not propagated.

The automatic detection by the compiler appears difficult and requires extensive use of data flow information. Below is a list of the conditions required:

1. The addressed location must be always computed as the sum of a constant (the base address) and an expression (the offset computation).
2. The last operation performed for the offset computation [7] must be an `and` (modulo operation) with a constant value equal to a power of two minus one.
3. The rest of the offset computation must consist of a fixed increment.

[7] Except for the shift left operation mentioned mentioned in subsection 6.1.3.

The wrap-around feature requires also a specific annotation to indicate a modulo constraint on the memory address. The scheduler must be capable to re-allocate annotated data in an appropriate area. Information about such allocation constraints could be also passed to the parallel MOVE assembler.

Accumulator-based instructions

Many important computations consist of a loop computing sums. Common discrete mathematical transformations, like DCT and DFT, belong to this category, and often represent the critical fragments of DSP applications. For such loops, add-accumulate or subtract-accumulate instructions are a natural choice. The advantages of accumulator based operations are summarized below:

1. At least one result by-pass move is saved. More realistically, 2 moves are saved and one register less is required for each loop iteration.
2. Add-accumulate and subtract accumulate are cheap to implement in hardware: they require only a connection and extra control hardware for the opcode detection.

An FU assigned to accumulator operations represents a resource not available for other operations within the loop. Therefore, only in small loops with relatively low resource requirements it is profitable to map a sum into an accumulate operation. To implement efficiently the scheduling of this operation we need a heuristic to decide when its application is profitable. For example, in all critical loops of the MOVE decoder such an operation is not applicable, as many add-accumulate operations are also required for index and address update. Data flow analysis is necessary to detect the registers used as accumulators inside loops.

6.4.2 Code generation

The choice of the high level language of the implementation and the programming style affect the performance of the generated code. As mentioned in section 5.2.1, many features of C++ produce inefficient code. Here are listed the main sources of overhead found in the MOVE decoder and the C++ features that produced it. It is important to remember that the following considerations depend on the compiler implementation, in this case the GNU g++ front-end, and that they do not represent flaws of the C++ language or its implementations. Instead, they should be considered as characteristics that do not match the efficiency and code size requirements of typical ASP applications, and therefore they should be avoided.

1. Constructor and destructor indirect calls.
2. Access of class data members.
3. Function inlining limitations.
4. Dynamic stack allocation for automatic variables.

Constructor and destructor indirect calls

Constructors provide an automatic means to initialize a class object whenever an object of that class is created [31]. Similarly, a destructor is called to clean up the memory occupied by an object whenever the program execution reaches the end of the object extent. Let us consider the constructors. For each class with a constructor, the compiler creates a function which explicitly calls the constructor for each global object of that class. The pointers of such functions are then stored in a global table. Before executing main(), a library function indirectly calls all the functions whose address is listed in such table. Two levels of calls are therefore added, plus the overhead caused by the loop that walks trough the table of constructor pointers.

Indirect access of data members

The way a data member of a class object is accessed also causes overhead and disables possible optimizations by the front-end. Whenever a data member is accessed by a function member of its class (called also method), a level of indirection is established. In a compiled code, the class object is specified by an implicit argument passed to the method. This argument is the pointer to the object (i.e. to the area occupied by its data members). Data members are accessed exclusively via this pointer: an offset value, statically determined for its class, defines the position of a data member within the data area of the object. Therefore, for each access of a data member, an add operation is needed to compute its address. Furthermore, the presence of an offset makes the job of the memory disambiguator more difficult. Usually no assumptions are made on the content of the memory location of the data member, and many unnecessary load and store operations are generated.

For example, this inconvenient was found in the loops of the Window() routine. Being a data member of the SBfilter class, the pointer to the current block of the V vector caused a load operation each time its value was accessed. In fact, the memory disambiguator was not able to distinguish its location from the locations of the other data member, the V vector, that was continuously accessed within the windowing loop.

In this case, a simple remedy to alleviate this problem is to assign the pointer value to a local value and to use it. Being assigned to a register, the need for load operations or memory disambiguation is excluded and the cycle count of Window() is reduced by 1%. However, this approach is not effective for those methods accessing data members only one or a few times per call.

Function inlining limitations

The current version of g++ cannot inline functions returning a class object, no matter how small and frequently called are these functions. There are situations when the possibility to return a class object and the need to avoid the call overhead are combined: for example, when we are designing functions for operator overloading.

Dynamic versus static stack allocation

The temporary variables that can not be assigned to registers are stored in stacks. The C++ compiler allocates automatic variables in dynamic stacks, using a stack pointer register. An interesting feature of the Mas assembler is the static allocation of local variables. As in the C language, a variable declaration within the function body represents a local variable that is allocated each time the function is called. The Mas assembler statically assigns a fixed location for each local variable encountered at the beginning of a function body. To save the instruction bandwidth, the first 16 locations of data RAM are first used for the static stack, so that a source immediate is sufficient to specify their address. Local variables belonging to functions that do not call each other can be overlayed into the same location. Both stack solutions have pros and cons:

1. Static stack allocation does not allow recursive calls, but this is not frequent at all in ASP applications. Static stack size should be tailored to the specific applications and determined at compile time. This restriction is acceptable for ASP applications, especially for embedded systems.

2. Using static stacks the variable address computation is reduced to an immediate move. If the static area is small, the immediate can be fit into the source field of the move. On the contrary dynamic allocation requires a stack pointer register and implies additional address computations.

3. Dynamic stacks require extra operations to update the value of the stack pointer value each time a function is entered or left. This may cause significant overhead for small functions.

To implement static stack allocation in the MOVE framework two alternatives seem possible. The first radical alternative is to adapt the g++ front-end module. This option seems difficult to implement and has the disadvantage of requiring changes of the only part of the software subsystem that should be maintained as standard as possible. In fact, one of the good qualities of the MOVE framework is the possibility to re-target for the MOVE code generation the front end of any standard compiler with relative ease, since in current front-ends the intermediate code generation is based on a description language. Second, a tool executed before the scheduler could transform all dynamic allocations in static. This corresponds to 'back-tracking' the allocation job performed by the front-end and substitute all the memory accesses via stack pointer with absolute address accesses.

A simpler approach to alleviate this problem is to avoid automatic variables and use static local variables. This substitution must be performed with care. In fact, static variables always reside in memory and must be updated each time the function where they are defined returns to the caller. Moreover, if the memory disambiguator is not able to resolve a memory access involving the variable, each time it is written (read) a store (load) operation is performed.

An experiment performed on the Matrix() function of the MOVE decoder shows this problem. In a first test, by declaring only the mirror vector as static, the cycle count of the function dropped by 0.8%. On the contrary, when the static qualifier is extended to all the local variables of Matrix(), the cycle count raises by 4.5%. This is because static variables must be stored in memory to 'survive' their function call. For example, the variable which accumulates the value of the sample to be transformed is read and stored in memory at each inner loop iteration.

n the contrary, the mirror vector is first computed and then only accessed for reading during the proper IDCT computation, therefore it does not require any memory disambiguation or additional load and store operations.

6.5 Conclusions

In this chapter we have pointed out the main causes of inefficiency of the MOVE compiled code and which features can be implemented in order to remove such inefficiencies. The analysis performed shows that:

1. The architecture template should be tailored around the specific application. A part of the present limits are unacceptable for a real ASP application. We pointed out 5 of these limits and discussed possible approaches to remove them. Chapter 8 discusses in detail one of these limits, namely the bit size of architecture data path.

2. A set of functionalities not included in the standard instruction set of the MOVE template gives a fundamental contribution to the performance achieved by the custom realization of the decoder. Such functionalities present the common drawback of requiring specialized hardware resources, which are not utilized for generic operations. A detailed analysis of the requirements of a large set of benchmark applications is therefore a preliminary step necessary to establish whether these functionalities may be cost-effective or not on the majority of target applications for the MOVE framework.

3. The high level language used for the implementation and the programming style has a relevant effect on the quality of the compiled code. Some features of a powerful, object oriented language like C++ are compiled into highly inefficient code. We presented four recommendations writing source code that do not produce such inefficient compiled code.

The results of such analysis can only suggest what can be the impact of a feature on the performance of a generated code, what are the costs of its implementation in terms of algorithm complexity, compilation time, and efficiency. The final test can only be the implementation of such extensions in the framework and the evaluation of the results. In this spirit, chapter 7 presents the results of a prototypical implementation of the hardware loop instruction and chapter 8 describes a possible approach to implement support for FUs with definable data sizes.

7 Hardware Loops

This chapter treats the implementation of the *loop instruction* in the MOVE software subsystem. Such instruction allows to specify the hardware supported execution for loops. The loops that satisfy the requirements needed to be executed in this way are called hardware loops. Section 7.1 introduces the hardware loops, gives an example of code using a loop instruction and a possible hardware implementation. The general requirements needed to detect and map hardware loops conclude this section. The detection and mapping algorithms implemented in the MOVE framework are the main subject of section 7.2. Since hardware loop instructions modify the control flow structure of the program, they affect the MOVE scheduler and require changes of the scheduling routines. Such requirements are presented and motivated. The effects of combining hardware loop mapping with loop unrolling and software pipelining conclude this section. A summary of the results obtained from the first implementation is presented in 7.4 and concludes this chapter.

7.1 Characteristics of the hardware loop instruction

In this section the general characteristics of the loops that can efficiently use hardware support are discussed. Hardware support reduces the execution cycle count by removing a set of operations called *loop overhead*, which are needed to control the trip count.

7.1.1 Problem addressed

In a loop body, part of the operations are necessary to control the number of iterations executed, but it does not perform any useful computation. These operations represent the loop overhead, and typically consist of loop index updating and conditional branch to the loop header. For applications spending most of execution time within loops, the loop overhead may have a significant performance impact. In the MOVE decoder, the loop overhead of the inner loops of the subband filter algorithm is responsible of roughly 19% of the total execution time. For a large part of these loops the overhead can be avoided by means of a *loop instruction*. A loop instruction provides a simple way to specify fixed iteration loops: its operand move specifies the number of times the loop body must be performed, its trigger move specifies the address of the last instruction of the loop body. The loop body starts at the instruction following the loop instruction[1] and it is executed the number of times specified. The advantages of hardware loops are summarized below:

1. Elimination of the jump latency due to the backward jump. The first instruction of the loop can be in fact easily pre-fetched, as each branch destination is known even before starting the loop.

2. Reduction of GPR pressure. The register maintaining the value of the loop index is no more necessary. Loops are usually the fragment of code with the highest register pressure, therefore the elimination of a register use within loops can lower the maximum number of registers live at the same time.

3. Elimination of the operations required to update and compare the loop index. These operations are in fact performed in parallel by specialized hardware of the loop instruction.

4. Possibility to combine the gain provided by hardware loop and the other loop enhancement techniques currently implemented in the MOVE framework: loop unrolling and software pipelining. Software pipeline and loop unrolling improve performance by enlarging the scheduling scope and allowing a better resource usage. Loop instructions remove operations that can be concurrently and more efficiently performed by a specialized hardware[2].

5. Reduction of code size.

Possible disadvantages of a hardware loop instruction are:

1. Specialized resources are needed within the instruction unit, which are not utilized apart from the hardware loop execution. These resources consist of three registers and one adder, plus the logic to overrule the instruction fetch. Whether they pay themselves or not, it depends on the type of application, i.e. how much loop intensive it is.

2. It is applicable only to a limited set of loops, whose iteration count can be computed before entering the loop.

3. Interference with scheduling is possible. Loop unrolling, software pipelining and hardware loops influence each other, restricting the scheduling freedom. The negative effect on code performance should be negligible.

7.1.2 Example of hardware loop mapping

In TTAs loop overhead can be a consistent percentage of the whole loop execution time: a conditional jump takes 4 cycles, while index updating can take 1 or 2 cycles, depending on the presence of by-passing. Below is shown an example of MOVE sequential code with fixed iteration number for loops with exit condition check at the end (*do-until* loop type).

[1]More realistically, at least one delay slots may be required after the loop instruction.

[2]Loop unrolling can also reduce part of loop overhead, but this is not implemented in the current MOVE scheduler.

```
        n_iterations -> reg;
header:
        loop body

        reg -> sub_o, 1 -> subt_t, subt_r -> reg;
        reg -> gt_o, 0 -> gtt_t, gt_r -> bool;
        bool: header -> jump;
```

Below is shown an example of MOVE sequential code with fixed iteration number for loops with exit condition check at the start (*while* loop type).

```
        n_iterations -> reg;
header: reg -> gt_o, 0 -> gtt_t, gt_r -> bool;
        bool: exit -> jump;

        loop body

        reg -> sub_o, 1 -> subt_t, subt_r -> reg;
        header -> jump;
exit:
```

Both loops can be mapped in the following code using a loop instruction. In the case of *while* loops, if it cannot be proved that the iteration number is always greater than zero, it is necessary to introduce a conditional branch to `exit` before the loop entry point. A fast hardware implementation of the loop instruction, as the one depicted in figure 7.1, requires in fact the hardware loop to be executed at least once. This problem does not exist when the number of iterations is known at compile time.

```
        n_iterations -> loop_o, end -> loop_t;
header:
        loop body
end:

exit:
```

7.1.3 Example of hardware implementation

Figure 7.1 shows a possible hardware implementation of the loop instruction. Its main building blocks are an adder with a fixed operand -1, a comparator, three registers and a multiplexer. The *lpend* register contains the address of the last instruction of the loop. It is compared to the program counter (PC) to detect when the end of the loop is reached. Each time the end of the loop is reached, the value of the loop index *lpno*, decremented by one, is latched back into the index register, and the demultiplexer selects the address of the subsequent instruction to fetch. If the index register is not negative, the first instruction of the loop, pointed by *lpstart*, is selected. Not shown in the figure, a path from the result of the multiplexer to the PC must be present to update it with the value of the loop header whenever a new loop iteration must be initiated.

The hardware configuration can be tailored over the application to minimize its resource requirements. By imposing limits on the size of the loop body and the maximum number of iterations of loops that can be mapped, the number of bits of *lpno* and *lpend* registers can be reduced, as well as the data-path of adder and comparator. For example, the test on the MOVE decoder described in section 7.3 shows that the maximum iteration count of mappable loops is 272, therefore a 9 bit data path is sufficient for *lpno* and the adder.

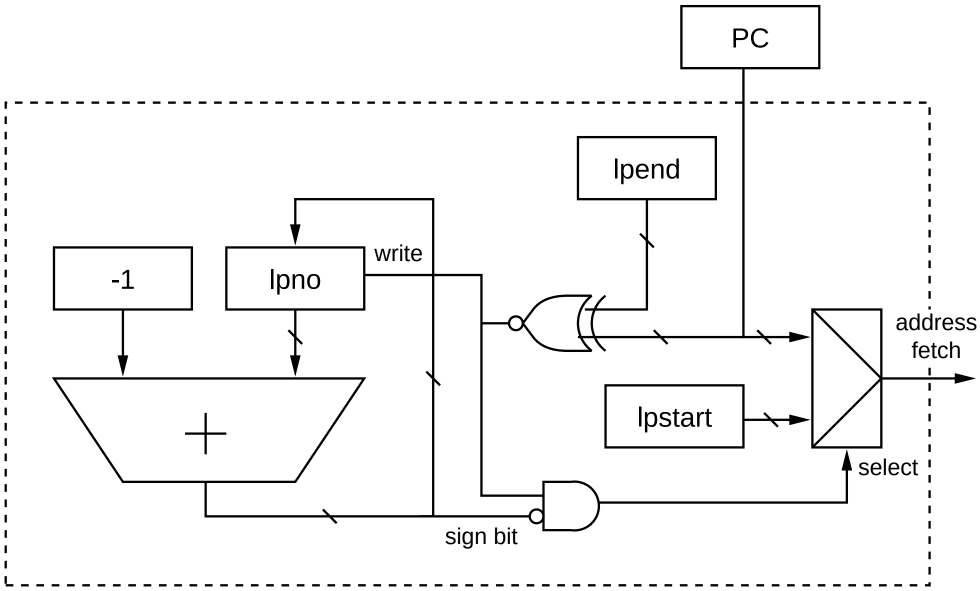


Figure 7.1: Scheme of a possible hardware configuration for the loop operation. The choice to update the loop index when the end of the loop is reached allows a simple design.

7.1.4 Estimation of performance improvement

As shown in subsection 7.1.2, the number of moves saved for each iteration count ranges from 7 to 8, depending on the type of loop. The cycle count reduction can not be determined, as it depends on the resources of the configuration and the code present in the loop. As extreme case, let us consider a large configuration and a large loop body. It may be the case where the gain given by hardware loop transformation is zero, when all the moves forming the loop overhead can be scheduled in move slots not utilized otherwise. The jump delay slot can be filled by instructions of the loop, making the header instruction pre-fetch and the jump elimination ineffective.

However, in a small configuration, especially for loop bodies consisting of few instructions, the expected speed gain is relevant, around 3-4 cycles per iteration.

7.1.5 Hardware loop requirements

A loop must satisfy a set of conditions to be mapped into a hardware loop, i.e. to be transformed as shown in the example of subsection 7.1.2 and executed with hardware support. It is important to point out that most of the critical loops present in computation-intensive ASP applications are expected to satisfy these conditions. Below are listed the most general requirements that a loop must satisfy to be mappable:

1. Since the resources used by hardware loops are unique, nested hardware loops are not allowed. This implies that the mapping must start from the inner loops (i.e. the most profitable to transform) and move to lower level loops only when all their inner loops have not been mapped.

2. If a loop contains function calls, these have to be recursively checked for the presence of already mapped hardware loops.

3. Multiple backward edges are not acceptable, since they cause a loop iteration not detectable by the hardware. In fact, the fastest hardware loop implementations require the internal index register to be updated when the PC equals to the last instruction of the loop.

4. The number of iterations performed must be known before the loop is entered. This implies that the loop index can not be re-defined within the loop, except for a single increment/decrement operation. The loop increment step must be constant within the loop.

A real implementation can impose further restrictions to make the detection and mapping algorithms simpler. The most significant restriction is to consider loops with constant iteration count. The trigger move of the loop instruction requires the trip count to be computed; if the loop iteration count is not constant (i.e. it depends from registers or values stored in memory), its number of iterations must be computed at run time, and extra operations must be introduced for this purpose. However, it is not always possible or profitable to map loops which require the introduction of a run time operation. For example, an expensive operation as integer division may be not profitable for loops with a small number of iterations; or it may be not even present in the instruction set. Therefore the machine configuration itself imposes limits to the set of mappable loops.

7.2 Generating hardware loop code

This section describes the implementation of the loop instruction in the MOVE framework software subsystem. First we describe the alternatives available for implementing the hardware loop functionality, and we motivate the chosen approach. The problem can be split in two subproblems: detection of hardware mappable loops and loop instruction generation. In the current version they are executed in strict sequence, but it may be profitable to detect hardware loops in advance and then perform the code transformation. The detection and mapping algorithm are presented. The problems related to the scheduling of mapped loops and their interactions with loop unrolling and software pipelining conclude this section.

7.2.1 Requirements of hardware loops detected by the implementation

To achieve a fast implementation we fixed a set of limits more stringent than those presented in subsection 7.1.5:

1. Only loops without function calls are selected. In this way any recursive check of function calls is avoided.
2. Only inner loops are selected. This restriction could be easily removed, but is unlikely to give noticeable improvements, since outer loops are not critical.
3. Only loops with a constant number of iterations are accepted. They require unique loop index initialization by an immediate value, constant loop boundary and fixed loop step. These conditions ensure that the number of iterations can be statically computed, and that the operand move for the loop instruction is an immediate value.

As explained in subsection 7.1.5, relaxing the last condition could significantly extend the set of loops which can be mapped, at the expense of the possible introduction of operations for computing at run time the number of iterations.

7.2.2 Implementation alternatives

The first question related to the implementation of loop instruction is: in which stage of the code scheduling process is preferable to introduce loop detection and mapping algorithms? The scheduling flow takes as input a MOVE sequential (unscheduled) code and consists of the following steps:

Sequential code pre-processing
Function inlining
Select software pipelinable loops
Loop unrolling
Loop peeling
Scheduling
Register allocation
Instruction scheduling
Complete software pipeline scheduling

The requirements of the algorithm give an indication of which are the reasonable boundaries for the implementation of a hardware loop code generation within such execution flow:

1. Both control flow and data flow information is used extensively, therefore the computation of this data should be performed before the hardware loop detection.

2. Function inlining can eliminate function calls inside hardware mappable loops, simplifying the task of hardware loop detection.

3. If the transformation is performed before the register allocation, the reduction of register requirements is automatically taken into account by the register allocator. This promotes higher by-pass ratio during scheduling and reduces the possibility of register spilling.

4. The current algorithm of loop unrolling does not allow to detect and map unrolled loops. On the other hand, hardware loops can not be unrolled by the current unrolling algorithm.

5. The current software pipelining algorithm does not work on hardware loops, therefore the software pipelinable loops must be skipped by the mapping function. The modifications required to extend hardware loops to software pipelined loops are not difficult, though.

In this preliminary implementation it was choosen to perform detection and mapping of hardware loops after the complete sequential code pre-processing, skipping software pipelined loops. By performing hardware loops after unrolling we give a higher priority to the latter. This choice presents the important advantage to make the implementation simpler, as it works on sequential code.

7.2.3 Hardware loop detection and mapping

Figure 7.2 shows how mappable loops are identified. The program starts discarding non-inner loops and loops containing function calls or multiple backward edges. Next, the loop type is determined according to the type of backward jump: a guarded jump implies a *do-until* loop type, an unguarded one implies a *while* loop type. In the latter case, the header block must contain a guarded exit jump. In both cases, the comparison operation defining the guard value is selected. To be compatible with the requirements stated in subsection 7.2.1, the operands of the comparison must be an immediate value, representing the loop index limit, and a register, that contains

Procedure Detect hardware loops
  Discard non-inner loops, loop with function calls, loop with multiple backward edges
  if backward jump is guarded
    check the *compare* instruction defining its guard
  else if loop header has a guarded exit edge
    check the *compare* instruction defining its guard
  else
    return FALSE
  select the operation *incr* which defines the non-constant operand *reg* of *compare*
  verify *incr* is an increment operation and check its def and use chains
  check initialization of *reg* outside the loop

Figure 7.2: Hardware loop detection algorithm

Procedure Map hardware loops
  Read values needed to evaluate loop parameters
  Compute the iteration number
  Insert the loop instruction
  if *while* type loop
    Delete conditional exit jump
  else
    Delete conditional backward jump
  Delete compare and index increment operations
  Update control flow information
  Set hardware loop flags

Figure 7.3: Hardware loop mapping algorithm

the loop index. The sole operation[3] within the loop which defines the register operand is then checked: it must perform a constant increment or decrement of the register. Furthermore, the so called def and use chains are checked. The sole loop-carried reaching definition (def chain) of the register operand move must be the result move of the same increment operation. Similarly, the sole uses (use chain) of the increment result allowed must be the compare operation and the increment operation itself (executed in the following iteration). After these conditions, the out-of loop initialization move of the register used as loop index is checked. It must unique and must reach either the compare operation (for *while* type loops) or the increment operation (for *do-until* type of loops) and have an immediate as source operand.

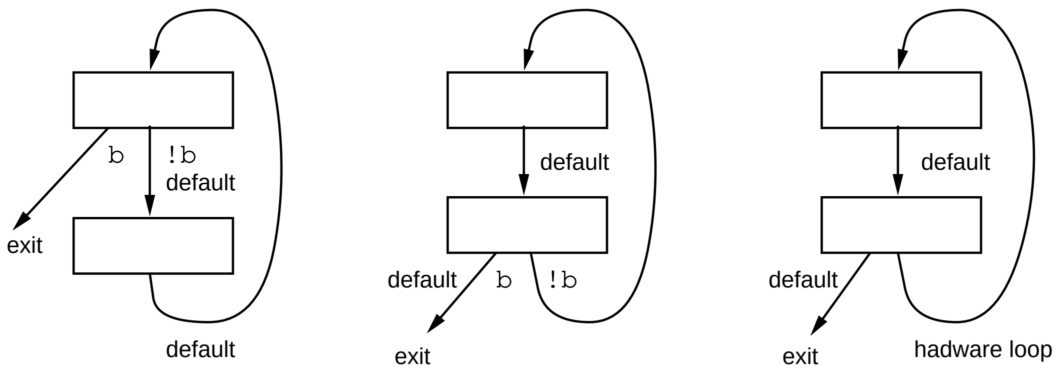
Figure 7.3 shows the algorithm for mapping hardware loops. First, all information necessary to establish the iteration count is collected. The immediate values of compare, increment and loop index initialization are read. The condition code to iterate is evaluated from the compare operation, the type of conditional jump (exit or backward) and the guard expression. The number of iterations, when the condition does not include the loop index limit, is then computed using the simple formula

$$\left\lceil \frac{\mathrm{index\_end} - \mathrm{index\_start}}{\mathrm{index\_step}} \right\rceil \tag{7.1}$$

when the condition includes the index limit the formula is

$$\left\lfloor \frac{\mathrm{index\_end} - \mathrm{index\_start}}{\mathrm{index\_step}} + 1 \right\rfloor \tag{7.2}$$

[3]An initialization out of the loop is also allowed.



(a) while type loop (b) do-until type loop (c) mapped hardware loop

Figure 7.4: Control flow transformation applied to mapped loops. For conditional jumps, the default edge corresponds to the non-taken jump direction. Guards are shown as `b` and `!b` (inverted guard).

otherwise. Second, the loop instruction is generated and the loop overhead operations are removed. After the code transformation, the control flow information is updated. Figure 7.4 shows the changes applied to the control flow edges of the loops. The exit edge of *while* type loops is deleted and a new default edge from the block containing the backward edge of the loop is inserted. Edge guards are removed. A set of flags has been introduced for passing information to other modules of the MOVE software subsystem. These flags are attached to the data structures which represent (1) loops, (2) basic blocks and (3) control flow edges. Such information is currently used by:

1. The output module, to visualize hardware loops in MOVE assembly code.
2. The extended basic block modules of the scheduler performing inter basic block code motion and software pipelining.

7.2.4 Scheduling with hardware loops

The substitution of the loop instruction in the sequential code and the elimination of the operations that represent loop overhead is straightforward. What poses problems instead is to schedule a sequential code containing hardware loops. Several scheduler functions assume in fact that a block with multiple successors must have a guard expression for each edge, and that only a single edge can be unguarded. As shown in figure 7.4, a loop instruction introduces an edge which is not always taken, but does not have a guard expression.

The functions related to data dependency graph construction, extended basic block scheduling and software pipelining expect to find a guard in such control flow edges. The situations in which the scheduler assumes that a guard must be available are listed below:

1. The scheduler when building a data dependency graph from the program structure and its control flow, looks for a guard expression for multiple control flow edges going out of a basic block.

2. In inter basic block scheduling, code motion between basic blocks may require speculatively executed operations to be guarded. When the code motion to hardware loop blocks from successor blocks out of the loop requires a conditional execution, the scheduler fails to find a guard for the exit edge.

3. In software pipelining, the scheduler tries to move operations in different iterations taking into account *Next iteration speculation* (NIS) dependences [10]. The scheduler assumes that a compare operation O_{comp} and a guard are present in the loop body and tries first to introduce a NIS dependency from O_{comp}, which requires imported trigger moves to be guarded.

In order to make hardware loops compatible with inter basic block and software pipelining scheduling, it is necessary to restrict the code motion in situations (2) and (3). In the current implementation, hardware loop mapping is disabled for loops selected for software pipelining, while inter basic block scheduling scope is not working.

7.2.5 Hardware loops and other loop enhancements techniques

In principle, a hardware loop instruction is applicable to unrolled and software pipelined loops, giving additional performance gain with both. The performance gain given by loop unrolling and software pipelining is due to the extension of the scheduling scope and the better intra iteration parallelism that can be achieved. Hardware loops simply remove a group of operations that can be performed more efficiently by a specialized hardware. This suggests that these techniques can be effectively combined. However, hardware loop mapping do interact with loop unrolling and software pipeline. The scheduling freedom of software pipelinable loops over backward control flow edges is restricted, as explained in subsection 7.2.4. This can enlarge the Minimum Initialization Interval (MII) and increase the cycles needed to execute the loop, as explained in [10].

In the current version of the scheduler, unrolled loops are not mappable according to the conditions presented in subsection 7.1.5. The loop unrolling routines duplicate the entire loop body while maintaining all index increments[4] , and compare operations. Unrolled loops contain thus multiple definitions and uses of the loop index. A better unrolling algorithm can eliminate unnecessary loop index definitions and uses, and allow hardware loop mapping. In this case, the performance gain given by hardware supported execution is divided by the unrolling factor (the number of times the loop body is duplicated). This negative effect is compensated by the gain given by loop unrolling, which is directly related to the unrolling factor. The test carried out in subsection 5.3.2 suggests the presence of an optimal point which combines resource utilization and size increase of loop unrolling with the performance gain obtained by hardware loop instruction. For small TTA configurations the optimal point is 'shifted' towards hardware loop mapping, while for configurations with several FUs and buses, the gain given by better scheduling of unrolled loops is more significant.

7.3 First results and future enhancements

This section analyzes the improvements achieved by the first implementation of hardware loop detection and mapping algorithm. To test the algorithm, we chose the MOVE decoder described in chapter 5. The input data set is the same Layer I stream used for the simulations described in chapters 4 and 5. Suggestions for extensions of the algorithm conclude this section.

[4] If the index increment in constant, copy propagation is performed to reduce the number of operations.

Category of loop	Mapped	Loop Number	Loop Percentage
Non-inner loops	No	4	11.4
Loops containing function calls	No	10	28.6
Loops with multiple backward edges	No	0	0.0
Multiple definitions of index	No	3	8.6
Multiple uses of index	No	1	2.9
Non-constant loop index end	No	7	20.0
Non-constant loop index step	No	3	8.6
Non-constant initialization	No	1	2.9
Hardware loops	Yes	6	17.1
total	–	35	100.0

Table 7.1: Results of the hardware loop detection algorithm applied to the MOVE decoder. Failed loop mapping and mapped loops.

7.3.1 Simulation results

The implementation test has been split in two parts: the first verifies the detection algorithm, establishing how many loops can not be mapped and their causes. The second part verifies through simulation the gain given by hardware loops on cycle count and code size.

The detection algorithm has been tested on the MOVE decoder, with all scheduling scopes and with loop unrolling aggressiveness 0 and 10. Only 6 loops are detected and mapped. Table 7.1 shows the reasons why 29 in 35 loops are not mapped. Note that the number of mapped loop itself is not much significant: a single mapped critical loop can give more gain than tens of loops. From this point of view, our implementation performs rather well, as 5 on 6 detected loops belong to the three most time consuming functions, responsible of roughly 88% of cycle counts.

The best detection results, obtained without loop unrolling[5], show that only 4 out of 11 critical loops present in Window() and Matrix(), which are responsible for 76% of the overall cycle count, are detected by the current algorithm. They all belong to Matrix(). Two more loops are detected in the reminder of the program. The former initializes the V vector and is executed only once, therefore it gives minimal gain. The latter belongs to Reconstruct() and performs sample rescaling. Reconstruct() is the third most critical function and is called 36 times throughout the simulation. It is important to remember that all the critical loops of Matrix() and Window() fulfill the requirements presented in subsection 7.1.5 and they could be mapped by a better algorithm.

The inner loops of Window() are not mappable by this algorithm because their iteration count is not constant. This can be due to different reasons. The most frequent is the presence of a variable operand in the compare operation that checks the loop index. In subsection 7.3.2 the algorithms necessary to remove this and other limitations are discussed.

The simulation of the code using loop instruction has been performed only with the decoder version scheduled with inter basic block scope. The implementation of hardware loops in inter basic block and software pipeline scheduling scopes requires several modifications of the scheduler module, not yet implemented. The part of the mapping algorithm for *do-until* type of loops works also with inter basic block scheduling scope, but the results are not interesting, as

[5]Loop unrolling does not allow the detection of 2 more loops.

Version	Cycle count	Operations	Size
Without loop instruction	1506359	1118622	2048
With loop instruction	1382763	1045878	2008
Improvement percentage	8.9	7.0	2.0

Table 7.2: Performance improvement achieved by loop instruction on the MOVE decoder.

Function	Loop description	Iteration count	Decrease % Size	Cycles
Matrix()	IDCT computation	16×17	29.4	19.03
Matrix()	Mirror vector computation	16	37.4	1.28
Initialize()	V vector initialization	272	44.4	44.34
Reconstruct()	Sample rescaling	64	12.0	10.09

Table 7.3: Effect of the hardware loop instruction on mapped loops. The first column of figures shows the number of iterations executed for each function call. The second column shows the code size reduction of the loop body. The third column shows the contribution to cycle count decrease of the whole function which the loop belongs to.

only two non critical *do-until* hardware loops are present in the MOVE decoder, and the performance gain is only 0.5%.

Tables 7.2 and 7.3 show the improvement achieved by using hardware loop instruction on the whole program and on the functions affected, respectively. The execution cycle count and operation count decreases roughly correspond to half the performance gain expected when all loops satisfying conditions presented in subsection 7.1.5 are detected and mapped. It is interesting to notice that the performance improvement is comparable to what can be achieved in inter basic block scheduling scope by applying loop unrolling, but it does not pay a relevant size increment: in this specific application, inter basic block scheduling scope gives a smaller improvement than our limited implementation. This gives an indication of how relevant it can be the gain due to hardware supported loop execution.

7.3.2 Implementation enhancements

Several extensions and enhancements of the current algorithm are possible.

1. De-coupling hardware loop detection from mapping.
2. Extension to inter basic block scheduling.
3. Extension to software pipeline scheduling.
4. Extension to loop with non-constant iteration count.

By making hardware loop detection an independent step from mapping, it is possible to pass information to other modules for sequential code pre-processing. In this way it is possible to select, by means of specific heuristics, the optimal combination of loop enhancement techniques to apply to each loop.

The inter basic block code motion to hardware loop blocks from outer blocks must be disabled if the operation imported to the loop needs to be conditionally executed. This restriction should have minimal effect to code performance.

In software pipeline scheduling, moving operations over backward edges (i.e. over iteration boundaries) requires NIS dependences and possibly conditional execution. In hardware loops, the tail of a NIS dependence edge from compare or conditional jump operations (O_{comp} and O_{jump}, respectively) must be re-routed to the last scheduled operation of the previous iteration. In non-hardware loops, the dependency from O_{comp} is prefered, as it allows to schedule the moved operation L_{comp} cycles after O_{comp}, instead of $L_{comp} + L_{jump}$, where L_{comp} and L_{jump} are the values of compare and jump operations respectively. In hardware loop code, NIS dependency tail must point to the last operation performed in the loop iteration, and the dependency delay corresponds to the operation latency. A part from this, the scheduling freedom of software pipelining is unaffected, and the gain of removing loop overhead should combine with the gain given by software pipelining and largely compensate the limitation above described.

The mapping algorithm can be easily extended for a large part of hardware loops with non-constant iteration number. Loops with fixed step value 1 and one or two index limits stored in registers require only the introduction of an extra subtract operation to be mapped, as it is evident from equation (7.1). An addition may be also necessary when both index limits are registers and the loop condition includes the index limit; see equation (7.2).

7.4 Conclusions

The preliminary implementation of the loop instruction shows that a restricting set of requirements significantly limits the percentage of mappable loops, but still gives a significant speed improvement: almost 9% cycles less are required for the MOVE decoder.

The test has been performed on a limited set of scheduling parameters: the most restrictive limitation is the scheduling scope, Intra Basic Block. Nevertheless, the performance gain encourages the extension to larger scheduling scopes, namely inter basic block with and without software pipelining. Such extensions require a significant number of modifications in the scheduler code. Also the detection and mapping algorithms have substantial room for improvement. Extensions to loops with non-constant trip count are expected to double the cycle count decrease on the decoder application. Conversely, extensions to loop containing function calls and non-inner loops should not give significant improvement.

The MPEG decoder application used for our test gives a motivating example for hardware loop implementation, but it must be demonstrated which is the average gain for a large set of benchmark. The requirements necessary to hardware loop mapping suggest that a significant contribution to performance is expected only for computation-intensive applications, and typically for digital signal processing, which is an important class of candidate applications for ASPs. It is also important to remember that loop processing, both at source code level and during the front-end compilation step, can significantly increase the number of mappable loops.

Support for multiple data widths

8

This chapter gives an introduction to the problem of extending the MOVE framework to architecture configurations with data word widths definable by the ASP designer. Implementing such feature is an important step towards a flexible architecture template capable to cover efficient design solutions for a wide range of applications. In fact, one of the fundamental architectural parameters of an ASP related to the application requirements is the data word bit width. In this chapter we will focus our attention on integer data types and integer data paths, but the extension on floating point is straightforward.

The current MOVE framework fully supports 32 bit architectures only. In section 8.1 we analyze the causes of such limitation, and define what are the specifications necessary to implement support for any data width in the MOVE framework. Owing to the high complexity of the modifications required, we narrow the problem to implementing support for multiple data type and FU data path widths. Section 8.2 presents the specifics of such subproblem and a possible approach for implementing them. The implementation, based on C++ classes and operator overloading, is then described. The first results, presented in section 8.3, although demonstrate it is possible to achieve the goal, are not encouraging, as they show a highly inefficient MOVE code, caused by the class implementation. Section 8.4 concludes this chapter mentioning possible alternatives to solve this problem.

8.1 Problem addressed

Custom ASPs can have any data path width, scaled to the precision requirements of their specific application. For small ASPs the possibility to scale down the width of data words can have a critical impact on performance and realization costs. Chip area and power consumption are in fact directly related to their data path width. For instance, the area occupied by multipliers is quadratically proportional, and reductions of data path have a dramatic effect on the layout: a 20 bit multiplier occupies roughly 40% of the area occupied by a 32 bit analogous unit. Also cycle time is affected by the data path width, although in a less direct fashion. An adequate template for ASP architectures should therefore provide the possibility to independently specify:

1. Multiple data path widths for processor components: FUs, RFs and buses.
2. Memory word width.
3. Width of the smallest addressable memory sub-word.

These features conflict with the high level languages used in the MOVE framework to design applications, C and C++[1]. Such languages are based on a determined set of rules for assigning type expressions, called *type system*[1].

8.1.1 Limits of the current MOVE framework

In the MOVE framework the data word width is not explicitly fixed by the architecture template, but it is restricted by the source language type system and by the front-end of the software subsystem. The gcc front-end produces an internal intermediate representation of the program based on three-address code [1]. Such representation, called *Register Transfer Language* (RTL), is designed to be as much as possible machine-independent and easy to be retargeted to a real architecture code. RTL is translated into assembly code of the target architecture by a separate module of the compiler for code retargeting, which specifies the rules for mapping RTL instructions into instruction of the target machine. In the MOVE software subsystem, the target code is MOVE sequential code for MOVE generic architectures, as described in section 2.1.

Both the source language and the RTL representation produced by the gcc front-end are based on a set of data types whose width is determined by the data path expected for the 'typical' target architectures. The main limitations to the data width encountered in the software design trajectory are listed below:

1. C and C++ languages define `char`, `short`, `int` and `long` as base integer types and set limits for their width: for example, the `char` type must be capable to represent any symbol of the character set of the architecture, which typically requires at least 8 bits, and its size must be smaller or equal to `short` type [24].

2. The gcc compiler sets `char` to 8 bits, `short` to 16, `int` and `long` to 32 bits, which are the typical values of 32 bit RISC architectures for workstations[2], and it produces RTL code based on these data widths: registers and FUs are considered 32 bit wide.

3. The use of 8 or 16 bit data words requires a sign extension each time they are to be copied to larger data words. During RTL code retargeting, sign extensions may be mapped into specific machine instructions. If these are not present in the processor instruction set, sign extensions are simulated using shift and `and` masking operations. The example below shows the operations necessary to sign-extend 8 bit words to 16 bits:

```
r1 -> shl_o,  24 -> shl_t,     shl_r -> r0;
r0 -> shr_o,  24 -> shr_t,     shr_r -> r0;
r0 -> and_o,  0xFFFF -> and_t, and_r -> r0;
```

   The rules for generating the code above presented are contained in the module for code retargeting. The bit shift trigger values 24 and the mask value 0xFFFF of the `and` operation imply the data word widths.

[1] A Fortran to C preprocessor is also available.

[2] The data type widths of the intermediate code are based on the DLX processor architecture, a model for early RISC architectures.

4. The memory load and store operations support a restricted set of data widths: 8, 16 and 32 bits. As in sign extensions, either a specific instruction is provided, or a group of operations must be introduced for the purpose. The example below shows the code introduced to simulate a byte store operation, in the hypothesis that an insert-byte instruction between registers is present in the instruction set. Similar considerations apply for load instructions.

```
r0 -> ld_t, ld_r -> r127;
r0 -> insb_o, r127 -> insb_o1, r1 -> insb_t, insb_r -> r127;
r0 -> st_o, r127 -> st_t;
```

It is interesting to point out that a load operation must be introduced, which may represent a major source of performance degradation for the program. The insert operation `insb` inserts the first byte of the word specified by the trigger move into the word specified by the second operand, `o1`, at the position specified by the first operand, `o`. If the insert instruction set is not present, the byte load is expanded into 10 operations. As in sign-extensions, some of their operand values imply the word data size.

Not considering the limitations listed above, the MOVE processor generator and the MOVE scheduler could easily handle GPRs of any bit width. However, bus width for integer data is currently fixed to 32 bits[3], therefore the register file widths different from 32 bits may require an additional hardware for sign extension or explicit sign extension operations to be introduced by the scheduler.

8.1.2 Requirements for supporting multiple data widths

To fully support multiple data widths, the MOVE framework must be extensively modified. The list shown below describes the main extensions that must be applied to the current framework, assuming that from the point of view of the software designer all that is necessary to specify is the width of data objects.

1. A formalism must be provided at source code level to independently specify the bit precision of base type objects. It should be possible to specify more than the 4 types mentioned in subsection 8.1.1. Data width information must be passed to the back-end through the sequential MOVE code.

2. The MOVE template syntax should be augmented to include data path width for each FU, RF and bus of the configuration.

3. The back-end should be able to optimize the resource assignment taking into account the bit width required for the operation, looking for the best fit between the operand bit width and the data path width of the available resources[4].

4. Data path widths of FUs and RFs should not be modified by the design space exploration process, for two main reasons: (1) the bit precision is related to the application requirements and is usually pre-defined and (2) the compiler can not always ensure what is the number of bits really required for FU operands or values stored into RFs. Instead, the exploration tool should be able to incrementally remove or add buses taking into account

[3]Buses 1 bit and 64 bit wide are also possible for boolean and floating point registers, respectively.

[4]The bus assignment algorithm of the scheduler already tries to assign a bus of the required width to the transport. If this is not available it looks for wider buses not yet assigned.

all the bus width available from the initial configuration specified by the designer, as described in section 2.2. For example, if 20 and 24 bit buses are defined, the removal of an integer bus should proceed in steps which comprise 24 and 20 bit widths, in addition to the default widths (32 and 1).

These features require extensive changes to the MOVE framework. The limited amount of information that the front-end can provide to the scheduler through the intermediate sequential MOVE code appears as the main obstacle of the implementation. Also the 32 bit machine model used by the front-end for generating RTL code seems to require a radical redesign of the module for MOVE code retargeting. It is important to remember that the changes to the software subsystem should not affect the compilation process until RTL retargeting. This is needed to maintain the capability to adapt any compiler which provides a retargetable intermediate representation to the MOVE framework without changing the early front-end.

8.2 Study of implementation

In this section we present an approach to dealing with part of the requirements listed in subsection 8.1.2. More specifically, we restrict our goal to adding support for multiple FU data path widths to the MOVE framework. The studied approach has been partially implemented by means of C++ classes and operator overloading. Buses and RFs are not considered: we assume a width of 32 bits and automatic sign extension by hardware before transporting the FU result to the bus and possibly to a RF.

8.2.1 Implementation requirements

The purpose of the presented implementation is (1) to provide a formalism for specifying the data objects with width smaller than 32 bits and (2) to assign operations to FUs whose data path width is appropriate. For appropriate we mean equal or higher than the operand widths. The necessary characteristics of this implementation are listed below.

1. **Compatibility.** The formalism to express the base type width must be compatible with C and C++ syntax.
2. **Adaptability.** A program already written should be adaptable to multiple width support with no or minor modifications. The same applies for applications whose precision requirements are changed.
3. **Type conversions.** Implicit type conversion and promotion must be supported between different data widths.
4. **Efficiency.** No overhead must be introduced: the new types should be comparable to built-in types as for code size, performance and possibly assembly code layout.

To express data widths smaller than 32 bits we chose the standard type declaration syntax. A new set of types is defined for widths smaller than 32 bits. In this way we associate with each variable its width, that in turn determines the data path width required from the FUs that perform operations on them. This approach satisfies (1) and (2). The sole adaptation required to programs already written is the re-definition of the type declaration statements with the new types. It is also possible to retain the organization of the C type system into 4 basic integer types while changing the data width associated to each basic type identifier, `char`, `short`, `int`, `long`.

```
class word24 {
   private: int data;
   public:
            word24() {};
            word24(int n) {data=n;};
            operator word20() {return *this;};
            operator int() {return data;};
            operator+(const &word24 w);
            operator++(int);
            operator++();
            operator+=(word24 w);
}
```

Figure 8.1: Example of class declaration for 24 bit word support. Only a restricted set of algebraic operators is shown.

In this case, it is sufficient to re-define the basic types in a header file to be included in every module of the program.

When an algebraic expression involves terms of different type width, implicit conversion of smaller types to the widest one is required. At machine code level, this implies that the operations specified by an expression must be assigned to a FU whose data path width is equal or higher than the widest operand type. An example of how multiple widths are expressed at source level is shown below.

```
word20  a,b;  // 20 bit variables
word24  c;    // 24 bit variable
 ...
c=c*(b+a);
```

The addition should be assigned to a 20 bit FU, the multiplication should be assigned to a 24 bit FU. To specify the data path required for each operation a new extended set of instructions, which cover any bit precision from 1 to 32 bits is provided. An example of parallel MOVE code generated for this expression is presented below. Note that no implicit conversion is necessary: the result of the first addition is simply passed to an operand of the multiply operation. Sign extension is performed before writing the 24 bit result on bus.

```
r1 -> add20_o, r2 -> add20_t;
add20_r -> mul24_o, r3 -> mul24_t;
mul24_r -> r3;
```

8.2.2 Proposed implementation through classes

To implement data types smaller than 32 bits we use C++ classes and operator overloading. Figure 8.1 shows part of a class for supporting 24 bit words. The sole data member of this class is an `int`. This restricts the data widths supported to values smaller or equal than 32 bits. There are several facilities provided by C++ classes which turn to be very favorable for implementing support for widths smaller than 32 bits:

1. Automatic constructors.
2. Implicit conversion.
3. Operator overloading.

The first constructor in figure 8.1 is necessary to allow definitions without initialization, which is a fundamental requirement of basic types. The second method allows to create a `word24` object and initialize it with an `int` value.

Using constructors to specify implicit type conversion from user-defined to basic types is not possible, as the latter are not classes and have no constructors. This problem is coped with by defining a *conversion operator* for the source type. In figure 8.1 conversion operators to `word20` and to `int` are defined. In C++ only one level of user-defined implicit conversion is legal in a type expression. This means that for each `word`*X* class a complete set of conversion operators to any other `word`*Y* type must be specified[5]. Conversion operators are implemented as simple assignments.

Operator functions provide a facility for specifying the operation set of a user-defined type. Both binary and unary algebraic operators can be easily overloaded to mimic conventional algebraic operations on our set of data types smaller than 32 bits. The operator precedence and the expression syntax can not be changed: the same rules as for built-in operator expressions apply. Figure 8.1 shows an example declaration for the binary operator + and the unary operator ++; for the latter both prefix and postfix forms are declared. To complete the algebraic operation set, for each binary operator @ the equivalent assignment operator @= should be provided. Figure 8.1 shows also the definition of the += operator function.

Since we do not want additional operations when generating code, the operator members must consist of a single assembler declaration `asm()`. Such declaration passes to the MOVE assembler the instruction required to perform the operation. In this way we bypass the RTL retargeting step and directly specify the bit precision required for the operation. An example of operator member implementation is given below for the += statement.

```
word24& operator+=(word24 a) {
     asm( "%1 -> add24_o; %2 -> add24_t; add24_r -> %0":
          "=r" (data) : "ri" (data), "ri" (a.data));
     return *this; };
```

The argument of the `asm()` statement is not defined by the C++ language, and it depends on the compiler and the assembler implementations. It specifies a machine instruction pattern; the code generator recognizes operand and result patterns in the form `%`*n* and performs the register matching: move sources and destination are assigned to the registers which contain the current values of the operator parameters. For example, the `a+=b` expression, holding the hypothesis that `a` and `b` are stored in registers `r1` and `r2` respectively, generates the following sequential MOVE instruction[6]:

```
r1 -> add24_o, r2 -> add24_t, add24_r -> r1
```

Note that a local `int` variable has to be defined when the operator creates a new object to return its value. Such variable is assigned to a register throughout its (short) extent, and it does not cause allocation overhead.

8.3 Implementation test

The first part of this section describes the results obtained from the implementation test of a 24 bit data type. This type is based on class `word24`, as described in section 8.2, and it has been

[5]If multiple conversions steps were possible, implicit conversion to `int` and class construction would be sufficient to enable conversion between any couple of `word`*X*, `word`*Y* types. Note that no information is lost in the process.

[6]The hypothesis that the += operator is inlined must hold, too.

```
void main() {
   word24 value;
   word24 total;

   for (value=0; value<5; value++)
   {
        printf ("%d\t", total);
        total = total + value;
   }
   printf ("total: %d, value %d\n", total, value);
}
```

Figure 8.2: Program to test `word24` data type implementation.

tested by means of several short programs. A small example, which performs a sum loop using 24 bit variables as loop index and sum accumulator is here considered. The generated MOVE code shows a high performance degradation with respect to an equivalent program using built-in `int` type instead of `word24` for the variables. The rest of this section analyzes in detail the causes of inefficiencies.

8.3.1 Test results

Figure 8.2 shows one of the program designed to test our implementation of 24 bit data types. This program uses overloaded operators + and postfix ++ to compute a sum[7]. The resulting sequential MOVE code consists of 46 instructions and requires 150 operations. These figures roughly correspond to size and operation count ratios of 3 and 3.5 respectively in comparison with the reference program using `int` built-in type. More specifically, 106 additional operations are executed. These represent therefore *class overhead*, i.e. additional instructions due to the class construct which are not present for built-in types. The 4 causes of class overhead and their contribution are shown below:

1. 28 operations are stack pointer offset additions. They are necessary to evaluate the addresses of `word24` data members, which are local variables allocated in dynamic stacks.
2. 33 load operations. They are necessary to read the current values of `total` and `value` from memory.
3. 25 store operations. They are necessary to copy back into memory the values of `total` and `value`.
4. 20 operations which represent function call overhead. These are due to the operator function +=, that is not inlined, and consist of argument passing copy moves, call move and return jump.

These results suggest that implementations of multiple data width support based on C++ classes and operator overloading, although possible, are not acceptable because of the high overhead they produce.

[7]Note that the operator $<$ used in the loop check has not been overloaded, therefore the comparison is performed using the standard instruction of the MOVE framework, `gt`. The conversion from `word24` to `int` for all the operators not overloaded is performed by the operator `int()`.

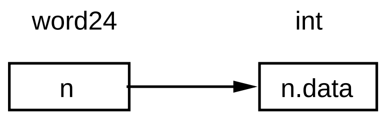


Figure 8.3: Memory layout of a `word24` variable `n`.

8.3.2 Causes of inefficiency

There are two fundamental reasons explaining why the code generated by gcc for C++ classes and their operator functions results so inefficient with respect to the code for expression using built-in types:

1. The memory layout of class data members establishes a level of indirection.
2. Function members which return class objects can not be inlined.

These two causes explain all the overhead found in the test program of subsection 8.3.1 and are described in detail below.

Memory layout of class object

All classes are supersets of C `struct`s, which consist of a pointer addressing a data area. Each member of a `struct` is accessed through this pointer and a specific offset, statically determined by the type size of every `struct` member. The implementation of our `word24` class consists of a pointer to the data member `data`, as shown in figure 8.3. When a `word24` object is passed, a pointer to its data member is really passed. The consequences on code generation and are listed below:

1. Load and store operations are introduced.
2. Address computations are introduced for dynamically allocated objects.
3. Expression optimizations are disabled.

Unlike built-in type variables, gcc can not allocate temporary class objects in GPRs throughout their extent. Class constructor always reserve memory locations for data members. This is a cause of memory accesses not present for built-in types. The test program described in subsection 8.3.1 shows that a realistic memory disambiguation algorithm, as the one provided by the MOVE back-end, can not get rid of a consistent part of these unnecessary memory accesses. Each time a memory write access can not be resolved from the data member of a `word24`, the subsequently access of that `word24` variable requires a load from memory, although its value might be also available in a register. Similarly, each time a `word24` value is changed and subsequent, independent memory accesses can not be resolved from its address, such value must be written back to memory.

Although the memory addresses of `word24` objects can be maintained in GPRs, the test program described in subsection 8.3.1 shows that a large part of addresses are recomputed each time, even after the optimizations performed by the back-end. For automatic `word24` objects, which are dynamically allocated, memory addresses must be computed adding their offset to the stack pointer. Static and global `word24` variables do not require any address computation instead, as their addresses are available at compile time as immediate values. However, immediate values representing addresses are likely to take extra instruction bandwidth, as described in subsection 6.2.2, thus degrading performances.

Since `word24` values are stored in memory, data flow analysis is ineffective: the compiler can not make assumptions on the values stored in memory. This disables algebraic optimizations that rely on data flow analysis, which are commonly applied to expressions whose terms are maintained in GPRs, like *algebraic simplification* and *reduction in strength* [1].

Inlining limitations

As explained in subsection 6.4.2, gcc does not inline function members returning class objects. Operator functions are not exceptions to this rule. The test performed in subsection 8.3.1 shows that, at least 4 additional operations are introduced for each algebraic operation performed via a non-inlined operator. These consist of (1) a call move, (2) a return jump, (3) a copy move for passing the result, i.e. the address of the modified `word24`, and (4) the address computation of the result address. Moreover, function calls represent a barrier that limits the scheduling scope of the MOVE scheduler [10]. It is important to point out that the scheduling step can significantly improve the code by inlining operator functions and partially removing unnecessary address computations and memory accesses.

8.4 Conclusions

Section 8.2 demonstrated that C++ classes can not provide an efficient solution. A valid alternative implementation could be the use of annotated built-in types. Such solution satisfies the compatibility and adaptability requirements related in subsection 8.2.1. The data width should not affect the code generated by the front-end; instead, the width of variables should be passed to the back-end through annotations in the sequential MOVE code. The back-end is charged with *propagating* data width information from the definition statement throughout the code. Each move source or destination register should be provided with a tag which indicates the width of the variable it carries. The required operation width, which specifies the minimum FU data path width required, is determined by the operand widths. The operation width is propagated to subsequent operations through the FU result register. Conversion between types is performed during information propagation.

Using built-in types avoids all the overhead causes found in section 8.2: stated simply, the front-end treats the annotated types exactly as normal types. However, it is not evident that the back-end is able to associate each width annotation to its variable. Register allocation throughout the variable extent and instruction reordering make it difficult to establish which value the annotation refers to. Further research should be done to clarify this problem.

Another problem of annotated built-in types is that some operations depend on the architecture data widths: sign extensions and load/store of sub-words. Such operations must be adapted by the back-end to the defined word widths. In the worst case, i.e. when the instruction set does not directly support these operations, rather complex data flow analysis is necessary to detect the instruction sequences corresponding to a sign extension or a sub-word load/store, described in subsection 8.1.1. It may be favorable to retarget RTL to a full instruction set MOVE configuration, and to perform a code substitution during the sequential code pre-processing. In this way we make the detection problem straightforward and shift more 'responsibility' for hardware-dependent code generation to the back-end.

Conclusions 9

This chapter concludes this thesis. Section 9.1 provides a summary of the work and the results achieved. Section 9.2 describes possible future research.

9.1 Summary

The main contributions of this thesis are summarized below:

1. A detailed, quantitative comparison between an ASP application developed with custom methodology and its synthesized equivalent has been completed. This comparison pointed out a largely expected performance gap in terms of code size and execution time.

2. The causes of such performance gap were investigated. We found several sources of inefficiency in the MOVE code and several limitations of the MOVE architecture template. We estimated the effect for each of them on the code size and the cycle count and proposed possible approaches for implementing it in the MOVE framework.

3. One of the proposed extensions to the architecture template, the introduction of variable data word sizes, was studied in detail. An implementation approach, based on C++ classes and operator overloading, was attempted without success. However, this attempt gave the possibility to better formulate the requirements of an efficient implementation.

4. One of the features proposed for reducing code inefficiency, namely hardware supported execution of loops with fixed trip count, has been implemented, with important limitations, in the MOVE software subsystem.

The analysis of the causes of inefficiency showed that the MOVE architecture template and the intermediate MOVE code generated by the compiler front-end are not enough flexible. The hardware configuration should be tailored around the specific application. The most significant recommendations to approach this goal are listed below:

1. Data types smaller than 32 bits are necessary for embedded system applications. Bit precision must be a design choice left to the designer.

2. Auto-incremented pointers are extremely advantageous for applications that require vector regular access patterns, especially for TTA configurations with limited resources. The wrap-around feature modulo the vector size gives a substantial contribution to the performance of the custom decoder.

3. Hardware loops can give an important contribution to performance without affecting code size. Loop unrolling and software pipeline can be applied to hardware loops with 'almost' additive effect.

9.2 Future work

The work described in this thesis can be extended in several directions.

1. Implementation of each recommendation proposed in chapter 6 and measurement of the performance gain achieved on a large set of benchmarks.
2. Improvement of the hardware loop implementation described in chapter 7. The test results of this preliminary implementation are encouraging and confirm the estimated performance gain that can be achieved by extending the algorithm to loops with pre-definable but non-constant trip count. Estimation of the number of loops that can be mapped in a large set of benchmarks.
3. Extension of data flow information used for hardware loop detection to other loop enhancement techniques currently implemented in the MOVE framework: loop unrolling and software pipelining.

The latter point is particularly interesting. The data flow analysis performed for detecting hardware loops with fixed iteration number can be re-used to implement a more efficient loop unrolling algorithm. The current algorithm converts backward edges of duplicated loop bodies into exit edges and maintains the loop index increment and compare operations. If the number of iterations executed is known at compile time, it is possible to avoid multiple index increments and exit branches within the unrolled loop body. The scheduler must ensure that the trip count of the loop is a multiple of the unrolling factor. Loop peeling can be applied to achieve a number of iterations multiple of the unrolling factor; in this way, a loop condition check at the loop end is sufficient.

The current software pipeline algorithm implemented in the MOVE scheduler rejects loops with multiple backward or exit edges. By eliminating conditional exit branches from unrolled loops, it is possible to apply unrolling to loops scheduled with software pipeline. This can give a significant performance improvement. In fact, when the minimum initiation interval (MII), the part of the loop executed during the steady state of the software pipeline, is a fractional number, the rounding up can cause a high performance degradation [10]. For example, if MII is 4/3, an iteration requires 2 cycles. By unrolling the loop 3 times MII becomes 4, that is, 4 cycles are required for three iterations.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesly Series in Computer Science. Addison-Wesly Publishing Company, Reading, Massachusetts, 1985.

[2] R. Brayton and R. Spence. *Sensitivity and Optimization*. Elsevier, 1980.

[3] Raul Camposano and Wayne Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[4] Hendrik Corporaal. *Transport Triggered Architectures – Design and Evaluation*. PhD thesis, Technical University of Delft, August 1995.

[5] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[6] Henk Corporaal and Jan Hoogerbrugge. Code generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*, pages 240–259. Kluwer Academic Publishers, 1995. Chapter 14.

[7] Henk Corporaal and Jan Hoogerbrugge. Cosynthesis with the MOVE framework. In *CESA 96*, 1996.

[8] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[9] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-level Synthesis; introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[10] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Technical University of Delft, February 1996.

[11] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *MICRO-27*, Santa Clara, December 1994.

[12] Anil K. Jain. *Foundamentals of digital image processing*. Prentice-Hall, 1989.

[13] Joint Technical Committee ISO/IEC JTC1. ISO/IEC 11172-1 "Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s" – Part 1: System. International Standard 1, ISO/IEC, August 1993.

[14] Joint Technical Committee ISO/IEC JTC1. ISO/IEC 11172-2 "Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s" – Part 2: Video. International Standard 2, ISO/IEC, August 1993.

[15] Joint Technical Committee ISO/IEC JTC1. ISO/IEC 11172-3 "Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s" – Part 3: Audio. International Standard 3, ISO/IEC, August 1993.

[16] Joint Technical Committee ISO/IEC JTC1. ISO/IEC 11172-4 "Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s" – Part 4: Conformance. International Standard 4, ISO/IEC, 1994.

[17] Joint Technical Committee ISO/IEC JTC1. ISO/IEC 11172-5 "Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s" – Part 5: Software Simulation. International Standard 5, ISO/IEC, 1995.

[18] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818-1 "Information technology – Generic coding of moving pictures and associated audio: Systems. International Standard 1, ISO/IEC, June 1995.

[19] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818-2 "Information technology – Generic coding of moving pictures and associated audio: Video. International Standard 2, ISO/IEC, June 1995.

[20] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818-3 "Information technology – Generic coding of moving pictures and associated audio: Audio. International Standard 3, ISO/IEC, June 1995.

[21] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818-4 "Information technology – Generic coding of moving pictures and associated audio: Conformance. International Standard 4, ISO/IEC, 1995.

[22] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818-5 "Information technology – Generic coding of moving pictures and associated audio: Software. International Standard 5, ISO/IEC, 1995.

[23] Joint Technical Committee ISO/IEC JTC1/SC29/WG11. ISO/IEC 13818-7 "Information technology – Generic coding of moving pictures and associated audio: Non-Backwards Compatible Audio. Draft International Standard 7, ISO/IEC, 1995.

[24] Brian W. Kernighan and Dennis M. Ritchie. *Linguaggio C*. Jakson, 2nd edition, 1989. The C Programming Language, Prentice-Hall, 1989.

[25] Reinoud Lamberts. A MOVE Processor Generator. Technical report, Delft Univ. of Technology, November 1993.

[26] Jae S. Lim. *Two dimensional Signal and image processing*. Prentice-Hall, 1990.

[27] Petra Michel, Ulrich Lauther, and Peter Duzy, editors. *The synthesis approach to digital system design*. Kluwer Academic Publishers, 1992.

[28] Davis Yen Pam. digital audio compression. *Digital Technical Journal*, 5(2):28–40, 1993.

[29] Ken C. Pohlmann. *Principles of digital audio*. McGraw-Hill, 3rd edition, 1995.

[30] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, September 1994. version 2.6.

[31] Bjarne Stroustrup. *The C++ programming language*. Addison Wesley, 2nd edition, June 1994.

[32] Steven A. Tretter. *Introduction to discrete-time signal processing*. John Wiley and Sons, 1976.

[33] Jan Vanhoof et al. *High-Level Synthesis for Real-Time Digital Signal Processing*. Kluwer academic publishers, 1993.

[34] Robert A. Walker and Raul Camposano, editors. *A survey of high-level synthesis systems*. Kluwer academic publishers, 1991.