TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

JARI HEIKKINEN

DSP APPLICATIONS ON TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Subject approved by Department Council
21st March, 2001

Examiners: Prof. Jarmo Takala
Prof. Markku Kivikoski

Preface

The work for this thesis was carried out in Institute of Digital and Computer Systems of Tampere University of Technology in 2000-2001 as a part of the Electronics System Design (ELSU) project funded by National Technology Association and Nokia Mobile Phones.

I would like to express my sincere gratitude to my thesis supervisor Professor Jarmo Takala for his guidance and valuable tips for the thesis. I would also like to thank Professor Henk Corporaal for giving me a possibility to work under his guidance in MOVE project in Delft University of Technology in the Netherlands during summer 2000. I am particularly grateful to Andrea Cilio for always having time and patience to answer all the questions I presented to him.

I would also like to thank the "coffee room gang" for giving valuable, relaxing moments during working days.

I wish to thank my family for their support to my studies. Most of all, I want to thank my lovely Nina for her love and support.

Tampere, May 25, 2001

Jari Heikkinen

Table of Contents

Abstract

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Electrical Engineering
Institute of Digital and Computer Systems
Heikkinen, Jari Mikko: DSP Applications on Transport Triggered Architectures
Master of Science thesis: 71 pages, 8 appendix pages
Examiners: Prof. Jarmo Takala and Prof. Markku Kivikoski
Funding: The National Technology Agency, Nokia Mobile Phones
June 2000
Keywords: transport triggered architecture, fast Fourier transform, discrete cosine transform, performance evaluation

Application-specific instruction set processors can be designed to respond to performance requirements of digital signal processing applications by still maintaining the programmability. The main advantage of such a processors is their area-efficiency, i.e., hardware is tailored for the given application. A public domain tool set, MOVE framework, was designed to overcome the long time-to-market, high non-recurrent costs, high design risk and poor programmability of application-specific instruction set processors by generating processors automatically. The MOVE framework consists of three components. Design space explorer searches the processor design space and finds the best possible architecture configuration for the given application. Hardware subsystem generates the layout of the processor design. Software subsystem compiles high-level language applications to binary executables by exploiting instruction-level parallelism. Processors designed with MOVE framework utilize the paradigm of transport triggered architectures (TTA). In TTA, data transports between function units and register files are programmed explicitly instead of programming operations. Operations occur as side effect of these explicit transports.

In this thesis, the performance of MOVE architecture was evaluated against a commercial DSP processor, Texas Instruments TMS320C62x DSP. Benchmark applications used were fast Fourier transform and discrete cosine transform applications. The results showed that MOVE architecture performed better general-purpose applications without much concern on the coding style. In thoroughly programmed DCT applications, the compiler of C62x was able to exploit optimizing techniques, such as software pipelining, designed for DSP applications and thus resulting in faster execution of applications. MOVE tools miss the optimizations for DSP. The code density of MOVE architecture was poor in all applications due to explicit specification of data transports.

No serious defects in the MOVE-architecture were found. The MOVE compiler was found insensitive to changes in coding styles. It was also found that the compiler is not optimized for DSP applications. With the modifications proposed to the compiler, comparable performance and even better code density can be obtained. As a conclusion, based on the results obtained in this thesis work, it can be stated that MOVE framework is a promising design environment for designing cost-critical embedded system products used in DSP-applications.

Tiivistelmä

TAMPEREEN TEKNILLINEN KORKEAKOULU
Sähkötekniikan osasto
Digitaali- ja tietokonetekniikan laitos
Heikkinen, Jari Mikko: DSP Applications on Transport Triggered Architecture
Diplomityö: 71 sivua, 8 liitesivua
Tarkastajat: Prof. Jarmo Takala ja Prof. Markku Kivikoski
Rahoitus: Teknologian kehittämiskeskus (Tekes), Nokia Mobile Phones
Kesäkuu 2001
Avainsanat: transport triggered architecture, nopea Fourier-muunnos, diskreetti kosinimuunnos, suorituskyky arviointi

Sovelluskohtaisia käskykantaprosessoreita voidaan suunnitella vastaamaan digitaalisen signaalinkäsittelyn vaatimuksiin säilyttäen kuitenkin ohjelmoitavuus. Tällaisten prosessorien suurin etu on tehokas pinta-alan käyttö, koska laitteistoresurssit on suunniteltu annettua sovellusta varten. Sovelluskohtaisten käskykantaprosessorien automaattiseen suunnitteluun on kehitetty julkinen työkalupaketti, MOVE framework. MOVE framework koostuu kolmesta osasta. Design space explorer etsii annetulle sovellukselle parhaan mahdollisen prosessoriarkkitehtuurin. Hardware subsystem generoi prosessorin layout-tason kuvaus. Software subsystem kääntää korkean tason kuvauskielellä toteutettuja sovelluksia ajettaviksi binääriohjelmiksi käyttäen käskytason rinnakkaisuutta hyväkseen. Automaattisen suunnittelun etuina ovat nopea markkinoilletuloaika, pienemmät ei-toistuvat suunnittelukustannukset ja pienempi suunnitteluriski. MOVE framework:llä suunnitellut prosessorit käyttävät transport triggered architecture (TTA)-ohjelmointimallia, jossa operaatioiden sijasta ohjelmoidaan datan siirrot toiminnallisten yksiköiden ja rekistereiden välillä. Nämä datan siirrot liipaisevat operaation suorituksen päinvastoin kuin perinteisissä ohjelmointimalleissa.

Tässä diplomityössä MOVE-arkkitehtuurin suorituskykyä verrattiin kaupalliseen, Texas Instrumentsin TMS320C62x DSP-prosessoriin. Testisovelluksina käytettiin nopeaa Fourier-muunnosta ja diskreettiä kosinimuunnosta. Tulosten perusteella MOVE arkkitehtuuri suoriutuu hyvin yleiskäyttöisistä sovelluksista, joissa koodin kirjoittamistyyliin ei ole kiinnitetty suurta huomiota. Huolellisesti ohjelmoiduissa digitaalisen signaalikäsittelyn sovelluksissa C62x:n kääntäjä pystyi käyttämään hyödyksi optimointimenetelmiä mahdollistaen nopeamman suorituksen. MOVE-työkaluissa näitä optimointimahdollisuuksia ei ole käytössä. MOVE-arkkitehtuurin koodintiheys oli kaikissa sovelluksissa huono verrattuna C62x:n koodintiheyteen johtuen datan siirtojen ohjelmoimisesta operaatioiden sijaan. Käyttämällä edistyneitä koodinpakkausmenetelmiä MOVE-arkkitehtuurin koodintiheyttä voidaan huomattavasti parantaa.

Suuria puutteita MOVE-arkkitehtuurissa ei löydetty, vaikka työkalut eivät ole kaupallisia. MOVE kääntäjän tehokkuuden todettiin olevan jokseenkin riippumaton käytetystä koodaustyylistä. Todettiin myös, ettei MOVE kääntäjää ole optimoitu digitaalisen signaalikäsittelyn tehtäviin. Ehdotetuilla parannuksilla voidaan kuitenkin saavuttaa C62x:ään verrattavissa oleva suorituskyky ja jopa parempi koodintiheys. Yhteenvetona voidaan todeta, että tässä diplomityössä saavutettujen tulosten perusteella MOVE framework on lupaava suunnitteluympäristö kustannuskriittisten, digitaalisen signaalinkäsittelyn tehtävissä käytettävien sulautettujen järjestelmien suunnitteluun.

List of Abbreviations and Symbols

α	Constant reflecting the importance of cost
β	Constant reflecting the importance of performance
1-D	One-Dimensional
2-D	Two-Dimensional
ALU	Arithmetic-Logical Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
C62x	TMS320C62x
CHDL	A combination of C and VHDL code
DC	Decode stage
DCT	Discrete Cosine Transform
DCT-SFU	Discrete Cosine Transform Special Function Unit
DSP	Digital Signal Processor or Digital Signal Processing
EX	Execute stage
FFT	Fast Fourier Transform
FU	Function Unit or Functional Unit
GNU	Gnu’s Not Unix
GPP	General-Purpose Processor
HLL	High-Level Language
ID	Identifier
IF	Instruction Fetch stage
IFU	Instruction Fetch Unit
ILP	Instruction-Level Parallelism
LSU	Load-Store Unit
MOVE	Design environment for embedded systems developed at Delft University of Technology
MPG	MOVE Processor Generator
MV	Move stage
OTA	Operation Triggered Architecture
PE	Processing Element

RF	Register File
RISC	Reduced Instruction Set Computer
RU	Register Unit
SFU	Special Function Unit
SIMO	Single Instruction Multiple Operations
SIMT	Single Instruction Multiple Transports
SVTL	Semi Virtual Time Latching
TI	Texas Instruments
TTA	Transport Triggered Architecture
TVTL	True Virtual Time Latching
VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VTL	Virtual Time Latching

1. Introduction

The advances in integrated circuit technology has enabled digital signal processing (DSP) applications to be realized on programmable processors rather than fixed application-specific architectures. DSP still sets high performance requirements for the architectures. For example, for real-time video coding the processor must have enough performance to fulfill the real-time requirements of the application. To respond to performance demands of DSP applications, application-specific instruction set processors (ASIP) can be designed. ASIP offers a specialized architecture for the requirements of the application still maintaining the programmability. In addition to designing an ASIP, existing general purpose processors (GPP) designed to execute many different applications with reasonable performance, or digital signal processors (DSP), whose hardware is tuned for signal processing applications can be used. These processors provide cost-effective solutions for applications with typical computational requirements. However, applications with special requirements often call for ASIPs.

The advantages of ASIPs are their tailored hardware for the demands of applications and a good performance. The problems are long time-to-market, high non-recurrent engineering costs, high design risk and poor programmability. To overcome these problems, a tool set, MOVE framework, was developed to automatically design ASIPs. Processors designed with MOVE framework utilize the paradigm of transport triggered architecture (TTA) proposed by Corporaal [1]. The MOVE framework consists of three components. Design space explorer searches the processor design space and finds the best possible architecture configuration for the given application. Hardware subsystem generates the layout of the processor. Software subsystem compiles high-level language applications to binary executables by exploiting instruction-level parallelism.

Two algorithms, fast Fourier transform (FFT) and discrete cosine transform (DCT) are often used in DSP applications due to their regularity and easy implementation. This thesis describes the performance simulations performed for these two DSP applications to evaluate the applicability of MOVE architectures, processors designed with MOVE framework, in DSP. The performance of MOVE architectures is measured against TMS320C62x DSP of Texas Instruments (TI). In addition, implementation of a special function unit, containing hardware designed for fast execution of DCT, is discussed.

The structure of this thesis is as follows. Chapter 2 describes the development from VLIW architecture to TTA, hardware and software aspects of TTA and gives an overview of the MOVE framework tool set developed to design ASIPs realized with TTAs and to generate object code for the designed processors. Chapter 3 introduces more thoroughly the components of the MOVE framework. First, the design space explorer responsible for finding a best architecture configuration for a given application is described. In addition, software subsystem responsible for generating instruction level parallel object code for the processor designs and hardware subsystem responsible for generating the layout of a processor design are discussed. The performance evaluations performed for DSP applications, namely FFT and DCT, are described in Chapter 4. In addition, analysis of the results is given to explain the results obtained. Chapter 5 describes the design of a special function unit for DCT. Necessary changes in the hardware and software subsystems are given first. Second the design of special function unit for DCT are discussed. Last, the implementation and verification of a processor design including the special function unit for DCT is discussed. Finally, Chapter 6 states the conclusions of this thesis summarizing the most essential issues and experiences.

2. Transport Triggered Architectures

Transport triggered architectures proposed by Corporaal [1] form a superclass of traditional very long instruction word (VLIW) architectures by exploiting in addition to operational style parallelism the parallelism available at the data transport level. VLIWs can be specified as SIMO, single instruction multiple operation, whereas TTAs are of type SIMT, single instruction multiple transports.

Section 2.1 introduces the development of the TTA concept from VLIW architecture. Section 2.2 describes the hardware aspects of TTA and section 2.3 the software aspects of TTA. Section 2.4 introduces the MOVE framework generated to design ASIPs realized with TTA.

2.1 Development from VLIW to TTA

VLIW architectures have been used to design ASIPs due to their scalability, i.e., more functional units (FU) can be added, and flexibility, i.e., the operation of an FU can be almost anything [1]. The principal organization of VLIW illustrated in Figure 1 consists of parallel FUs performing RISC style operations. VLIW contains also a multi-ported register file, which is shared by the FUs [2], [3]. Additional instruction fetch and instruction decode units fetch the instructions and decode them concurrently with the operations of the FUs. The transport network for mutual FU traffic and data transport to the register files must be specified for worst case situation even though this kind of situations rarely emerge. Design for worst case situation in two-operand instructions requires that each FU

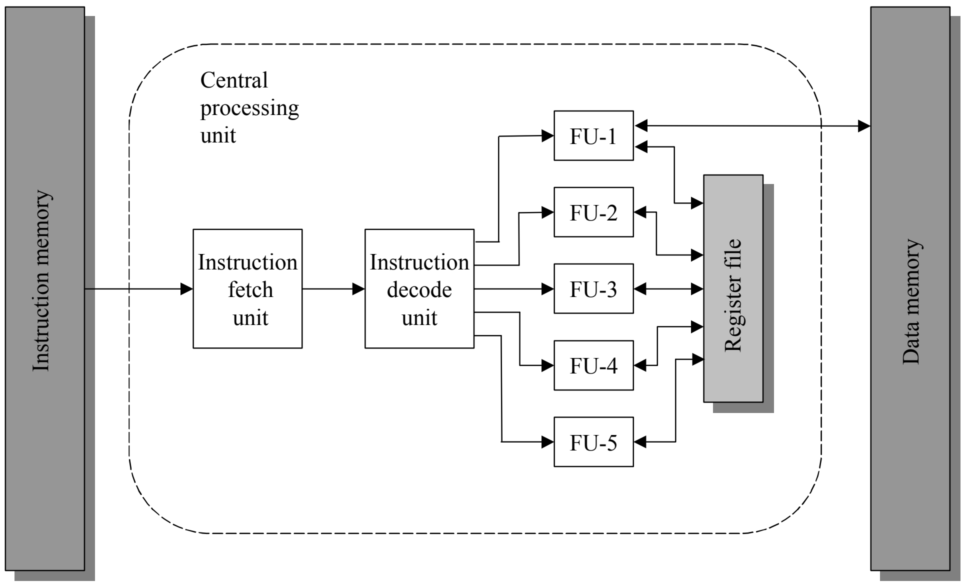


Figure 1. General organization of VLIW architecture.

needs 3 ports for the register file to perform two reads and one write simultaneously [4]. A bypass circuit between the FUs for forwarding results directly from the outputs of FUs to the inputs of FUs, and thus bypassing the register file, must also exist to keep FUs busy. Due to this, improving the performance of VLIW processor design by adding more FUs would cause problems when the number of FUs increases; the complexity of the data path, especially the register file and the bypass circuit would increase rapidly. Clustered VLIW, i.e., a partitioned register file, depicted in Figure 2 and described in [5], and a two-level hierarchical register file for VLIW, described in [6], are two approaches to overcome the high register read/write port requirements. Clustered VLIW is realized, e.g., in TMS320C62X family DSP processors from Texas Instruments.

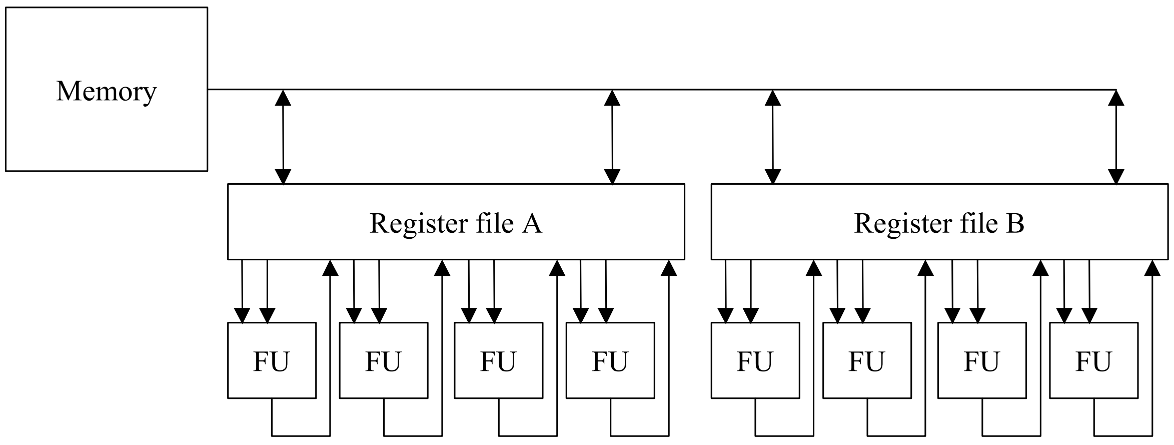


Figure 2. Structure of clustered VLIW.

TTA architecture was developed to reduce more effectively the data path complexity and the underutilization of the register file and the bypass circuit. In VLIW, there are several reasons when all the three ports in FU to the register file are not needed:

All operations do not require two operands, e.g., register-to-register copies, immediate operations, jumps, and call. Furthermore, all operations do not produce a result for the result register, e.g., jumps and calls. Values can also be bypassed directly between FUs meaning that they do not have to be read from the register file. If all usages of a result value can be bypassed, the value is not needed to be written to the register file and is thus said to be dead. An operand value may also be used multiple times by succeeding operations meaning that the value is needed to be read only once from the register file. In addition, register file ports may be shared by multiple reads. This happens when multiple operations read the same register in the same clock cycle.

Exploiting these facts the register file can be implemented as a separate FU, called register unit (RU), having limited number of read and write ports. The utilization of the RU is determined at compile time by the compiler. Thus the complexity of the register file is reduced. [1]

The bypass complexity can be reduced by making the bypass registers visible at architectural level by assigning them to the inputs and outputs of FUs. This way the spilling of bypass values into the RU is made under program control. The bypass complexity can also be reduced by reducing the number of read and write connections and by reducing the number of bypass buses. This requires that, in addition to operations, also transports need to be scheduled at compile-time as is done in the case of RUs. The bypass transports become also visible at the architectural level implying that operations are hidden; data transports trigger the FU operations implicitly [1]. The traditional programming model is mirrored and the concept of transport triggering is introduced.

Figure 3 depicts the principle of TTA organization. A TTA processor consists of a set of function units and register units, or register files (RF), connected by some kind of interconnection network [1]. RUs contain general-purpose registers. In the figure, FU-3 acts as a load/store unit providing an interface to the data memory. TTA does not limit the number of computational units or the number of the number of ports in the units. Also special function units (SFU), having a dedicated operationality, can be added easily.

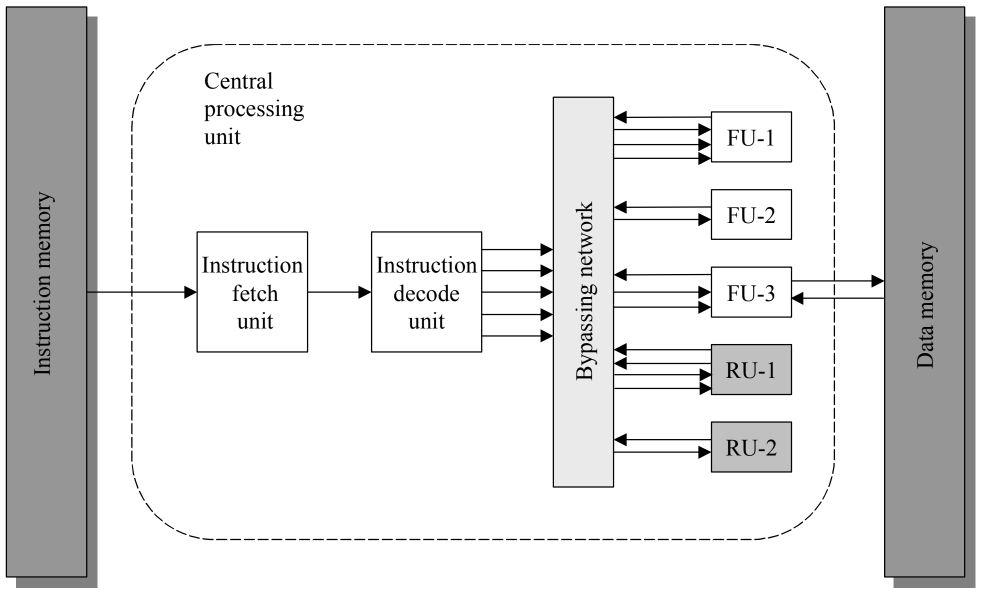


Figure 3. General organization of TTA.

TTAs are constructed of a restricted number of building blocks and are thus modular. The flexibility and scalability properties are also fulfilled. The interconnection network is separated from the FUs and both can be designed independently. FUs can implement any functionality and they can be added to improve the performance without having to change the transport capacity.

2.2 Hardware Aspects

In TTA processor, FUs and RUs are connected by an interconnection network as illustrated in Figure 4. The interconnection network consists of buses and connections to these buses are established through input and output sockets. An input socket contains

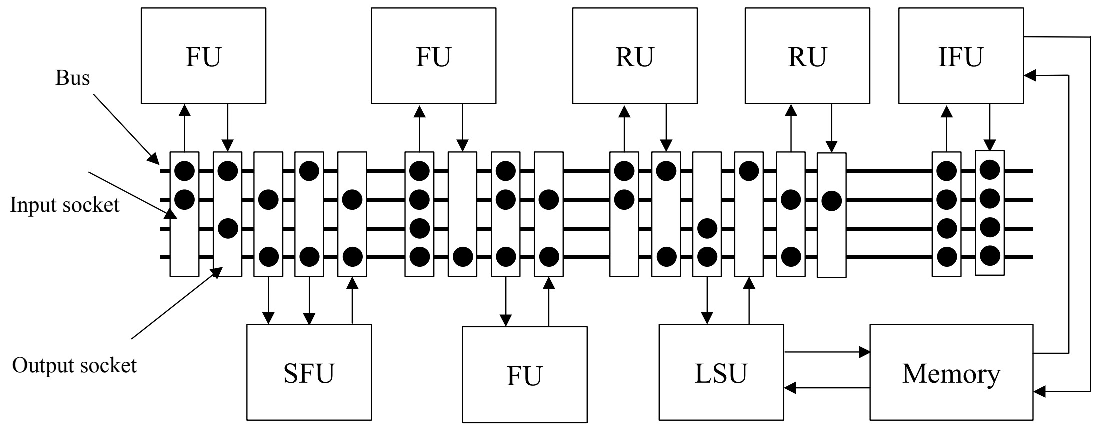


Figure 4. General structure of TTA processor.

multiplexors, which feed data from the buses to the FUs. An output socket contains de-multiplexors setting FU results on the buses. A socket is not necessary connected to each bus. Instruction fetch unit (IFU) is responsible for fetching instructions from the memory and load-store unit (LSU) provides an interface to the memory. TTA may contain also user-defined units, special function units (SFU).

2.2.1 Function Units

In TTA, the function units are responsible for performing operations on data. FUs receive data from the input sockets and when the operation is completed, the result data can be read from the result socket.

Figure 5 illustrates the structure of a function unit. In this case, the function unit has only one trigger socket, one operand socket, and one result socket although there could be more of them. Function units are internally pipelined. In the first stage, the T and O registers hold the data read from the input sockets. Since the data sent through the operand socket is held in a register here, data can be written to the operand sockets in earlier cycles. The operation is triggered when data is written to the trigger socket. The first stage contains also a register for an opcode. Opcodes are used to tell the function unit, which operation to perform. The opcode is extracted from the destination ID used to address the trigger socket of the function unit. [7]

Pipeline stages of a function unit contain combinatorial logic and a pipeline register. All the pipeline stages in a function unit run synchronously to the instruction stream as specified by the compiler. Each time an instruction is issued the FU pipeline processes one step. In case of a global lock, all the pipeline stages lock until the lock signal is cleared. This latching method is known as virtual time latching (VTL) [1]. Two different versions of VTL exist: true virtual time latching (TVTL) and semi virtual time latching (SVTL).

In TVTL, moves to both trigger and operand register start new operations. The hardware becomes this way simplest but it restricts the scheduling interval. Basically, in TVTL all operands must be transported to the function unit in the same clock cycle. In SVTL, only the store to trigger register starts new operations. This version results in more scheduling freedom. A store to operand register can precede the store to trigger register. Due to SVTL also the output of FU changes only when trigger move exists and thus results can stay longer in FUs.

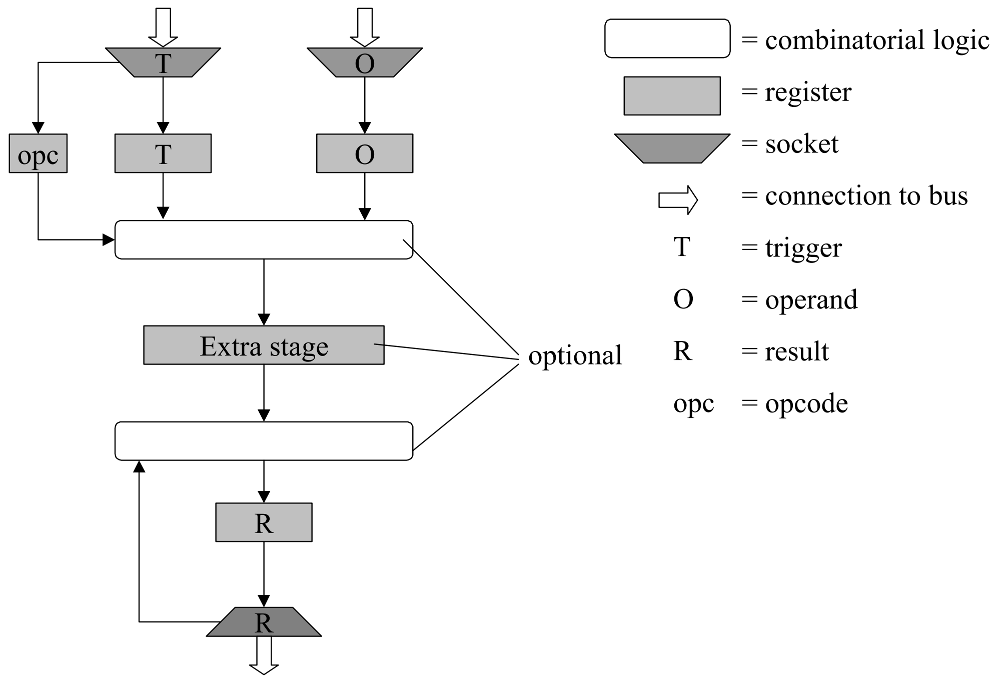


Figure 5. Structure of function unit.

2.2.2 Sockets

Input sockets are used to transport data from the move buses to the FUs or RUs. Result sockets are responsible for transporting data from the FUs or RUs to the move buses. Two kinds of input sockets exist, the trigger socket and the operand socket. The only difference between these two sockets is that the trigger socket, in addition to the data, passes an opcode to the FU to select the type of operation the FU is to perform. Figure 6 depicts the principal organization of an input socket connected to two move buses. Destination IDs are used to select an input socket to pass the data from the buses to the FU by comparing each destination ID to a value identifying the input socket. If they match, the input socket is selected and a corresponding *select* bit is set. Since the destination IDs arrive one clock cycle before the actual data according to the three-stage transport pipeline, illustrated in section 2.2.3, the *select* signal is stored into a register. The opcode used to select an operation the FU is to perform is also extracted from the destination ID and stored into a register. In the next stage, the data is available on the buses. Select signals controls multiplexors to obtain data from the move bus and to pass the correct opcode to the FU. The data sent to the FU can also be sign extended. [7]

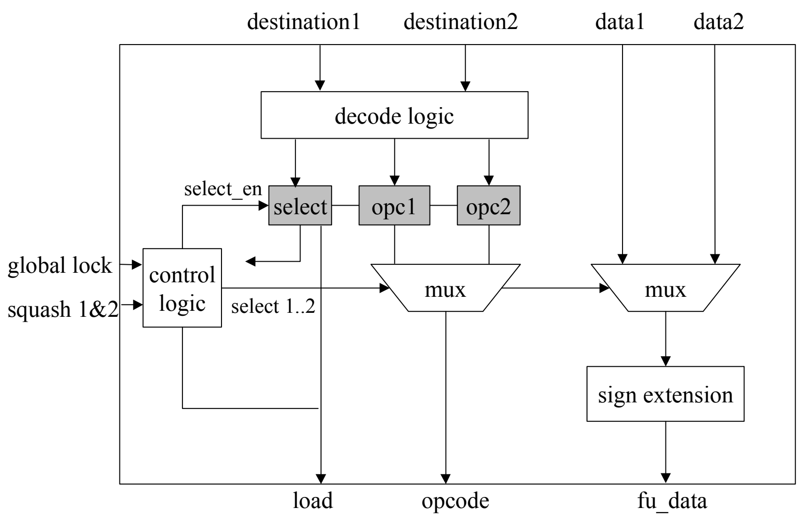


Figure 6. Principal organization of input socket.

Principal organization of a result socket is depicted in Figure 7. As can be seen from the figure, the implementation of a result socket is very similar to that of an input socket Instead of destination IDs, the result socket receives source IDs. These are used to select a result socket for placing result values on a bus. The result socket compares each source ID to a value that identifies the result socket. If they match the result socket is selected and a corresponding *select* bit is set. The result socket will also extract an opcode from the source ID and pass it to the FU in the next pipeline stage. At that stage also the result data of the FU is put on a move bus selected with the demultiplexor. Data put on a move bus can be sign extended.[7]

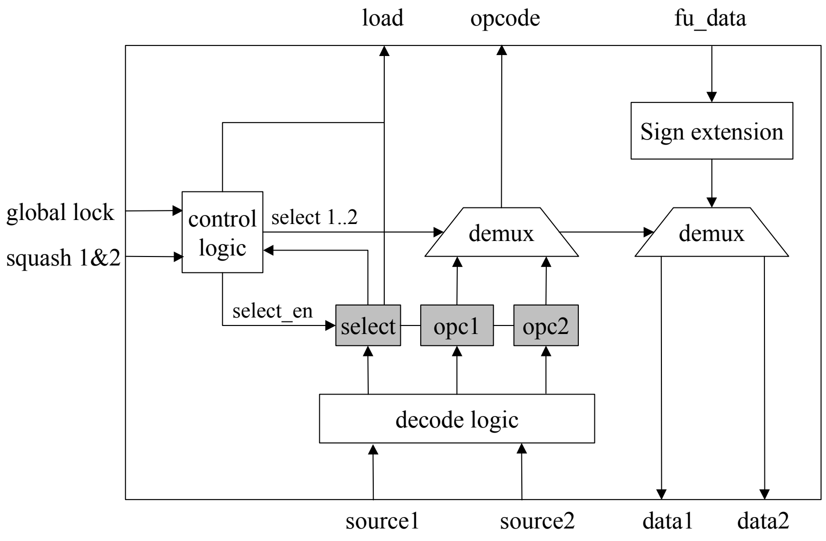


Figure 7. Principal organization of output socket.

2.2.3 Pipelining

TTA supports two levels of pipelining. In addition to transport pipelining, which means that the execution of instructions is pipelined, also function units can be pipelined. Decisions for these two pipelining schemes can be made independently.

There exist two transport pipelining schemes, three-stage and two-stage, from which the three-stage pipeline is typically used. The three-stage pipeline consists of three stages: instruction fetch (IF), decode (DC), and move (MV).

In the IF stage, the instruction fetch unit fetches the next instruction from the cache or the memory. In the DC stage, the source and the destination IDs are transported to, and decoded in the sockets. The input and output sockets will be selected for data transport in the next stage. In MV stage, the actual data transports take place. FUs receive data from the buses and opcodes from the sockets.

The three-stage pipeline is illustrated in Figure 8.a). In the two-stage pipeline, the DC and MV stages are combined into single decode-move-stage (DC-MV) where both the instruction decode and the actual data transport are performed in the same clock cycle.

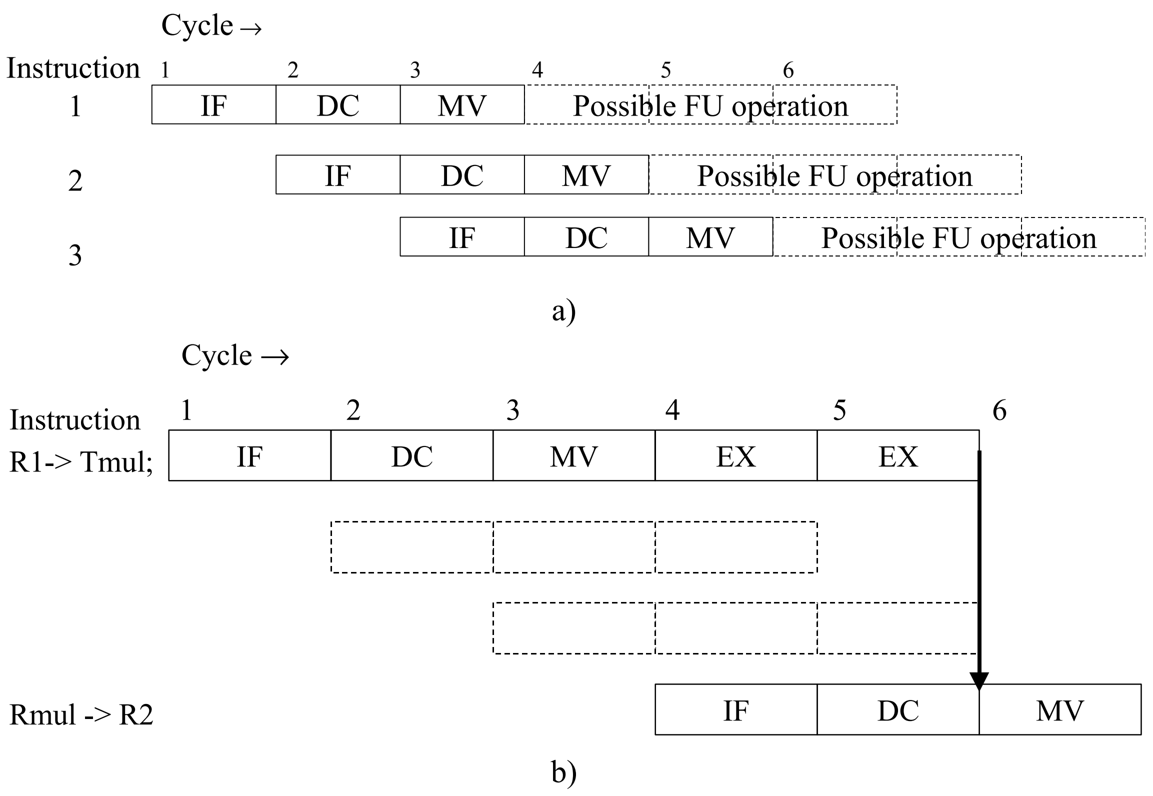


Figure 8. TTA pipeline: a) example of three-stage pipeline and b) example of operation with latency of three.

When function units are pipelined, this will add cycles to the latencies of operations as depicted in Figure 8.b). The first data transport triggers a multiply operation. The multiply FU has two extra pipeline stages, thus the result of the operation is not available in the next cycle after the MV stage but two clock cycles later. In the two extra stages, the multiply operation is performed. These extra stages are called execution (EX) stages. [7]

2.2.4 TTA Instruction Format

The instruction format of TTA processor depicted in Figure 9 contains all the data transports, namely moves. Since one bus can perform a single move per cycle, the instruction contains as many moves as there are buses. The instruction shown in the figure contains two moves and a reserved field for long immediates, which is optional. Each move corresponds to a move bus, e.g., move 1 specifies the data transport performed in move bus 1.

Each move consists of a guard ID (grd), a destination ID and a source ID as shown in the figure. One field of the instruction containing guard, destination and source ID fields is often called a move slot. The guard ID is used by the guard unit of the TTA processor to control the execution or to squash the move operation. The destination ID is used to select a destination socket to where the data is to be sent. The source ID is used to select a result socket as a source from where the data is transported to the destination. Both destination and source IDs contain a socket address and an optional opcode. The socket address is used to select the socket. The opcode is sent in the MV stage to the FU to select the operation of the FU to be performed to the input data. When the opcode is not used, destination and source IDs contain only the socket addresses.

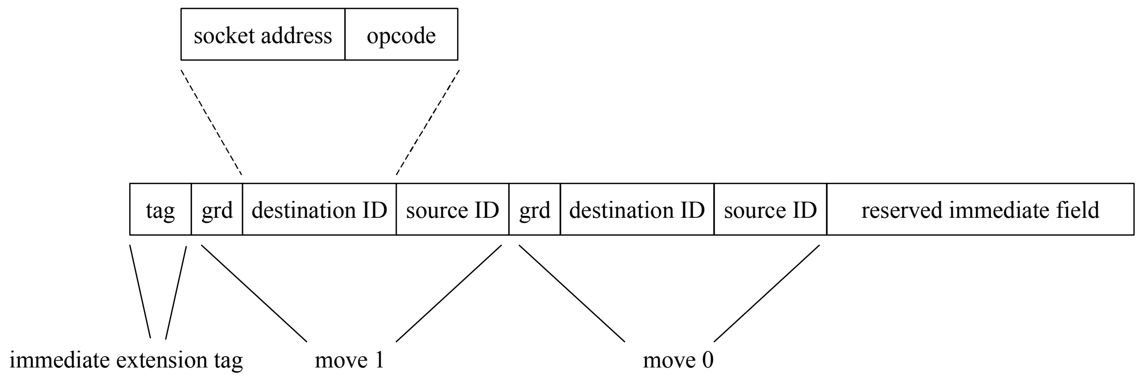


Figure 9. TTA instruction format.

TTA processor supports also immediate extensions. Each move can be used to represent immediate value extensions. These extensions are used to construct long immediate. When a move is used for long immediate extension the sockets are not allowed to decode the information in the move as guard, destination and source IDs. Instead, all the ID fields are used to represent the long immediate. To control when a move is used for long immediate extension a dedicated long immediate extension tag exists at the beginning of the instruction. The tag specifies in each instruction, which of the moves are used for normal data transport and which for long immediate extension. In addition to the moves, there can also exist a dedicated field for long immediate bits. This field is only used to construct a long immediate value. [8]

Apart from long immediates, also short immediate can be constructed. In a case of a short immediate, the value of the source opcode is used to transport the short immediate bits. The length of a short immediate is restricted to the size of the opcode field of the source ID. [7]

2.3 Software Aspects

The main difference between TTAs and traditional operation triggered architectures (OTA), as VLIW, lies in the way they are programmed. For OTAs the program specifies the operations the processor is to perform, e.g., add, sub, or mul. In TTA, these operations occur as side effects of explicitly programmed data transports. A TTA program specifies the data transports to be performed by the interconnection network [1]. Only one type of operation is supported: the move operation, which performs a data transport from source to destination. Due to this, TTA is also called MOVE architecture.

Programming an operation on TTA processor consist of moving operands to the input registers of an FU that is able to perform the operation, and moving the result from the output of the FU to another FU or to RF after the FU has performed the operation. These moves are classified into three move classes: operand, trigger, and result moves.

Operand move is responsible for transporting input data of an operation to the operand register of an FU. One FU can have several operand registers but typically only one operand register exists. Trigger move transports input data to the FU, namely to the trigger register of an FU. The trigger move starts the operation in the FU. Result move is responsible for transporting the result values from the output of an FU to the input of another FU or to a RF.

Operand moves must precede or be performed in the same clock cycle as the trigger move in order to take the input data of the operand move into account when executing the operation. In addition, the trigger move must precede the result move in order to transport the correct result data from the FU to another destination. Typically one RISC operation correspond to three move operations. The correspondence between RISC operations and move operations is presented below.

RISC-type add operation adds together value from register locations r1 and r2 and stores the result to register location r3.

```
add  r3, r1, r2
```

This operation corresponds to three move operations:

```
r1 -> Oadd;
r2 -> Tadd;
Radd -> r3;
```

The first move transports the data from the register location r1 to the operand register of FU performing the add operation. The second move transports the data from the register location r2 to the trigger register of the FU and the operation starts. The last move transports the result data from the output of the FU back to the register file into location r3.

Compared to OTAs, TTAs are programmed at lower level. OTAs specify the operations to be performed whereas TTAs specify the physical data transports. The previous example shows that typically one RISC operation corresponds to three move operations. It would be too tedious to program all the moves required to perform the desired application by hand. This is why the applications are programmed with high-level language (HLL), typically with C or C++ language. The HLL code is then compiled with the compiler and the data moves are scheduled to the hardware resources available to perform the desired application. The compiler of the TTA is explained more thoroughly in chapter 3.

2.4 MOVE Framework

The MOVE framework is a set of software tools that can automate the design of application-specific instruction set processors [7]. Fast design process is inevitable for ASIPs, since a good cost/performance ratio can only be achieved when the design time is short. The MOVE framework provides a semi-automatic design method, which shortens

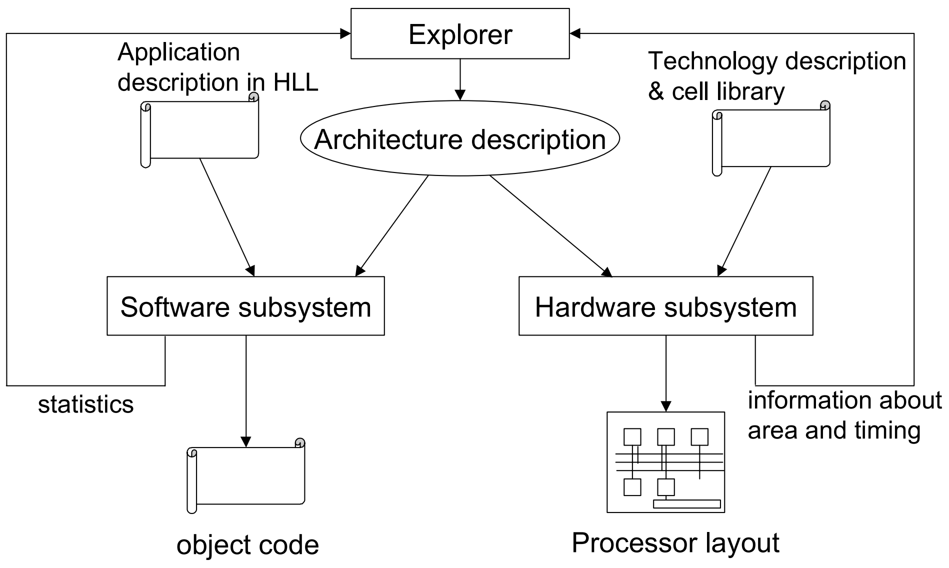


Figure 10. MOVE framework.

the design-time. Processors designed with MOVE framework are based on transport triggered architecture: the MOVE architecture. The scalability of this architecture allows processor configuration to be optimized for a selected application. Furthermore, the flexibility and simplicity of the MOVE architecture allows function units specific to the given application to be easily integrated with the framework to enhance performance. The MOVE framework is designed to exploit these advantages. The MOVE framework consists of three components as illustrated in Figure 10. [7]

The explorer searches the design space for a processor configuration, which yields the best cost/performance ratio for a given application. The hardware subsystem generates the processor layout and produces statistics of timing, area and power consumption. The software subsystem generates instruction-level parallel object code for the selected processor architecture. It provides statistics, e.g., on cycle count, instruction count and hardware resource usage. [7]

2.4.1 Design Space Explorer

Two main design evaluation criteria of the design space explorer are cost and performance, where performance is defined as the inverse of execution time. Cost may include the chip area, the number of pins, the power dissipation, and the code size. Execution time is dependent on the number of executed operations, latencies, cache misses, and the clock cycle time. The explorer searches the processor design space for an optimal design configuration by varying several architecture parameters such as the

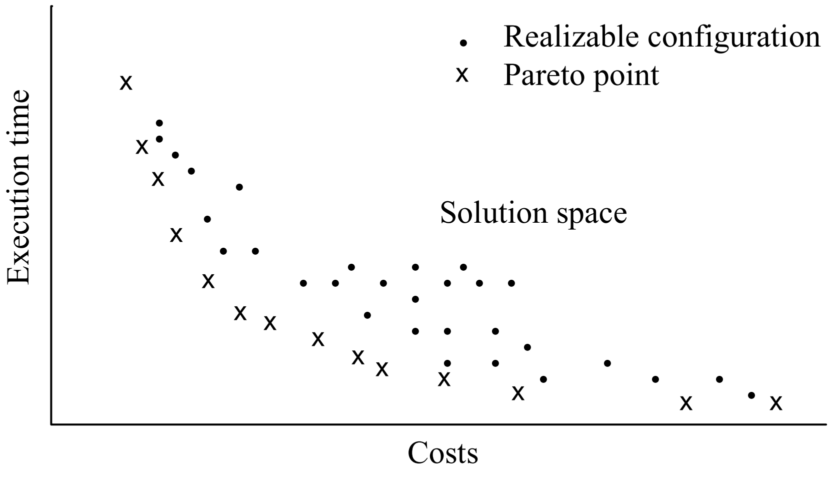


Figure 11. Solution space and Pareto points.

number of move buses, number and type of functional units, connectivity etc. Each solution found is evaluated against the evaluation criteria presented above. [9]

The solutions space is given by all the possible configurations that can be generated by the MOVE framework in the two-dimensional cost-performance space. Figure 11 shows the cost-performance space, which is searched by the explorer. Each marked point corresponds to a specific processor design. The design space is lower bounded by a curve connecting Pareto points. These points are local optimal solutions for a certain cost or performance.

The explorer finds its way through the solution space by iteratively trying different architecture solutions and letting the hardware and software subsystems produce relevant information about these solutions. Information is given about cycle time; cost and the number of cycles elapsed in the executions of the application. Based on this information, the next design point, architecture configuration, is chosen by modifying the architectural parameters. The iterative search process is shown in. [9]

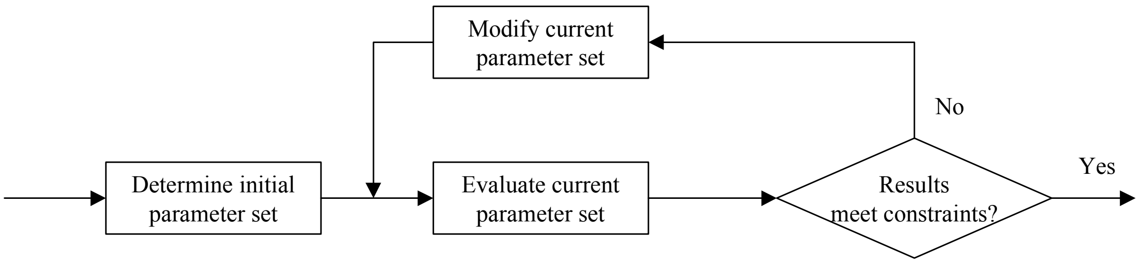

Modify current parameter set
No
Yes
Determine initial parameter set
Evaluate current parameter set
Results meet constraints?

Figure 12. Iterative searching process.

2.4.2 Software Subsystem

The software subsystem of the framework is responsible for generating object code for the selected MOVE processor. The software subsystem consists of a compiler, simulators for sequential and parallel code, profiling and trace analysis tools, and code viewers. The main component is the compiler, which is responsible for generating instruction-level parallel code for the given MOVE processor. It also produces statistics about cycle counts, code size, usage of hardware resources etc. The software subsystem is presented in Figure 13.

The compiler front-end is based on GNU gcc. The output of the front-end is sequential TTA code. Sequential code simulator may simulate this code to generate profiling data. The back-end of the trajectory uses this profiling information when scheduling the sequential code to parallel (scheduled) move code. The parallel code simulator can verify the correctness of this code by comparing the results with the output of the sequential code simulator. [8]

The compiler back-end maps the sequential move code onto the available hardware of the target MOVE processor. It exploits the advanced inter basic block scheduling and software pipelining techniques in order to compact the code as much as possible and to efficiently exploit all the available hardware. It generates code under the assumption that the target

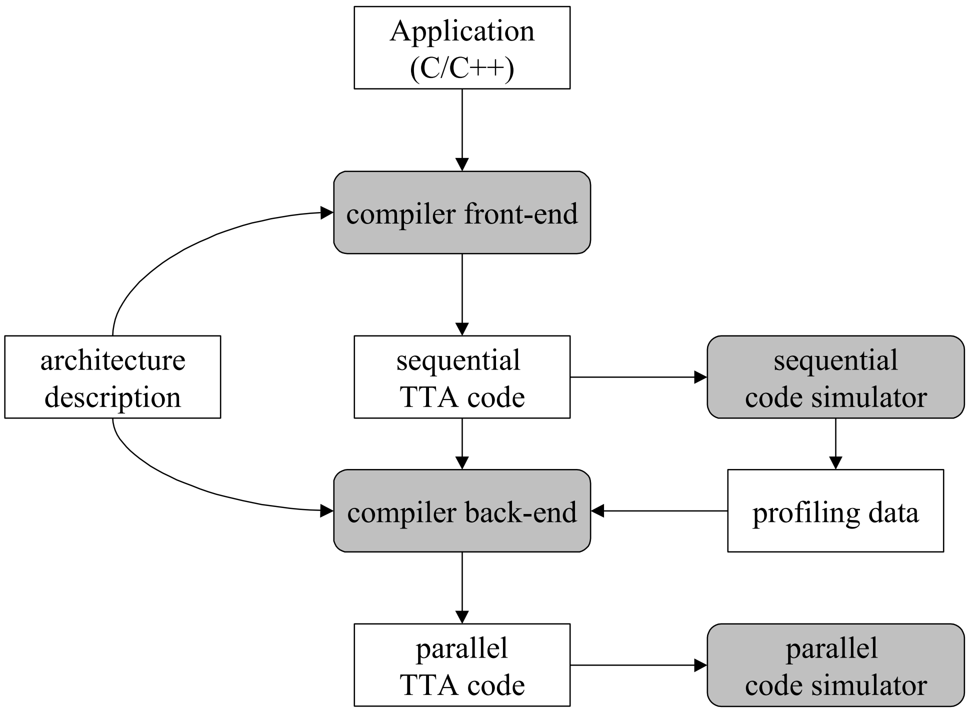


Figure 13. Software subsystem.

processor provides sufficient functionality, e.g., when the compiler front-end generates shift instructions, the hardware needs a shift unit since otherwise the scheduler cannot generate correct code.

2.4.3 Hardware Subsystem

The hardware subsystem of the MOVE framework is responsible for the realization of a MOVE processor on silicon. Organization of the hardware subsystem is illustrated in Figure 14. It consists of three components: processor generator, silicon compiler, and hardware modeler.

The processor generator accepts an architecture description existing as a set of parameters and some function unit designs and generates a synthesizable description of the processor in VHDL, very high-speed integrated circuit hardware description language, in which all parameter dependencies are resolved. The VHDL description can be used as input for synthesis. The silicon compiler takes the VHDL description, the technology information and a cell library as input and produces a VLSI layout of the generated processor. The hardware modeler is a tool, which can quickly make area and timing estimations. This is required in order to be able to evaluate many different architecture descriptions during the synthesis process. The hardware modeler exploits the fact, that during the synthesis process many configurations, which differ only slightly from each other, are researched. The timing and area information of unchanged parts is kept in a database for quick access.

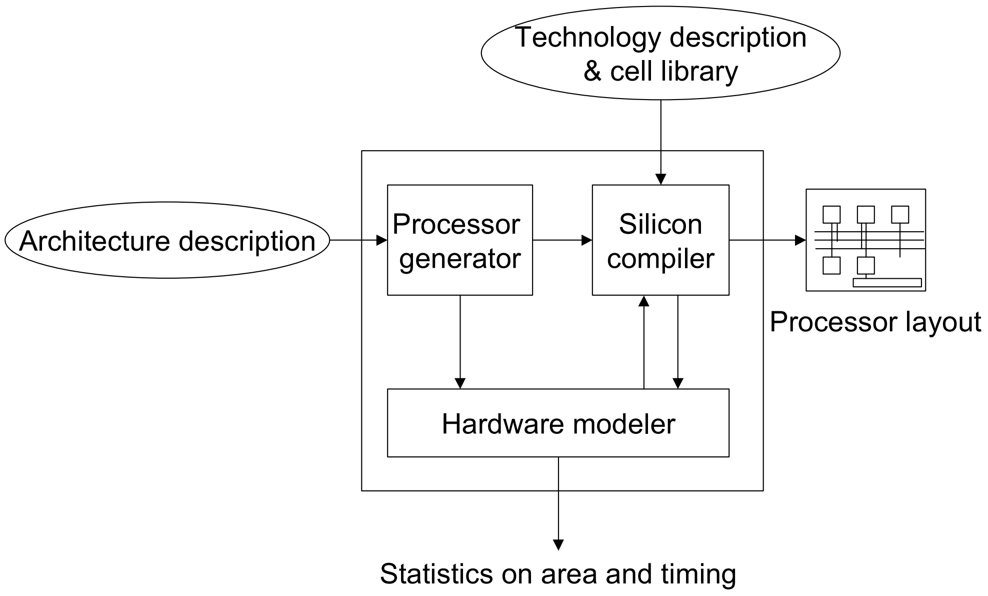


Figure 14. Hardware subsystem.

3. MOVE Tools

Section 2.4 of previous chapter introduced the MOVE framework and its components briefly. This chapter explains more thoroughly these components and how they work. Section 3.1 introduces the design space explorer, Section 3.2 explains the software subsystem and Section 3.3 the hardware subsystem.

3.1 Design Space Explorer

Subsection 3.1.1 gives on overview of the design space explorer, Subsections 3.1.2 and 3.1.3 describe the two phases of the design space exploration; resource optimization and connectivity optimization.

3.1.1 Overview

Designing an ASIP by means of a templated ASIP, like TTA, consists of finding a proper configuration for the given application. The configuration corresponds to values for the architecture parameters of the templated ASIP. These parameters describe the processor design; function units, registers, buses, sockets, and the connectivity between the units. The design objectives are to minimize the cycle count, cycle time, and cost. The objectives are conflicting; a fast processor is typically also an expensive one. Finding a proper configuration for the application requires information about the performance and cost of the processor design. Information about performance is obtained from the compiler of the software subsystem. The hardware modeler of the hardware subsystem gives information about the area of the processor design, which is directly proportional to the cost. Performing this exploration manually would be too tedious. The Design space explorer automates this search procedure. It explores the interesting area of the design space.

The design space has many dimensions but the evaluation function maps it into two-dimensional (2-D) cost/execution time space. Only a subspace of the cost/execution time space will be realizable and from that, only a subspace will be of interest for the designer. These are called Pareto points [9]. A Pareto point is a configuration that is realizable and there exists no other configurations that are both faster and cheaper.

The design space exploration is divided into two separate steps, resource optimization and connectivity optimization. Resource optimization searches the best configuration, which is fully connected, with the right cost/performance ratio. The influence of the full connectivity, i.e. every socket is connected to every bus, is not taken into account. The second step, connectivity optimization, reduces the bus load by removing connections.

3.1.2 Resource Optimization

The objective of the resource optimization is to find a large set of Pareto points, from which the designer can make a choice. Exploration starts with a designer specified architecture configuration. This configuration should be oversized. From this configuration the exploration advances into next configuration by removing one resource of it. This procedure is repeated until a minimum configuration that is needed to perform the application is reached. The following quality function determines, which resource is removed

$$quality(config) = \frac{1}{\text{cost}(config)^{\alpha} \times \text{execution\_time}(config)^{\beta}}$$

where α and β are constants reflecting the importance of cost and performance respectively. After the initial configuration has been reduced to the minimum configuration, the process is reversed and resources are put back until initial configuration is reached. Which resource is put back is also determined by the quality function. Several of these reduce/extend passes are made with different values of α and β [9]. After several reduce and extend passes the design space explorer determines which configurations are the Pareto points. The Pareto points are plotted into the cost/execution time design space.

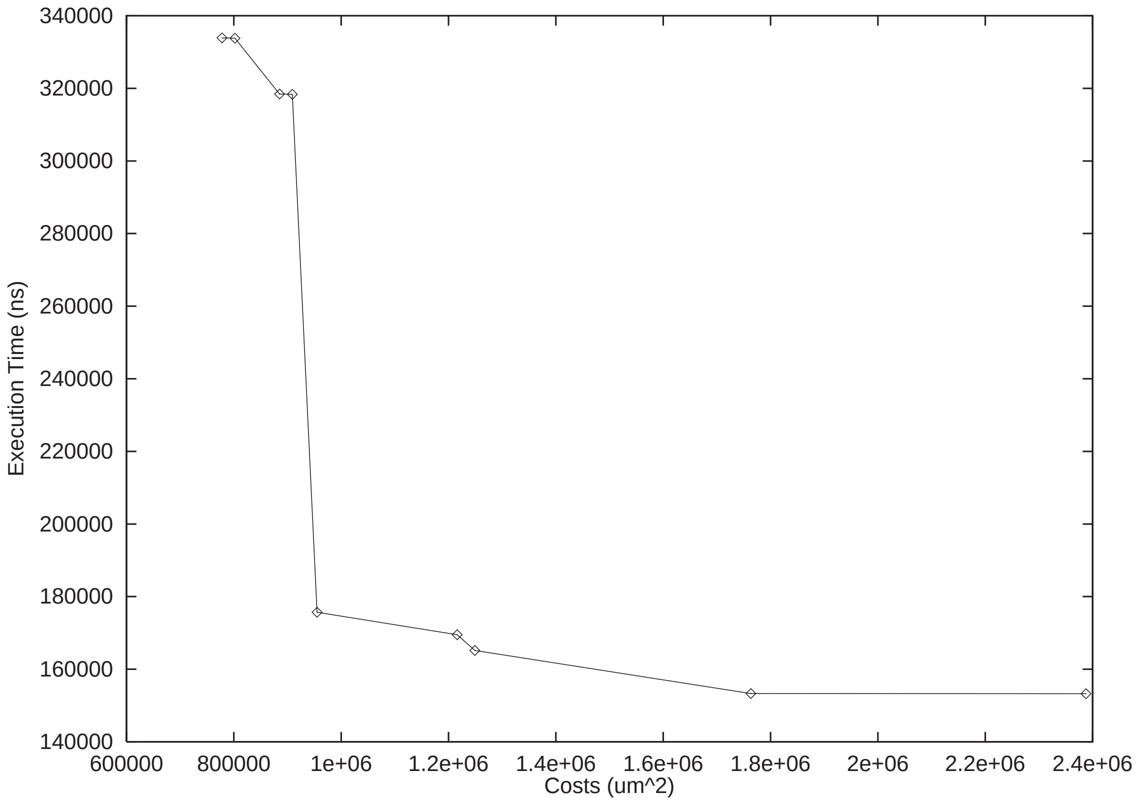


Figure 15. Design space of resource optimization.

Figure 15 illustrates the Pareto points of resource optimization. From the Pareto points the designer chooses few most attractive ones for more detailed evaluation. The designer can choose, e.g., the fastest, the cheapest or the best compromise between cost and execution time. These configurations are evaluated in more detail by generating processors for them for an accurate cycle time and cost estimation. The designer has to choose one of these configurations for the next step, connectivity optimization. [9]

3.1.3 Connectivity Optimization

Connectivity optimization is done for the fully connected configuration selected after the resource optimization step. Connectivity optimization transforms this fully connected configuration into partially connected configuration that has less load on the move buses and therefore shortens the clock cycle. This is done by removing move socket connections from the move buses in a round robin fashion [9]. The connection that is removed has no effect on the clock cycle count. If no such connection exists, the connection that has the lowest effect on the cycle count is removed. This procedure is repeated until the clock cycle time remains constant and then starts to increase. By removing the connections in

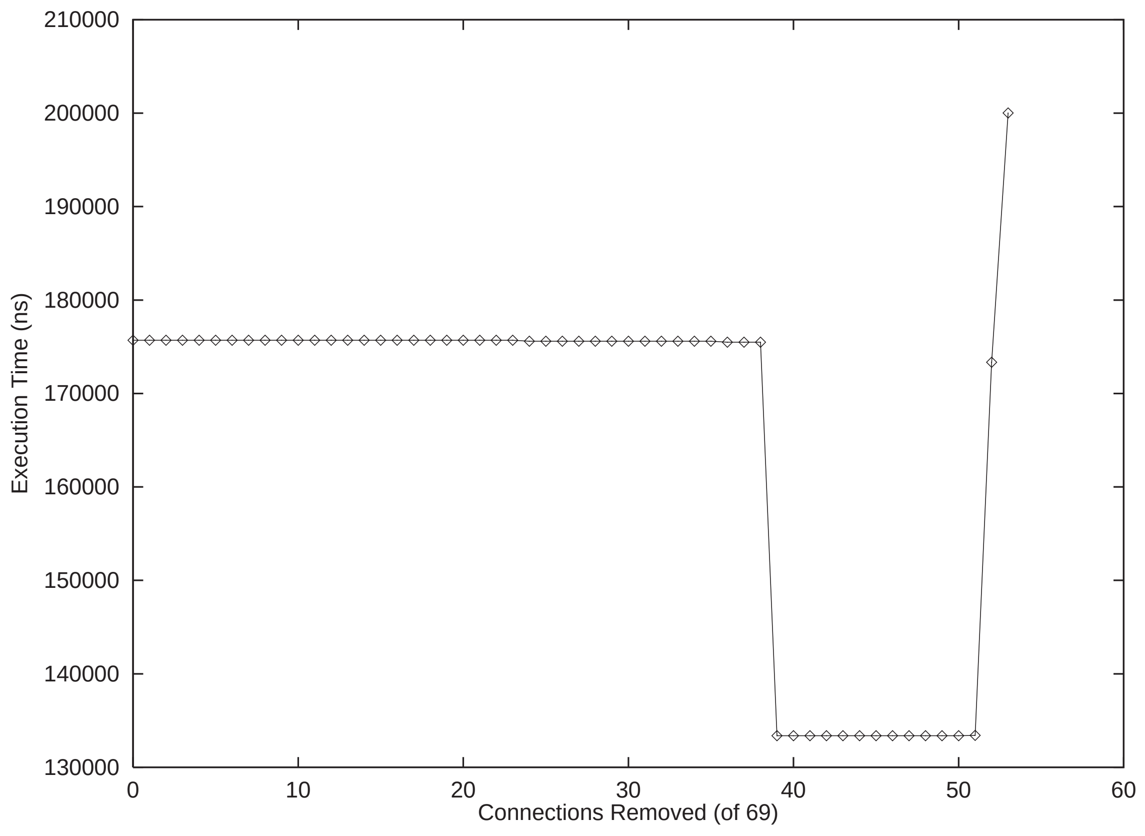


Figure 16. Design space of connectivity optimization.

round robin fashion the bus load is balanced, since all move buses have approximately the same number of connections. The design space of connectivity optimization is depicted in Figure 16. It illustrates how many connections were removed and the execution time of each configuration. [9]

3.2 MOVE Software Subsystem

The MOVE software subsystem consists of two components, the front-end and the back-end. The front-end is described in Section 3.2.1 and the back-end in Section 3.2.2.

3.2.1 Front-End

The front-end of the MOVE software subsystem is responsible for generating an intermediate format of an application written in high level language, C or C++. This intermediate format is called sequential move code. The front-end is illustrated in Figure 17. The C or C++ application is first compiled into assembly format. The assembler of

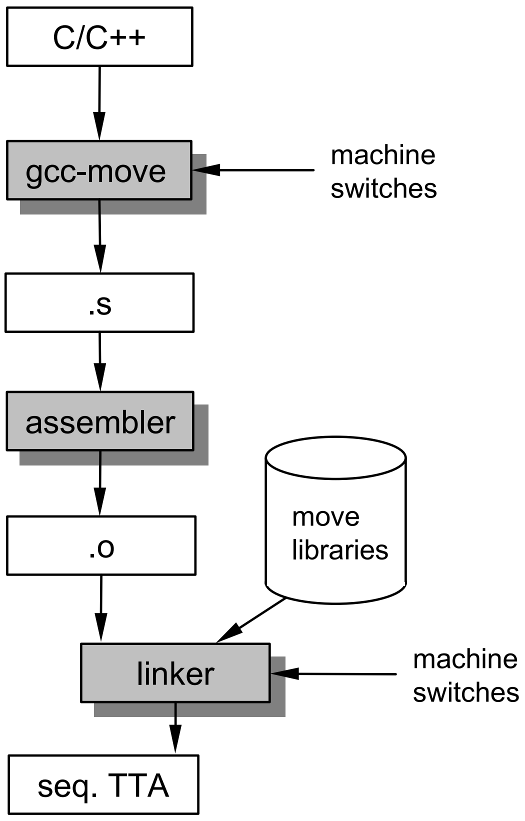


Figure 17. Front-end of software subsystem.

the front-end is ported into generic MOVE architecture representing RISC-like instruction set and a large register file. The generic MOVE architecture is depicted in Figure 18.

The register file contains 128 32-bit integer and 128 64-bit floating-point registers and one boolean and one return address register. The MOVE instruction set contains all the possible instructions the MOVE processor is able to perform. The resulting MOVE

Register Sets

128x32 interger register file:

r0…r127

128x64 floating-point register file:

f0…f127

1 boolean register: b0

1 return address register: ra

Generic MOVE Instruction Set

jmp, call, trap,
add, sub,
mul, div, divu, mod, modu,
and, ior, xor, shl, shr, shru,
eq, gt, gtu,
eqf, gtf,
ld, st,
lhd, sth, ldb, stb, ldhu, sthu, ldbu, stbu,
ldd, std, lds, sts,
f2i, f2u, i2f, insb, insh, extb, exth

Figure 18. Generic MOVE architecture.

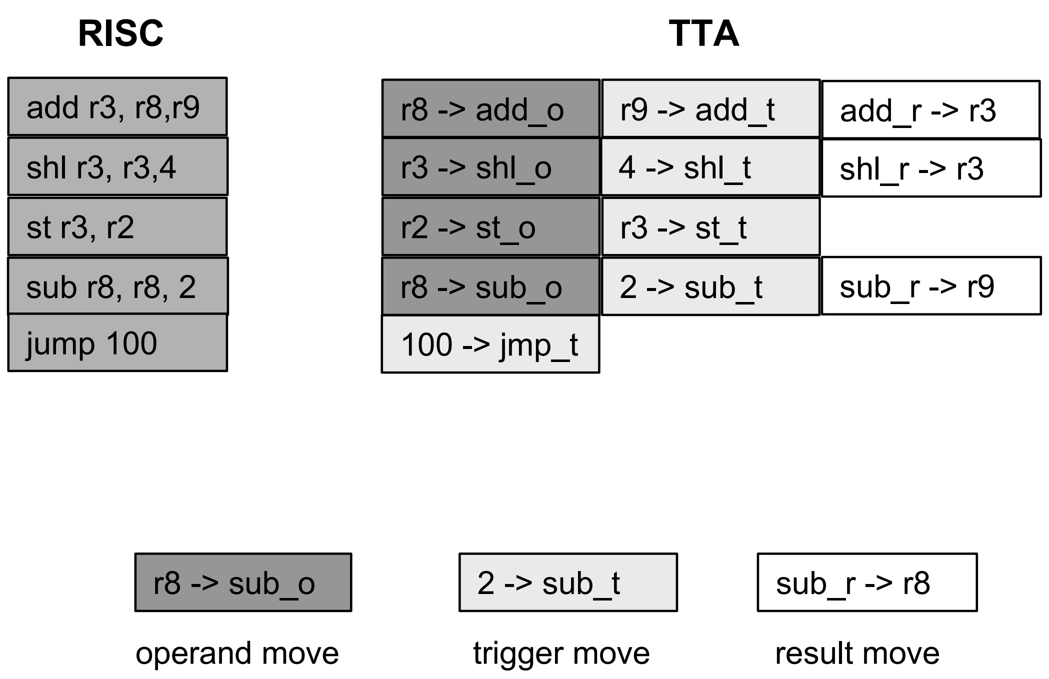


Figure 19. Correspondence between RISC and sequential MOVE assembly code.

assembly code consists of moves realizing RISC-like operations. There are normally three moves, operand, trigger and result moves, corresponding to one RISC-like operation. The correspondence between RISC and move assembly code is illustrated in Figure 19. Operand and trigger moves are input moves transporting data into input registers of a function unit. Result move transports the result from the result register of the function unit to another destination. The operation starts when the trigger move emerges. If the operand move has not been realized before or in the same cycle with trigger move, the result will be wrong, since the value of the operand move is not yet present in the function unit. The generated move assembly code is then assembled into object code. The object code is linked with the MOVE libraries into sequential move code, which is in binary format. [10]

3.2.2 Back-End

As a result from the front-end a sequential move code is obtained. This code is then parallelized and mapped onto the available hardware resources of the target architecture. This is the task of the back-end, which is depicted in Figure 20. The sequential move code is first simulated with the sequential simulator, which produces profiling information for the scheduling. Profiled information from the sequential simulator is taken as input by the scheduler, the main component of the back-end. The scheduler also gets the sequential move code and the architecture description describing the target MOVE processor. In the

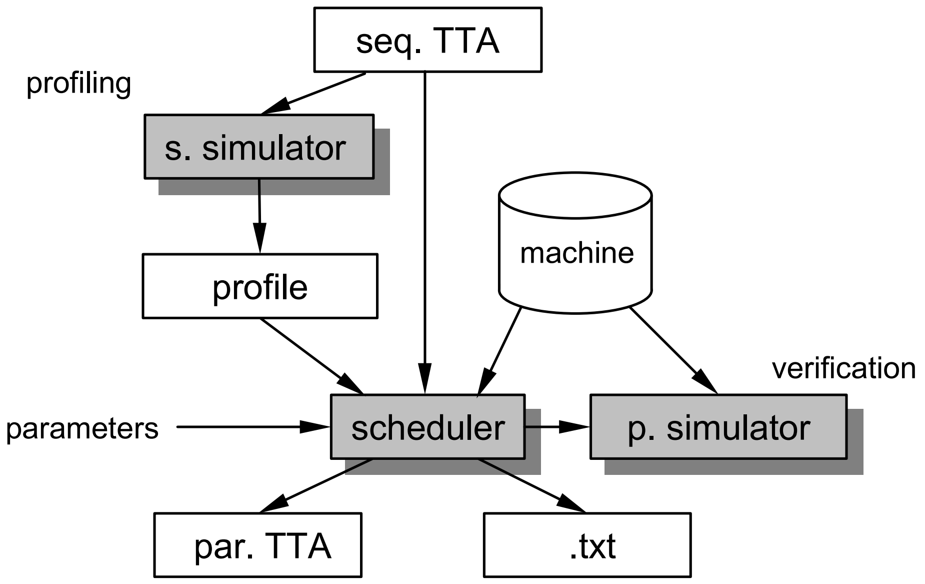


Figure 20. The back-end of software subsystem.

architecture description, the user specifies the resources of the MOVE processor, i.e., move buses, sockets function units and register files. In addition, the connectivity of the transport network is also specified in the architecture description file. With these three input files the scheduler then performs five steps steps: reading the sequential move code, reading the architecture description, optimization, generating data dependency graphs, and scheduling and resource allocation.

First two steps deal with reading the sequential move code to be parallelized and parsing the necessary information required for scheduling from the machine description file. Optimization is a complex procedure consisting of optimizing control-flow graphs and data flow analysis. A data dependency graph is then generated for every block of code to give information about the dependencies between moves and resources in each block of code. Figure 21 shows the data dependency graph of the MOVE code depicted in

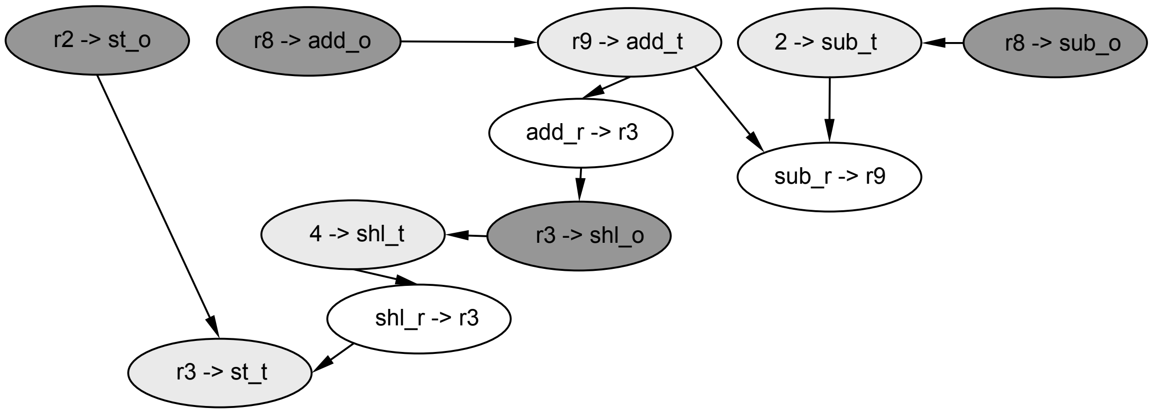


Figure 21. Data-flow graph.

Figure 19. The data-flow graph presents the dependencies between different moves and resources. Operand move of an operation must always exceed trigger move and the trigger move must exceed corresponding result move.

After the generation of data dependency graphs, the moves are scheduled into clock cycles and allocated into the resources specified in the machine description. The scheduling is operation based meaning that the operations are scheduled one at a time. Scheduling an operation is illustrated below in Figure 22. Scheduling starts from the trigger move.

A function unit having the hardware to perform the operation to be scheduled is chosen. Then a source socket setting the value on a bus and a destination socket obtaining the data from the bus to the function unit are selected. Finally a free bus is selected to where the trigger move is set. If the move cannot be scheduled in current cycle, the same procedure is done in the next cycle. After the trigger move is successfully scheduled the same is done for the operand move taking into consideration that it must be scheduled before or in the same clock cycle as the trigger move. The result move is scheduled last. It must be scheduled in a later clock cycle than the trigger move. [10]

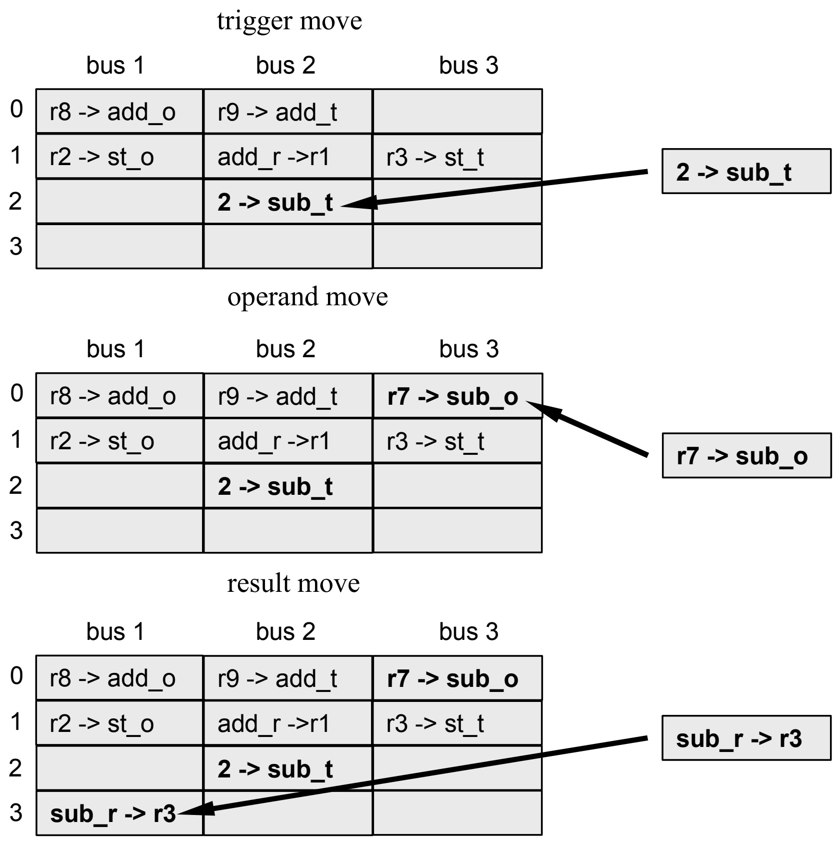


Figure 22. Operation scheduling.

Sequential move code

```
1:  // predecessors : { 0 8 }
    // successors   : { 2d }
    // live in      :
    // live out     :
    // live def     :
    // live use     :
    // a.out range  : <424, 519>
    // frequency    : 9 (* 6 = 54)
    // moves/cycle  : 2
    // loop id      : 0x0
    r28 -> r12;
    0 -> r19;
    r21 -> shl_o, 1 -> shl_t, shl_r -> r22;
    r23 -> shr_o, 1 -> shr_t, shr_r -> r27;
    r23 -> shl_o, 1 -> shl_t, shl_r -> r24;
    r23 -> r20;
```

Parallel move code

```
1:  // predecessors : { 0 7 }
    // successors   : { 2d }
    // live in      :
    // live out     :
    // live def     :
    // live use     :
    // a.out range  : <424, 519>
    // frequency    : 9 (* 2 = 18)
    // moves/cycle  : 6
    // loop id      : 0x40b7f118
    r28 -> fu3.shl_o, 1 -> fu3.shl_t,
    r26 -> fu4.shr_o, 1 -> fu4.shr_t,
    r26 -> fu5.shl_o,1 -> fu5.shl_t;
    rv -> r3, 0 -> r30, fu3.shl_r -> r27,
    fu4.shr_r -> r22, fu5.shl_r -> r25,
    r26 -> r29;
```

Figure 23. From sequential to parallel Move code.

After scheduling the parallel move code is obtained. The scheduler produces the parallel move code both in binary and textual formats. With a special tool called viewcode the original sequential move can also be seen in textual format. Figure 23 illustrates sequential and parallel move codes of the same block of code. Performing the sequential code would require 6 clock cycles whereas performing the parallel code of the same block requires only 2 clock cycles. The moves in the sequential code are mapped into resources of the target machine, which has 6 buses. Due to this only 2 clock cycles are needed.

Performance of the parallel code can be obtained by using the parallel simulator. The current version of the parallel simulator is not a stand-alone program. It is invoked by giving special options for the scheduler. The parallel simulator gives statistics about the performance of the processor design and also some other evaluation information. Statistics

```
Some execution counts:
        #Moves                      :    346,047
        #Cycles                     :    130,904
                                     (   130,907)
        #Basic blocks               :      9,745
        #Procedures                 :          1
        #GPR reads                  :    169,277
        #GPR writes                 :    103,756
        #Operations                 :    130,907

Code size:
        #Moves                      :        324
        #Instructions               :        132
```

Figure 24. Output of parallel simulator.

is obtained, e.g., about the number of moves performed in the application, the number of clock cycles elapsed, the number of operations and the number of instruction. An example of the statistics is depicted in Figure 24.

3.3 MOVE Hardware Subsystem

As described in Section 2.3, MOVE hardware subsystem is responsible for generating the hardware of the processor design. The hardware subsystem consists of MOVE processor generator (MPG), silicon compiler and hardware modeler. The MPG and the silicon compiler are explained in Section 3.3.1 and the hardware modeler in Section 3.3.2.

3.3.1 MOVE Processor Generator

The MOVE processor generator generates the VHDL of a MOVE processor according to architecture parameters given by the user. The architecture parameters are divided into several files given as input to the MPG. The overview of MPG is depicted in Figure 25. It illustrates which files are needed as inputs and which files are produced as outputs.

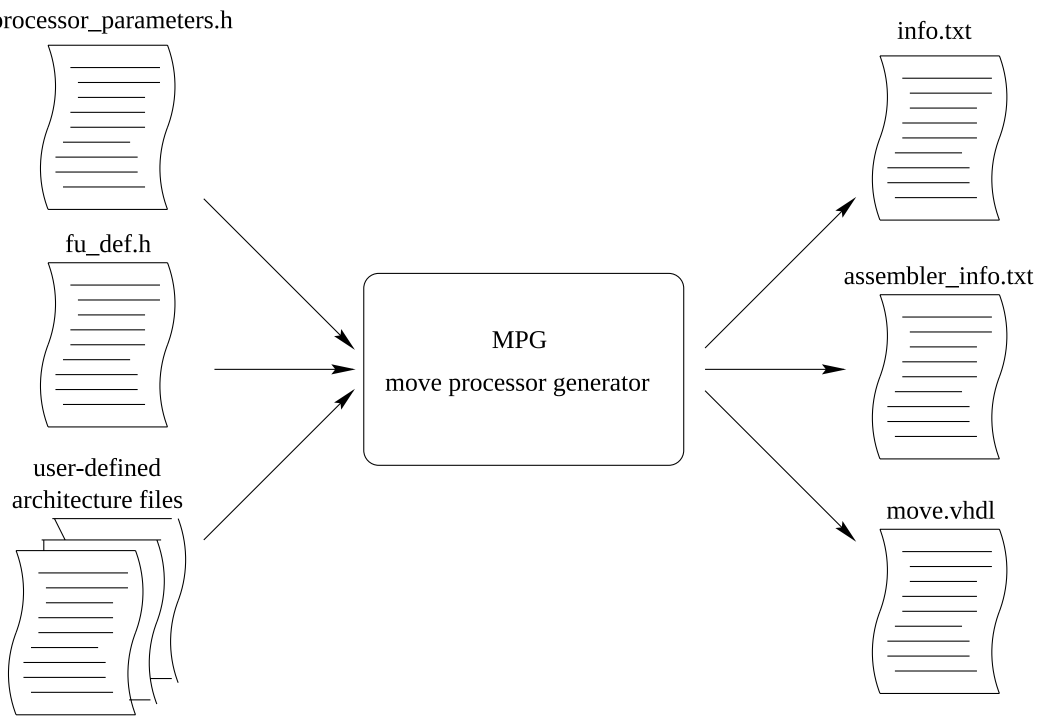


Figure 25. Overview of MOVE processor generator.

“processor_parameters.h” is an input file, which contains a list of parameters changeable by the user. The parameters specify global characteristics of the processor (number of buses, buswidths) and characteristics of some function units, e.g., instruction fetch unit and load-store unit. “fu_def.h” is the second input file, which is used for defining function units and adding them to the architecture of the processor. Two arrays, “fudefs[ ]” and “fuinsts[ ]”, are defined in this file. The array “fudefs [ ]” contains all function unit type definitions, e.g., the number of sockets, number of id-bits, datawidth of sockets and whether function unit is user-defined or built-in. The array “fuinsts[ ]” contains the instantiations of function units defined in array “fudefs[ ]”. The function unit is added to the processor design and connections of the function unit to buses are specified in this array. User-defined-architecture-files are the last input files containing the VHDL architectures of the special function units developed by the user. When a user wants to add a special function unit to the processor architecture, MPG needs the VHDL code describing the behavior of this function unit. Therefore, the user only has to design the internal parts of the function unit. Other parts needed, e.g., sockets, connections to buses and input and output registers are automatically generated by the MPG. User-defined-architecture-files are in CHDL format, which is a combination of C and VHDL languages. The architecture description of SFU is written in VHDL inside a C-function, which is then called by the MPG and correct VHDL code is generated for the SFU.

“info.txt” is an output file containing information about the generated processor architecture. It contains an overview of the chosen and calculated parameters, which describe the designed processor. “assembler_info.txt” output file contains chosen and calculated parameter values of processor characteristics. The assembler program, generated for testing the processor design, needs these values for generating correct code for this particular MOVE processor design. MPG generates this file each time a new processor design is generated. “move.vhdl” is the most important output file containing all generated VHDL code for this particular processor design describes in input files.

When all necessary input files have been updated to correspond to current processor design, the MPG can be run to generate VHDL code. Before it is run, it must be compiled. The complete trajectory of the MPG is shown in Figure 26.

First step of the trajectory is the CHDL pre-processing. The MPG is almost completely written in CHDL, which is a combination of C and VHDL languages. The pre-processing will create C files of all CHDL files. The C files generated by the CHDL pre-processing are compiled in the next step by the gcc compiler. Executable file of MPG is generated.

The generated executable is then run. It generates file “move.vhdl” containing the VHDL description of the processor. Also files “info.txt” and “assembler_info.txt” for the assembler of the MPG are generated. There is a makefile, which does pre-processing, compiling and running automatically.

The generated VHDL code will be then tested and synthesized. For testing the processor design a testbench is created. Also necessary signals and external memory for executing a test program are created. The external memory will read a memory file with the program when the simulation is started. The program is a list of memory locations with the binary data, which should be put in these memory locations. To make the writing of the test program easier, an assembler program was created for this purpose. The assembler generates a binary format memory file from a file including the data moves in the processor written using names of destination and source sockets instead of binary code. Commercial tools, e.g., Design Compiler of Synopsys are used to synthesize the processor design.[7]

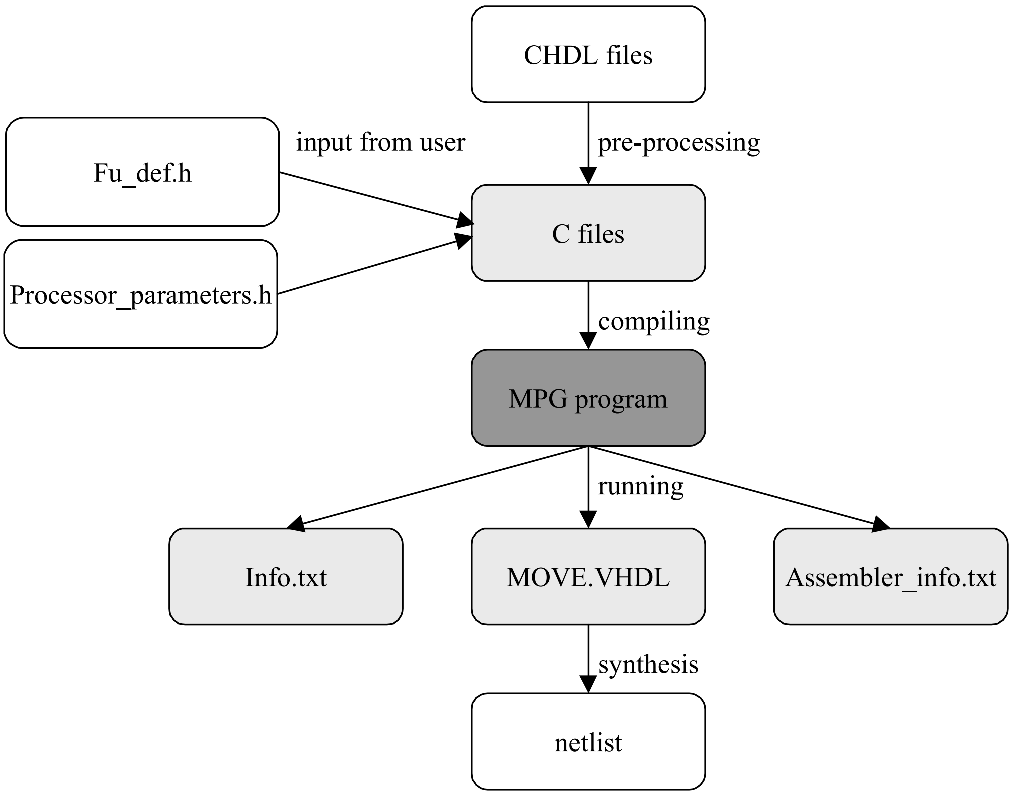


Figure 26. The MPG tool flow.

3.3.2 Hardware Modeler

The hardware modeler is a tool, which can quickly make area and timing estimations. This is required in order to be able to research many different architecture descriptions during the design space exploration process.

Area estimation is straightforward: the area of the configuration is the sum of the area needed for FUs, sockets, register files, transport buses, and bonding pads. There are separate area expressions for each type of FU, from which the area occupied by all FUs is calculated. The area of input and output sockets depends on the number of buses connected to the sockets and the width of these buses. With the help of area estimations for the sockets and FUs, the area occupied by the transport-buses can be calculated. The area of a bus increases quadratically with the number of wires on the bus. The number of bonding pads for data is determined by the number of bits in the instruction word plus the number of data pins on the load/store unit. The size of the instruction word depends on the number of source and destination addresses. [11]

The calculation of the attainable cycle time depends on the used pipelining scheme. In three-stage pipeline the attainable cycle time is the maximum of the longest FU stage delay and the longest bus transport delay of the entire processor design. In the case of two-stage pipeline, the FUs do not have result sockets meaning that final FU-stage is connected directly to the output sockets and hence to the transport network. Due to this, the path dictating the cycle time is the slowest final FU-stage/transport-bus combination. [11]

4. Performance Evaluation

Performance of a processor is an important property. Applications run on processors have become more complex and thus heavier for the processors to perform. Execution time of an application is a property of great concern when evaluating the performance. Throughout the history of processors the aim has always been to develop faster processors being able to perform more complex applications. There are two ways to increase the speed of a processor. Either to decrease the clock cycle time or to decrease the number of clock cycles elapsed performing the application. In addition to the execution speed, also the size of the executable binary code is important when evaluation the performance of a processor. When the code size increases also the memory size requirements are increased. Memory space is a property affecting heavily on the cost of a processor design.

The MOVE software subsystem described in chapter 3 was used to investigate the performance of MOVE architecture in discrete trigonometric transformations: fast Fourier transform (FFT) and discrete cosine transform (DCT). The performance of MOVE architecture was measured against the performance of TI TMS320C62x (C62x) digital signal processor by configuring the architecture of MOVE processor design to match as much as possible to the architecture of C62x. In addition, second architecture for MOVE processor was configured by using the design space explorer to find the best possible architecture for the application in question. The effect of different coding styles on the performance of MOVE architectures and C62x was also examined.

Comparing the performance of a MOVE processor design with an architecture configuration similar to the architecture of C62x gives some insight about the performance of MOVE architecture. Comparing also the optimized architecture configuration gives information about the effectiveness of the design space explorer to generate ASIPs. The performances of these three processors were compared with four different applications: 512-point complex FFT and three versions of 8x8 DCT. To compare the execution speed of MOVE architecture and C62x processors the focus was put on to compare the number of clock cycles elapsed performing the applications instead of comparing the execution time. The cycle times of the MOVE processor design depend on the architecture configurations. The cycle time estimations are based on rather old 0,7 μm technology, which results quite long cycle times and thus has quite a big impact on the execution time. Resulting clock frequencies of MOVE vary between 25 and 74 MHz depending on the architecture configuration. For comparison, the TMS320C6203 processor is realized using 0,15 μm technology with maximum clock frequency of 300 MHz.

Section 4.1 describes the architecture configurations used in the simulations. Sections 4.2 and 4.3 describe the FFT and DCT simulations performed. Section 4.4 describes the simulations performed to investigate the effect of different coding styles on the performance of MOVE and C62x. Lastly, Section 4.5 analyses the results obtained in sections from 4.2 to 4.4.

4.1 Architecture Configurations

For the simulations the MOVE architecture was configured to correspond as much as possible to the structure of TMS320C62x DSP. Due to the differences of the architectures, full match between these two architectures is impossible to be obtained. To configure the architecture of the MOVE processor to match to the architecture of C62x requires investigation of the architecture of C62x. Thus, subsection 4.1.1 describes the architecture of C62x. Subsection 4.1.2 describes the architecture configuration made for the MOVE processor designs.

4.1.1 Architecture of TMS320C62x

C62x includes eight functional units, from which two are multipliers and the rest six are arithmetic logic units (ALU). The processor contains two general-purpose register files each having 16 registers. The processor is divided into two data paths. Each data path contains four functional units, three ALUs and one multiplier, and a register file.

Functional units in one data path are almost identical to the units in the other data path. Basic operations of the functional units are illustrated in Table 1 [12], [13].

Table 1. Operationality of functional units.

Functional unit	Operations
.L unit (.L1 and .L2)	30/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits
.S unit (.S1 and .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit-field operations 32-bit logical operations Branches Constant generation
.M unit (.M1 and .M2)	16 x 16 multiply operations
.D unit (.D1 and .D2)	32-bit add, subtract, linear and circular address calculations Loads and stores

Each functional unit contains two 32-bit read and one 32-bit write port to the register file. .S and .L units also contain additional 8-bit read port for providing 40-bit integer operands [12]. Each data path has also a cross path to read operands from the other register file. Herewith, both register files contain in total 16 ports, 10 read and 6 write ports. In addition to these register read and write paths going to the functional units, both register files contain one store-to-memory data path and one load-from-memory data path to the memory. The simplified architecture of C62x, not showing the cross paths and the 8-bit read ports, is depicted in Figure 27.

All of the instructions that C62x can perform have functional unit latency of one. This means that a new operation can be started each clock cycle in all functional units. The processor is thus able to start eight new operations each cycle. The execution latency of a functional unit depends on the operation it performs. Single-cycle operations, e.g., ADD, have execution latency of zero meaning that when input operands are read in cycle i, the result can be read in cycle i+1. Execution latency thus corresponds to delay slots. Execution latencies of most common operations are illustrated in Table 2 [12]. One functional unit can perform operations with different latencies.

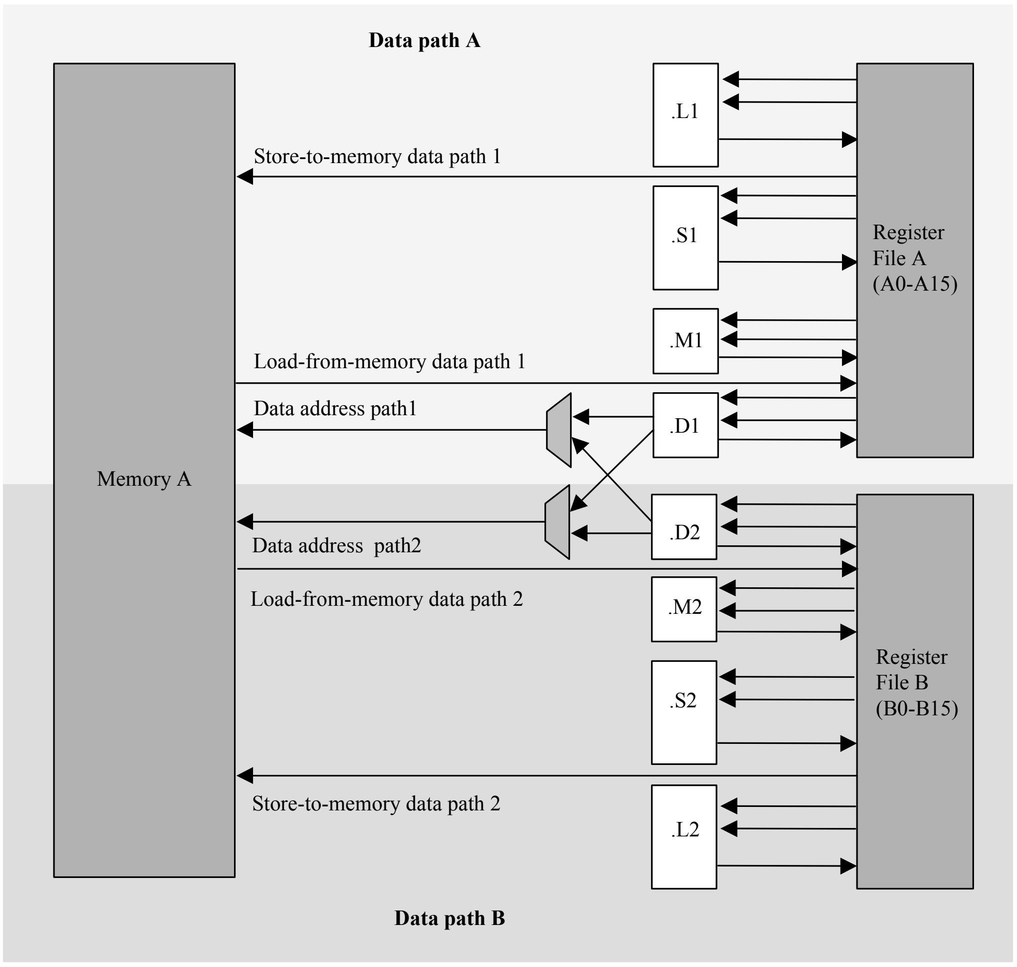


Figure 27. Architecture of TMS320C62x.

Table 2. Execution latencies of most common operations.

Operation	Execution latency
NOP (no operation)	0
Store	0
Single-cycle	0
Multiply 16 x 16	1
Load	4
Branch	5

4.1.2 Architecture Configuration of MOVE Processor

The architecture configuration of the MOVE processor was specified in the machine description file. To make the functionality of MOVE and C62x processors compatible, two multipliers and six ALUs were defined also for the MOVE processor. The operation sets of the function units were defined to correspond to as much as possible to the operations of the functional units of C62x. The multipliers were defined to perform only multiplication and for ALUs, three different types were defined. These types and the operations they can perform are described in Table 3. ALU_1 correspond to .L units, ALU_2 to .S unit and ALU_3 to .D units of C62x.

Table 3. ALU configurations for MOVE processor.

ALU type	Operations
ALU_1	Arithmetic Compare Logic Sign extension
ALU_2	Arithmetic Shift Logic Sign extension
ALU_3	Arithmetic Sign extension

In addition to these eight FUs, two load-store units (LSU) had to be added. In C62x, the .D units perform load-store operations. As was illustrated in Table 2, the execution latency of the load operation is four meaning that the .D units perform operations with different latencies. However, this is not possible in MOVE architecture and thus separate LSU had to be added. The single-cycle store operation was also moved to LSU, since it is reasonable to perform both memory access operations in the same FU. .S units of C62x perform in addition to single-cycle operations branch operations. The branch operation is not present in the corresponding ALU type of MOVE since the branching of moves in MOVE architecture is performed in another way. Jump instructions that can be squashed are used. The guard field at the beginning of each move slot specifies whether a move to jump destination is squashed or not according the value of the guard.

Function units of MOVE architecture are pipelined meaning that they can start new operations each cycle, as the functional units of C62x. The latency of the FUs of MOVE architecture must be specified explicitly. The latency specifies in which clock cycle the results can be read from the FU, e.g., if the latency of an FU is 1, results of an operation initialized in cycle i can be read in cycle i+1. Following this principle, the latencies of the multipliers of MOVE processor were defined to be 2 and the latencies of ALU_1, ALU_2 and ALU_3 FUs performing only single-cycle operations were defined to be 1 to correspond to the latencies of C62x. The result of the load operation of MOVE architecture can be read in cycle i+3 and thus latency of 3 was defined for the LSUs.

In MOVE architecture, the registers are divided into boolean and integer register files. The boolean register file is not so important, but there must be at least one boolean register. For MOVE processor a boolean register file of four registers and two integer register files of 14 registers were specified. This way the total number of registers is 32, the same as for C62x. The boolean register file is used so rarely that it is enough to specify one read and one write port for it. Both integer register files of C62x have 16 ports, 10 read and 6 write ports. For the two integer register files of MOVE, eight read and four write ports were defined. This is well enough, since in the MOVE architecture register files are handled as FUs and thus need not the worst case situation register accesses. In addition to integer registers, the MOVE processor must also contain at least one long immediate register to handle immediate values that are longer than 8-bits. Long immediates are used quite rarely and thus one long immediate register of 32-bits was specified for the MOVE processor design. In C62x, long immediates (constants) are generated in S-units [12].

Initializing an operation of MOVE architecture normally corresponds to two moves. To perform these two moves in the same cycle requires two buses. Thus, starting eight operations each cycle, as C62x, requires 16 move buses. In addition to move buses needed in order to start eight operations each cycle, 8 additional buses are needed to transport the results of the previous operations. Thus 24 buses would be required. This is however overkill since the scheduler of the software subsystem is rarely capable of finding enough parallelism from application to be exploited by scheduling 24 moves each clock cycle. To determine the number of move buses to be specified for the processor designs the performance was evaluated with different number of move buses for each of the applications simulated. The evaluation was started from an architecture configuration with 16 move buses from which two move buses were removed at a time. After the number of buses was reduced to 10, one bus was removed at a time. The performance was measured by investigating the number of clock cycles elapsed, the number of instructions required and the width of the instructions. These values for different number of move buses for FFT application, described in more detail in section 4.2, are depicted in Table 4. The number of clock cycles starts to increase when the number of move buses is reduced to eight, although the change is only one clock cycle compared to configuration with nine buses. With eight buses the number of instructions is increased by one but at the same time the instruction width is reduced 24 bits from 249 bits to 225 bits thus resulting in much smaller code size. On the basis of these simulations architecture configuration with eight move buses was selected for FFT simulations.

Table 4. Cycle counts for different number of move buses.

Number of move buses	**Cycle count**	**Number of instructions**	**Instruction word width (bits)**
16	140,158	117	417
14	140,158	117	360
12	140,158	117	321
10	140,158	117	273
9	140,158	117	249
8	140,159	118	225
7	142,234	123	200
6	143,241	123	177

Similar simulations to choose the number of move buses were performed also for other application simulated, namely DCT algorithms of Loeffler, Wang and Kwak described in more detail in section 4.3. For all these three DCT applications a configuration with ten move buses turned out to be the best one when taking into account the cycle count and the code size.

Each function unit of MOVE architecture has two input registers and one output register. Due to this, each unit requires two input sockets and one output socket to realize the connections between the unit and the move buses. The transport network of the MOVE processor configurations was first configured fully connected meaning that every socket is connected to every bus. Fully connected network is, however, overkill and it increases the instruction size. The drawback of this is the increase in code size. Due to this the connectivity optimization step of the design space explorer was performed. It optimized the transport network by removing unnecessary connections between sockets and move buses. Connectivity optimization was performed for FFT and DCT applications to configure the best possible interconnection network for all the applications. From the results of the connectivity optimization a configuration with the smallest cycle count and least connected transport network was selected. An example of resulting architecture configuration, namely for 512-point FFT application, is illustrated in appendix A.

In addition to MOVE architecture configurations matching to the architecture of C62x, the design space explorer of the MOVE framework was used to obtain the best architecture configuration for each application. An initial architecture configuration containing double the resources of C62x was first given to the explorer for resource optimization. From the results of the resource optimization an architecture configuration with the lowest cycle count was selected. The interconnection network of the selected architecture configuration was then optimized with the connectivity optimization step of the explorer. Once again, the architecture with the lowest cycle count and the least connected network was selected according to the results.

The architecture configurations obtained from the explorer for all FFT and DCT applications contained the same resources, which are illustrated in Table 5. This can be explained by the similarity of FFT and DCT algorithms. The basic element of both algorithms, called butterfly, performs one addition, one subtraction, and one multiplication, which require two ALUs and one multiplier as shown in Table 5. The architecture configurations differed only in the connectivity of the interconnection network. Resulting architecture configurations contain only few resources. The experiments made showed that with more resources better cycle counts could be obtained. The reason why the optimized architecture configurations, obtained from the explorer, contain so few resources is that the explorer optimized execution time, not the cycle count. Lower execution time can be obtained by reducing either cycle count or the cycle time. The explorer tries to minimize both. Reducing the cycle time is more effective. This is done by reducing the bus load, which is done by removing components from the architecture thus reducing the number of FUs. This results in lower execution time but greater cycle count. Optimized architecture configuration for 512-point FFT application is presented in appendix B.

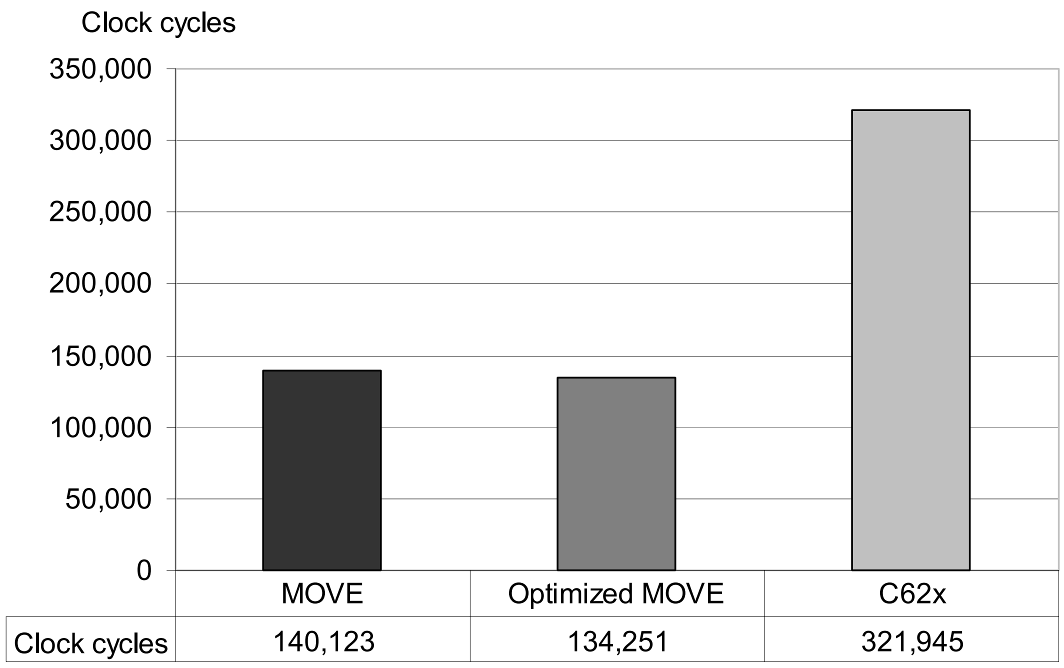
Table 5. Hardware resources of optimized architecture configurations.

Resource	Configuration
Buses	1 1-bit bus 6 32-bit bus
Function units	1 load-store unit 1 multiplier 1 ALU of type ALU_1 1 ALU of type ALU_2
Registers	2 integer register files with 16 register 1 boolean register file with 3 boolean registers

4.2 Fast Fourier Transform Simulations

For the simulations, an FFT application was written in C-language. The application performs 512-point in-place decimation-in-frequency radix-2 Cooley-Tukey complexvalued FFT described in [14]. The code was written with 16-bit fixed-point arithmetic in fractional representation. The in-place algorithms are often preferred in software implementations since the results can be stored into the same memory locations as the input operands. To avoid using floating-point function units and registers the application was programmed to use fixed-point numbers. Due to this the input values and factors used in the transform had to be scaled by multiplying the floating-point values with 2^{15} resulting in 16-bit integer values. This was done with Matlab. Resulting inputs and factors were stored into a header file, which was included in the C-code. Due to scaling to fixed-point representation, every time when values were added together in the transform, they had to be divided by two to avoid overflow. To avoid using division, the divide by two operation was performed by shifting the value one bit to the right. Also when values were multiplied together, they had to be divided by 2^{15} to scale the result back to 16-bit representation. This was done by shifting the values 15 bits right. After all phases of the transform were made, the results were scaled back to the original form by shifting the final results the same amount of bits left as they were previously shifted right. Due to these scaling some accuracy was lost.

The performance of the MOVE processor design was evaluated using the MOVE software subsystem. The FFT application was compiled with the highest optimization level to parallel move-code. In the scheduling phase, the parallel simulator was invoked and statistics about the performance was obtained. The performance of TMS320C62x was evaluated using the Code Composer Studio 1.2. The FFT application was compiled with the highest optimization level and the program was loaded and run in the C62x simulator from which statistics was obtained about the performance. The clock cycle count and the code size of the binary executables were measured. The number of clock cycles of the MOVE processor design was obtained directly from the statistics of the parallel simulator. The cycle count of C62x was obtained by setting profiling points to the beginning and the end of the FFT application and then running the processor. Clock cycles elapsed between the profiling points were thus obtained. Information needed to evaluate the code size of MOVE processor was obtained from the simulator and the mapping file generated by the mapgenerator. Mapgenerator is one of the additional tools in MOVE framework. It creates a mapping file containing all the socket address tables and the structure of all the slots in the instruction. The width of the instruction, which is constant, can be obtained from this structure. The number of instructions required to perform the application was obtained from the statistics given by the parallel simulator. The code size in bits was then calculated by multiplying the number of instruction by the instruction width. Instruction size of the



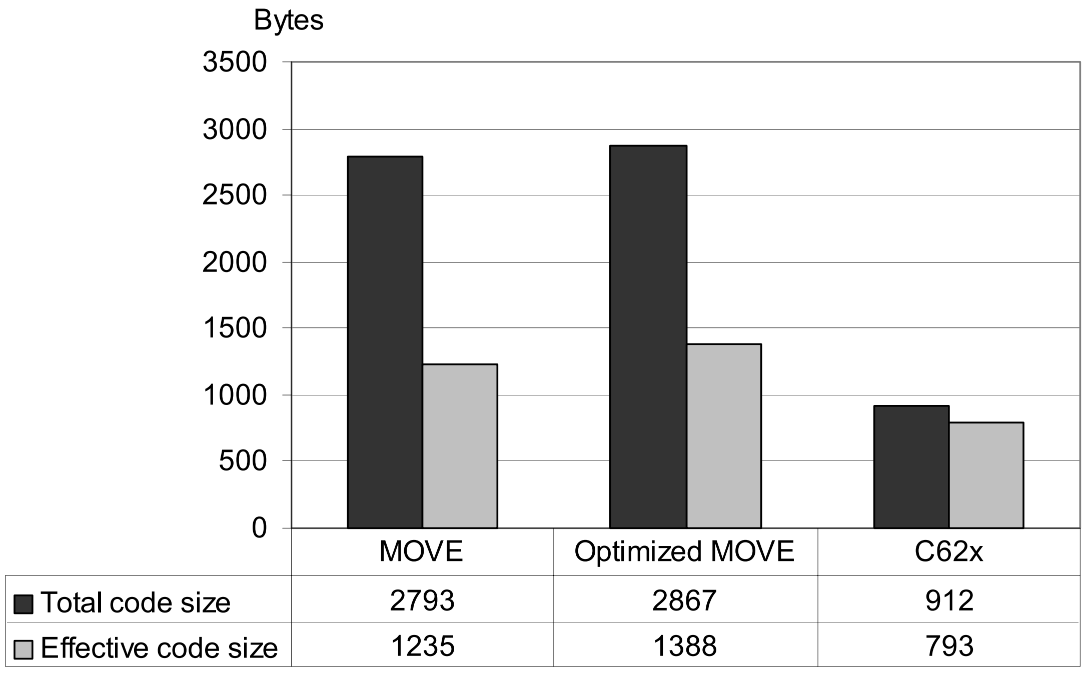
	MOVE	Optimized MOVE	C62x
Clock cycles	140,123	134,251	321,945

Figure 28. Cycle counts of 512-point complex FFT.

C62x was obtained from the simulator of the Code composer studio 1.2. From the disassembly code of the source C-code the number of bytes required for the FFT application was calculated by subtracting the start address of the FFT application from its end address. The code size in this case was obtained in bytes. To compare the code sizes, the code size of the MOVE processor design was divided by eight to represent it in bytes.

The cycle counts for both MOVE processor designs and C62x are shown in Figure 28. From the figure can be seen that the MOVE processor with corresponding architecture to C62x performs the FFT application in approximately 140,000 clock cycles whereas C62x requires more than two times that amount of clock cycles. With the optimized MOVE architecture even better performance is obtained. The difference in the results of MOVE and C62x can be explained by the fact that the compiler of MOVE framework is designed to be used for general-purpose applications whereas the compiler of C62x is designed to be used in DSP applications. The FFT application was written without much concern on the coding style and thus resembles more general-purpose application than DSP application. Due to this, MOVE performs FFT application in less clock cycles than C62x. The results are analyzed in more detail in section 4.5.

The code sizes of the binary executables are illustrated in Figure 29. The code size for C62x is approximately 900 bytes, which is approximately 3 times smaller than the code sizes of MOVE processor designs. As described in [15], programming data transports



	MOVE	Optimized MOVE	C62x
■ Total code size	2793	2867	912
□ Effective code size	1235	1388	793

Figure 29. Code sizes of 512-point FFT .

explicitly results in less dense code. For example, in C62x an operation is specified with 32 bits. In MOVE architecture, each move requires approximately 20 bits meaning than an operation consisting of three moves requires approximately 60 bits. In addition, the compiler of MOVE framework is not able to fulfill all the move slots in the instructions and thus many empty move slots occur. As can be seen from Figure 29, when only effective moves are considered and all empty moves removed, the code size is reduced to less than half of the total code size. This is discussed in more detail in section 4.5.

4.3 Discrete Cosine Transform Simulations

The performance of the processors in discrete cosine transform was evaluated by simulating three different DCT algorithms for the 2-D DCT: algorithms of Loeffler [16], Wang [17], and Kwak [18] depicted in Figure 30. Loeffler algorithm has the lowest number of multiplications in known DCT algorithms. Wang algorithm provides in-place algorithm often preferred in software implementations. The algorithm of Kwak offers an algorithm with regular kernel and constant geometry interconnections. 2-D transform was realized with row- column deposition, i.e., with the aid of two 1-D transforms separated by matrix transposition. This kind of DCT is popular in image processing.

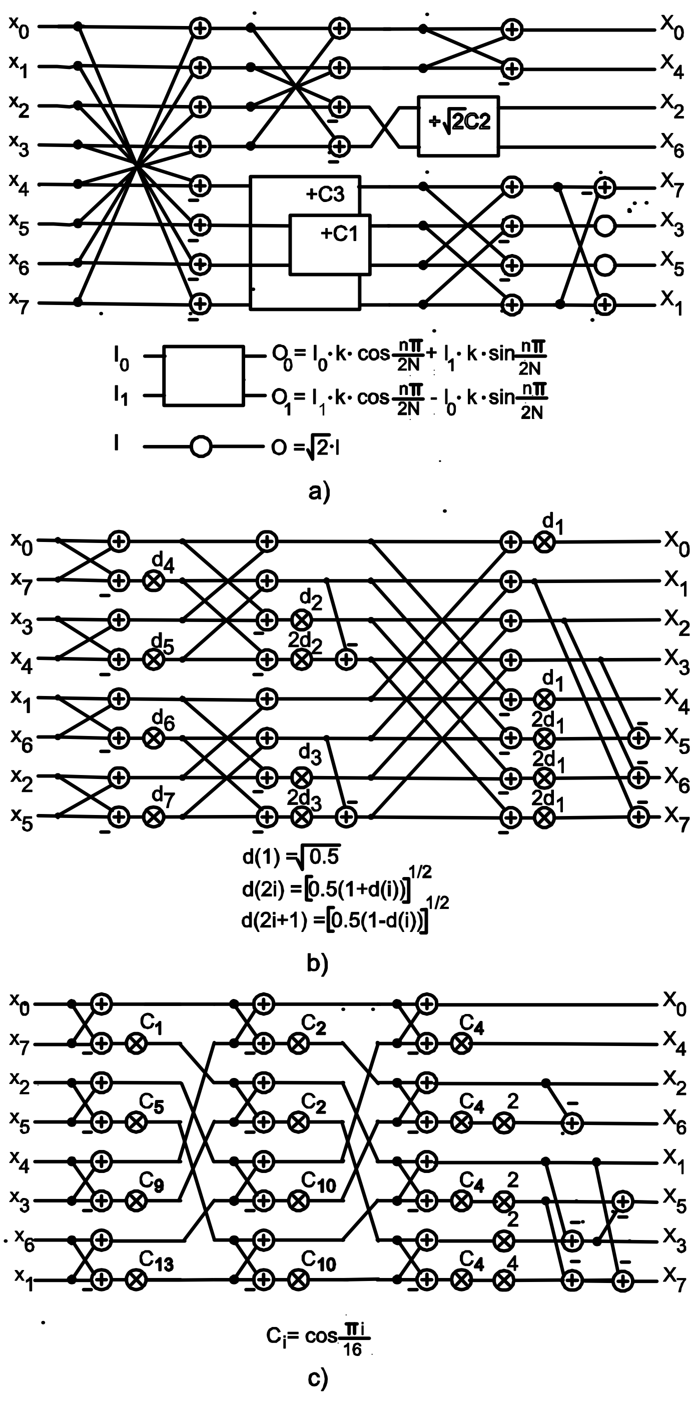
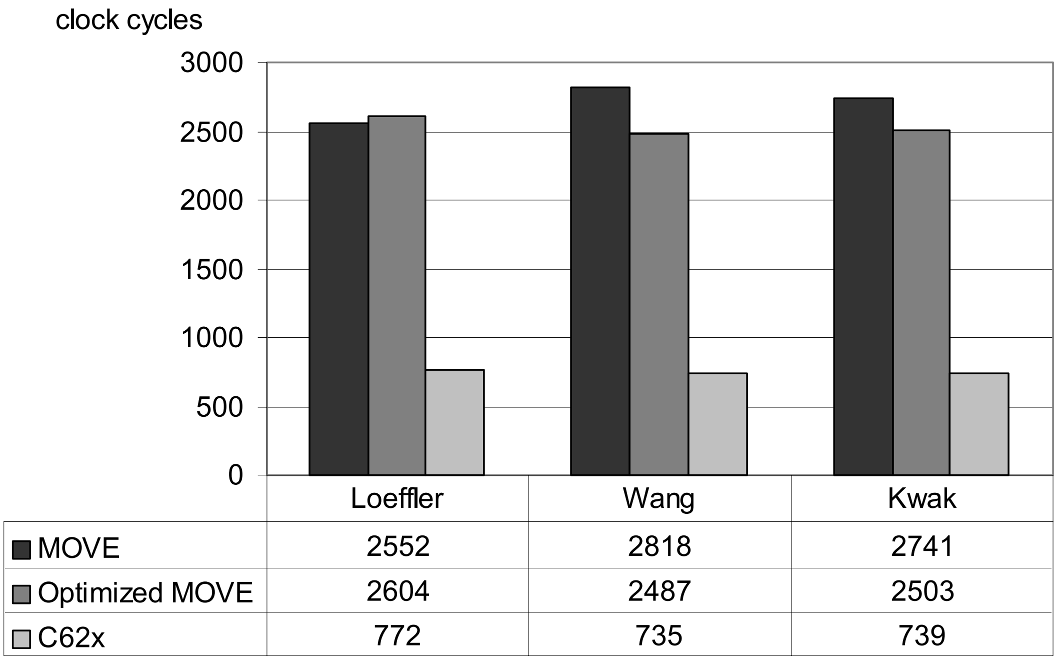


Figure 30. 2-D DCT algorithms: a) Loeffler, b) Wang, and c) Kwak.

The values of the input matrix are represented in 16- bit integer format since no floating-point FUs or registers were used in the architecture. Due to fixed-point representation, scaling of the intermediate values of the transform was required to avoid overflow. The results of multiplications were divided by 2^{15} to scale the intermediate results back to 16-bit format. This was done by shifting the values 15 bits to the right to avoid divide operation. After summations the values were divided by two, if necessary, to avoid overflow.

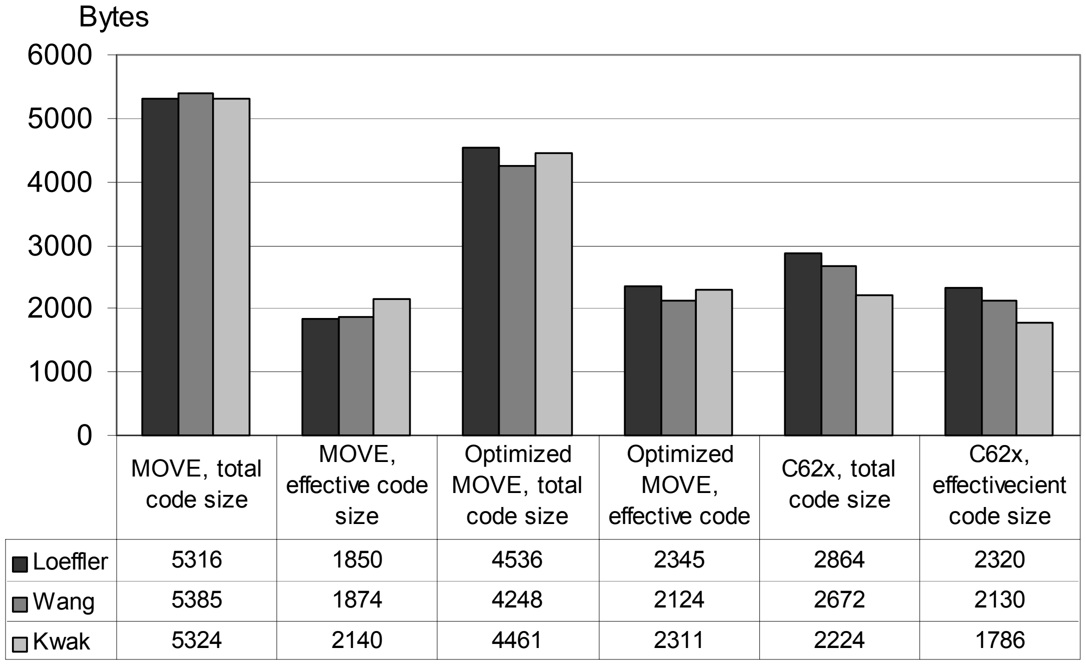


	Loeffler	Wang	Kwak
■ MOVE	2552	2818	2741
▨ Optimized MOVE	2604	2487	2503
□ C62x	772	735	739

Figure 31. Cycle counts of DCT applications.

The performance evaluations of DCT applications were performed similarly as for FFT application. Performance statistics of the MOVE processor were obtained from the simulator invoked in the scheduling phase of the compilation. Statistics of the performance of C62x were obtained by using the simulator of Code Composer Studio 1.2.

The number of clock cycles elapsed performing the three different DCT applications on MOVE and C62x processors are illustrated in Figure 31. The results are the opposite compared to the results obtained for FFT application. C62x outperforms both MOVE processor designs by a factor more than three. In the case of FFT, MOVE outperformed C62x with a factor of two. More effort was made to program the DCT applications as parallel as possible in order for the compilers to exploit the parallelism in the algorithm. According the results it seems that the compiler of C62x can find enough parallelism from the application to exploit software pipelining and other DSP optimization techniques, such as patter detection, and thus result in better and faster code. Software pipelining is implemented to the compiler of MOVE framework but it is not working correctly at the moment and thus was not used. More explanation to these performance differences between FFT and DCT applications is given in Section 4.5. Between the three DCT algorithms there is no major differences but the lowest cycle count for MOVE processor corresponding the C62x is obtained with the algorithm of Loeffler whereas lowest cycle



	MOVE, total code size	MOVE, effective code size	Optimized MOVE, total code size	Optimized MOVE, effective code	C62x, total code size	C62x, effectivecient code size
■ Loeffler	5316	1850	4536	2345	2864	2320
▨ Wang	5385	1874	4248	2124	2672	2130
□ Kwak	5324	2140	4461	2311	2224	1786

Figure 32. Code sizes of DCT applications.

count for optimized MOVE processor design is obtained with the algorithm of Wang. For C62x the algorithm of Wang produces the lowest cycle count.

The code sizes of the binary executables of DCT applications were also measured. illustrates The code sizes for MOVE and C62x processors are depicted in Figure 32. As in the case for FFT application, the code size of MOVE processor is bigger than the code size of C62x. When effective code size is considered, the code size of MOVE reduces considerably being in the case of Loeffler and Wang algorithms even lower than the effective code size of C62x.

4.4 Effect of Coding Style on the Performance

Applications for digital signal processors have traditionally been written with assembly language code. Applications have nowadays become more and more complicated and thus also more tedious to write with assembly code. Due to this the abstract level of writing applications has risen from assembly language code to high-level language code like C or C++. Compilers have been developed to compile high-level language code into assembly code. Compiling high-level language code sets high requirements for the compilers to take use of all the hardware resources as can be done by writing applications with assembly code. DSP-applications written with high-level language gain hardly ever the performance of applications written with assembly language code. The way the high-level application is written can have an effect on the effectiveness of the compilers. Compilers perform differently for different types of structures used, e.g., FOR loops and WHILE loops. Processor families have unique compilers, which perform differently for certain types of code structures used. This must be taken into account when writing applications for processors. However, every software programmer has his own individual style to write application code. Due to this, explicit guidelines are needed to guide the software programmers to write best possible code for the processor in question. If the compiler performs equally for different code structures and is thus resistive against different coding styles, no guidelines explaining how the code should be written are needed.

The effect of different coding styles on the performance of MOVE processor designs and C62x were evaluated. The FFT application used in the performance evaluations described in section 4.2 was used as a test application with the exception that instead of having input vector of 512 complex values the input vector consisted only of 32 complex values. Reducing the length of the transform made it possible to easily change the coding style of the application. MOVE architecture configurations for the 32-point FFT application were constructed similarly as for FFT and DCT applications in simulations described in sections 4.2 and 4.3. An architecture configuration corresponding to the architecture of C62x and an optimized architecture for 32-point FFT application were constructed and their performance simulated. The performance of these two MOVE processor designs and of C62x was then simulated. Five different C codes of the FFT application were simulated. The changes in the codes were quite small concentrating on loop structures. The explanation about the codes used are presented below:

Code A

The first version of the 32-point application was written without any attention to the coding style. The application code includes both FOR and WHILE loops with loop indices iterating downwards and upwards. Factors and data elements in the input sequence were accessed with the aid of pointers.

Code B

In code B, all the loops were transformed to FOR loops. Different loop structures are executed differently and thus it was decided to use only one type of loop structure. By using for-loops the compiler has the information of how many times a certain structure is executed and the performance can thus be possibly improved.

Code C

The C-compiler of TI C62x automatically changes the loops to iterate downwards regardless of the direction written in C-code. If loops are already written to iterate downwards, processor performance might possibly be improved. Thus the FOR loops in code B were written to iterate downwards.

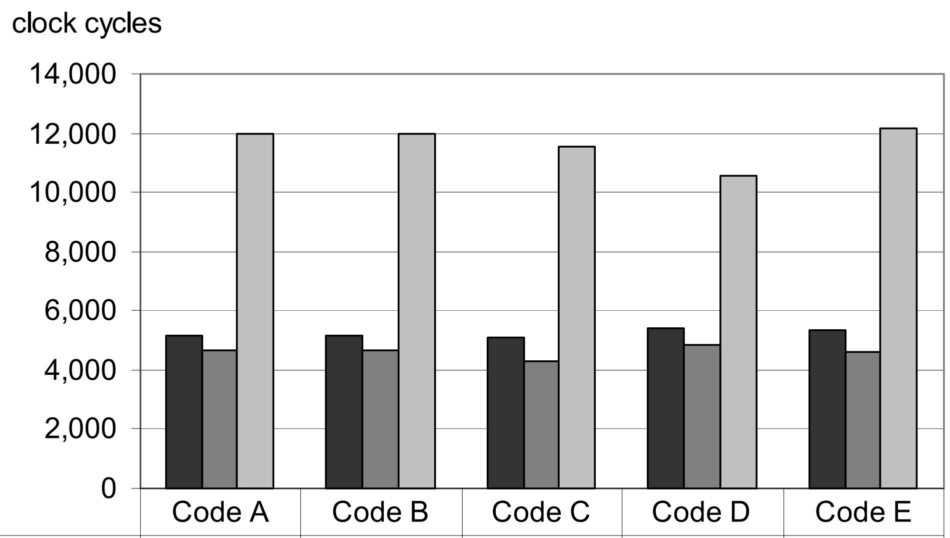
Code D

The code A included several nested loops. This kind of structure can result in inefficient code. The nested loops were removed from the code A, i.e., the code D was written with consecutive loop structures.

Code E

In addition, a special case was made investigate the use of pointers. In code E, data elements were accessed with indexing and all the pointers were removed.

Simulations for the different coding styles were performed similarly as for FFT and DCT simulations. Performance statistics of the MOVE processor were obtained from the parallel simulator invoked in the scheduling phase of the compilation. Statistics of C62x were obtained from the Code Composer Studio 1.2.

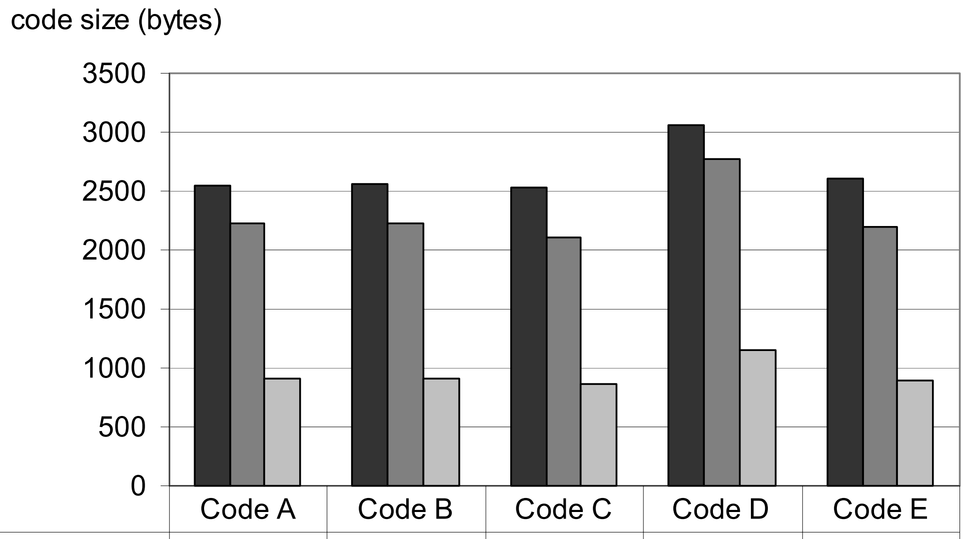


	Code A	Code B	Code C	Code D	Code E
■ MOVE	5,185	5,186	5,076	5,418	5,316
▨ Optimized MOVE	4,647	4,647	4,278	4,865	4,587
□ C62x	11,950	11,950	11,539	10,537	12,139

Figure 33. Cycle counts for 32-point FFT with different coding styles.

The number of clock cycles elapsed performing the 32-point FFT with different coding styles are shown Figure 33. The figure shows that coding style does not have a major effect on the number of clock cycles. For the MOVE processor design corresponding to C62x the maximum difference in the number of clock cycles with different coding styles is 7 percents and for the optimized MOVE processor it is 14 percents. For C62x the maximum difference is 15 percents. Best cycle count for both MOVE processor designs is obtained with code C, which has FOR loops iterating downwards. For C62x the best cycle count is obtained with code D, where the nested loops were changed to consecutive ones. As in the case of 512-FFT simulations the MOVE processor designs outperform C62x with a factor more than two.

The code sizes of the binary executables are depicted in Figure 34. The results show that the code size varies more than the number of clock cycles. The maximum variation in the code size for MOVE processor design corresponding to C62x is 21 percents and for optimized MOVE processor design 32 percents. In C62x, the maximum difference in the code size is 32 percents. Smallest code size for both MOVE processor designs is obtained with code C, as was the case with the lowest cycle count. For C62x the smallest code size is also obtained with code C. But as in the case of 512-point FFT simulations, the code sizes of MOVE processor designs are more than twice as big as the code sizes of C62x.



	Code A	Code B	Code C	Code D	Code E
■ MOVE	2538	2562	2528	3055	2599
▨ Optimized MOVE	2230	2230	2100	2780	2197
□ C62x	912	912	868	1144	900

Figure 34. Code sizes of 32-point FFT with different coding styles.

Simulations performed with different coding style show that the compilers of both C62x and MOVE framework are quite resistive against different coding styles when comparing different loop structures. The results also show that the compiler of MOVE framework seems to be a bit more resistive. The biggest variation in the clock cycles for MOVE processor design corresponding to C62x was 7 percents, when it was 15 percents for C62x. The same happened in the case of code size. The maximum variation of code sizes for MOVE processor design configured to match to C62x architecture was 21 percents when it was 32 percents for C62x. Optimized MOVE processor configuration results in more variation in the cycle count and the code size than the fixed MOVE processor design corresponding to C62x. Since the architecture configuration for each coding style in this case is optimized, each coding style takes the best use of the hardware resources and thus the variations in the results increase.

4.5 Analysis of the Results

The results of the simulations performed for FFT applications, both 512-point FFT and 32-point FFT with different coding styles, illustrate that MOVE is faster than C62x according to clock cycles. On the other hand, results obtained from the DCT simulations, show that C62x would be faster. However, the relationship between the code sizes of binary executables of MOVE and C62x is the same for both transformations. The code size of C62x in every application simulated was smaller. This section gives reasons why these kinds of results were obtained. Subsection 4.5.1 deals with the number of clock cycles and Subsection 4.5.2 with the code size of the binary executables.

4.5.1 Clock Cycles

Comparing FFT and DCT algorithms shows that the basic processing element in both of the algorithms is similar. The basic processing element performs a so-called butterfly for two inputs and stores results to two outputs. The first result is obtained by adding the second input to the first one. The second output is generated by subtracting the second input from the first and then multiplying the result of the subtraction by a factor.

Simulations were made to see how the compilers of C62x and MOVE framework succeed in compiling a loop, whose kernel performs the butterfly described above. The code of the loop kernel is given below:

```
for (j = 0 ; j < 64; j = j + 2)
{
     y[j] = x[j] + x[j+1];
     y[j+1] = (x[j] - x[j+1])*coeff[j];
}
```

The loop described above iterates 32 times. The architecture of the MOVE processor was configured to correspond to the resources of C62x. The resulting parallel move code of the compiler of MOVE framework and the assembly code generated by the compiler of C62x were examined. From the assembly codes it was noted that the assembly code of C62x performed two butterflies each iteration and thus the number of iterations was reduced to 16 with 7 clock cycles required for each iteration. This was accomplished by using software pipelining. With software pipelining the compiler detects the parallelism available in the loops to perform more operations in parallel if there is enough hardware resources available. Performing the butterfly requires two ALUs for addition and subtraction and one multiplier for multiplication. The C62x contains six ALUs and two multipliers, thus performing two butterflies simultaneously is possible. Software pipelining has been implemented to the compiler of the MOVE framework but it is not currently working properly and thus was not used. Due to this the compiler of MOVE framework could not exploit the parallelism available in the loops and only one butterfly during one iteration of the loop was performed. As a result, the kernel of the loop had to be performed 32 times, twice as many as for C62x. One iteration of the loop kernel required 9 clock cycles. As a result, MOVE processor requires 288 clock cycles to perform the kernel of the loop whereas C62x requires only 112 clock cycles. By writing the kernel of the loop to perform two butterflies resulted in same number of iterations with 8 clock cycles per iteration for MOVE and 7 clock cycles per iteration for C62x. Thus the number of clock cycles in this case is almost the same for C62x and MOVE.

Taking the previous into account, the contrary results of the FFT and DCT applications can be explained. The FFT application was written without much concern on the coding style and DSP optimization techniques, e.g., software pipelining and pattern detection could not be used. Since the compiler of MOVE framework is tailored for general-purpose applications, it was able to compile the FFT application better than the compiler of C62x and the number of clock cycles for MOVE processors turned out to be lower. More effort was put to write the three DCT applications and thus the compiler of C62x was able to use the software pipelining and other DSP optimization techniques thus resulting in better code and much lower cycle count than for MOVE processor. When the software pipelining was disabled form the compiler of C62x, the cycle counts of C62x increased considerably and the cycle count of MOVE processor designs became lower, approximately as much as in the FFT applications.

4.5.2 Code Density of Binary Executables

The code sizes of all the applications simulated were much smaller for C62x than for MOVE. In FFT applications, the code sizes of MOVE processors were approximately three times bigger and in DCT applications approximately two times bigger than the code sizes of C62x. In C62x, eight operations can be programmed in parallel thus meaning that 32 bits is required to perform one instruction. Examining the instruction format of MOVE processor designs showed that programming one move operation requires approximately 20 bits. If one operation corresponds to three moves, two operand moves and one result move, performing an operation in MOVE architecture requires approximately 60 bits, which is already twice as many as in C62x. Thus it is concluded that explicit specification of data transports results in less dense code. In addition, the scheduler is rarely capable of performing the maximum number of moves each cycle and thus empty slots occur wasting the 20 bits reserved. This also increases the code size.

The code density of the MOVE architecture could be improved by using variable instruction widths. This could be realized by replacing an empty move slot with 2-3 bits and a totally empty clock cycle with 4-5 bits. As the results in 512-point FFT and 2-D DCT applications showed, the effective code fraction of MOVE architecture is at maximum 50 percent of the total code size thus offering a possibility to reduce the code size dramatically by exploiting variable instruction widths. In addition, entropy encoding of the socket addresses could be used. In entropy encoding, the addresses of most often used sockets are defined small thus requiring fewer bits. Rarely used sockets get a bigger address requiring more bits. Since mainly the sockets with small addresses are used, the widths of the instructions decrease.

5. Special Function Unit Design

The modularity and flexibility of MOVE offer a possibility to specify user defined special function units (SFUs), whose operationality can be chosen freely. In MOVE architecture, implementing SFUs is fairly easy. Having a processor implementation with SFUs offers a possibility for better performance, since special hardware, for the needs of the application in question, can be generated.

The architecture of the SFU is defined in VHDL. The VHDL description of the SFU is taken into account in the MOVE processor generator and hardware for the SFU is generated in the processor design. In addition to the hardware of the SFU, an interface for the compiler must be generated to be able to compile HLL code for the MOVE processor containing SFUs.

In this chapter, design of a special function unit for discrete cosine transform (DCT-SFU) and adding it to the processor design is discussed. Sections 5.1 and 5.2 describe briefly specifications to be done in hardware and software subsystems in order to take the SFU into use. Section 5.3 describes the design of a special function unit for DCT and section 5.4 a MOVE processor design, which contains the designed DCT-SFU. Although the modifications required in the software subsystem to compile application code for a processor design including special function units are described in section 5.2, these modifications for the DCT-SFU were not made and thus no performance results were obtained. The compiler of the current version of the MOVE tools can support special function units with only two inputs and one output. Since the DCT-SFU has six inputs and four outputs, using the compiler to compile HLL application code for a processor design containing the DCT-SFU was not possible.

5.1 Special Function Unit in Hardware Subsystem

The move processor generator was described in Chapter 3. It takes as input the description of the processor configuration in files “processor_parameter.h” and “fu_def.h” and possible user-defined CHDL files containing the VHDL implementations of the SFUs. The name of the user-defined file must end with “fua.chdl”, e.g., “logfua.chdl”. The CHDL file contains a C-header in which the architecture of the function unit is described. The C-header must be pointed by a pointer in the function unit definition in file “fu_def.h”, that is to say, the name of the pointer must be the same as the name specified for this unit in the array “fudefs[ ]” in file “fu_def.h”. In the function body, the VHDL code is written between the “$[“ and “]$” characters. Architecture is written in VHDL. An example of user-defined CHDL file describing the architecture of an SFU performing logic operations is presented in the following:

```
FUA_HEAD(logfua)
{
$[
architecture arch of logfua is
begin
     process(T1,O1,T1opc)
     begin
          case T1 opc is
               when “00” => R1 <= T1 and O1;
               when “01” => R1 <= T1 or O1;
               when “10” => R1 <= T1 xor O1;
               when “11” => R1 <= T1 not O1;
               when others => R1 <= (others => ‘-‘);
          end case;
          end process;
end arch;
]$
}
```

In addition to the VHDL description, the SFU must also be defined in the tables “fudefs[ ]” and “fuinsts[ ]” in file fu_def.h. There it is defined similarly as the built-in function units. The number of sockets, number of id-bits, datawidths, connections to buses etc. must be defined for this SFU.

After writing the architecture of the special function unit and adding it to arrays “fudefs[ ]” and “funists[ ]” the MPG can be compiled and run. As a result a VHDL description of the processor including the special function unit is obtained.

5.2 Interface for Special Function Unit in Software Subsystem

The gcc-move compiler of the software subsystem cannot recognize operations of the SFU since these operations interface special hardware, namely the SFU. Due to this reserved functions must be configured to generate correct move assembly, which realize the data moves in and out from the SFU. User-defined instructions are then defined in C or C++ as calls to these reserved functions. These calls can be made, e.g., as functions or macros, which can then be called in the application code to use the SFU.

A header file “userdef.h” of the move include library contains all possible operation type formats with their equivalent assembly format. This is to say that for every operation type there is an operation format specifying the moves required to perform the operation. Operand types vary in the number of operand and result moves, e.g., add operation has two operand moves and one result move whereas jump operation has only one operand move. If the SFU has an operation format that is not specified in the “userdef.h” file, it must be added there. The operation format of the DCT-SFU design described in Section 5.3 contains six operand moves and four result moves. This kind of operation type format does not exist and thus it must be added.

After the operation type format has been created the functionality of the operation of the SFU must be implemented in a C function. The C function calls appropriate operation type with correct parameters when this C function is called in the application program. As a result, correct sequential move assembly is obtained. The C function, described above, must also contain C code that emulates the operation of the SFU. This is needed in case the application is compiled for the host machine, not for the target move machine. Distinction between these two machines is made through a definition *__move__*, which will be defined, if the target is the move machine.

Except for the pure functional description of the SFU, additional information must be provided for the scheduler and the simulator. This information is organized in files “userdef.c” and “userdef.h” located in the source directory of the scheduler. “userdef.c” contains all code needed for the scheduler and the simulator to operate correctly. In order to be able to use the operations of the SFU, an interface must be provided in the "userdef.c" between the functions of the scheduler and simulator and the user-defined operation.

To be able to use the design space explorer, cost of the SFU must be specified in a dedicated model file containing costs of all the hardware. The costs are defined in the model files simply by defining a value for the operation set of the SFU as has been done for the other operation sets also. Lastly the SFU must be added as an FU to machine description of the processor design.

5.3 Design of Special Function Unit for Discrete Cosine Transform

This chapter introduces, how a DCT can be adjusted for parallel calculation. Section 5.3.1 introduces a processing element, which can perform the whole discrete cosine transform. Section 5.3.2 describes how this processing element can be further modified to be used as a special function unit in a MOVE processor design.

5.3.1 Processing Element for Discrete Cosine Transform

In general, DCT algorithms do not have a regular structure like some FFT algorithms, e.g., Cooley-Tukey. However, suitable algorithms with constant topology interconnections can be obtained by regularization method proposed in [19]. Resulting signal flow graph of 8-point DCT is presented in Figure 35. The figure illustrates that there exist three processing columns and two permutation stages between them. By isolating the 4-input, 4-output

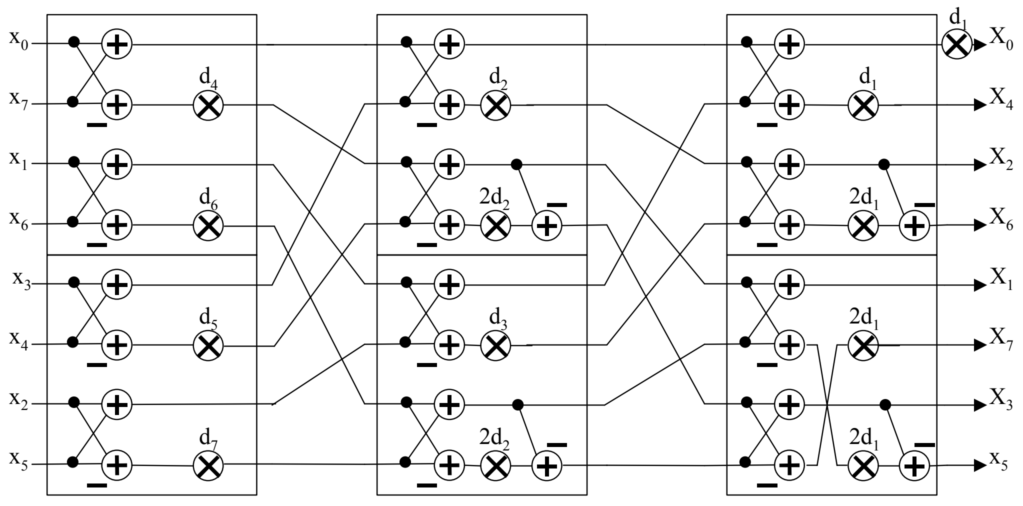


Figure 35. Signal flow graph of 1-D 8-point DCT.

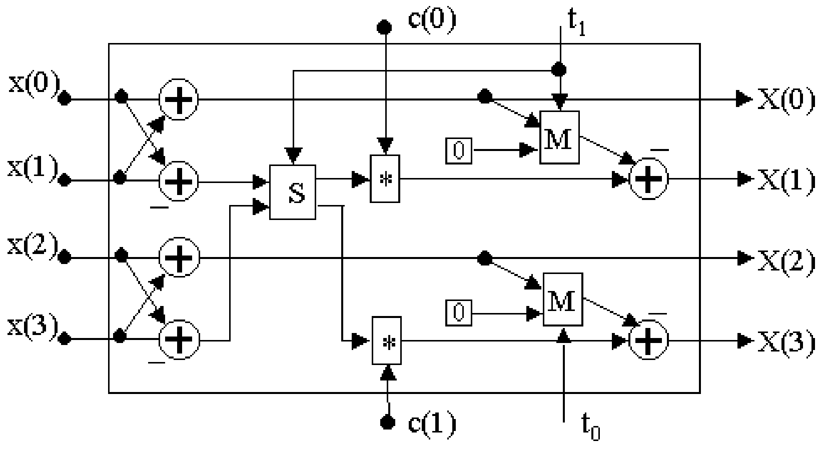


Figure 36. Processing element for DCT.

block at each processing column in the signal flow graph, it can be seen that there exist three different node functions. [20]

However, the node functions do not differ much from each other. Using parameterization, a single processing element can be constructed, which can calculate all three node functions and thus the transform can be calculated using only one processing element. The realization of this processing element is shown in Figure 36. The PE performs the calculation according to 2-bit control signal *t*, which controls the multiplexors and the switch. If *t* is "00", multiplexors select zero for the last summation and the switch does not cross the signals. When *t* is "01", the lower multiplexor selects the signal X(2) instead of

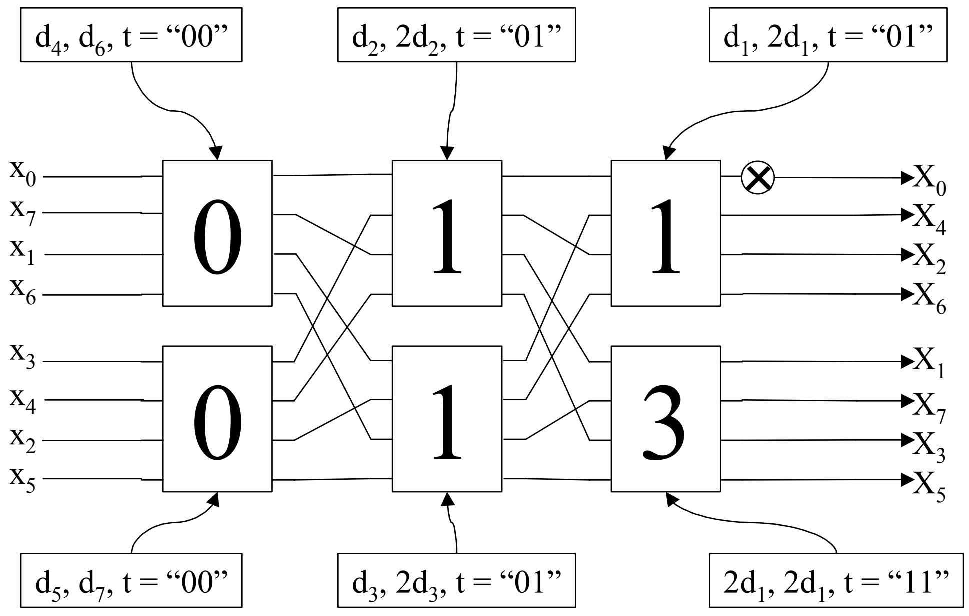


Figure 37. Signal flow graph of 1-D 8-point DCT with PEs.

zero. When *t* is “11”, the upper multiplexor selects signal X(0) and the lower multiplexor X(2). Also the switch crosses the signals to the multipliers. This way the PE can perform every node function of the processing columns of the transform. The DCT can now be performed using the processing element with different coefficients and values of *t* according to Figure 37.

5.3.2 Special Function Unit for Discrete Cosine Transform

In the previous Section 5.3.1, a processing element for DCT was presented. This processing element can be realized as a function unit with six inputs and four outputs. To make the function unit more usable and to operate faster, it is made pipelined in a way that each cycle a new operation can be started. Having a pipelined function unit means that intermediate stages (registers) must be added. These pipeline registers hold the values calculated in the previous combinatorial stage. In the following clock cycle, these values are fed as inputs to the next combinatorial stage.

The structure of the pipelined special function unit is depicted in Figure 38. The input values from *x(0)* to *x(3)*, coefficients *c0* and *c1* and the control signal *t* are first loaded to the input registers of the function unit. During the first clock cycle first summations of the input values and the switching according to signal t_1 are performed. The results are stored

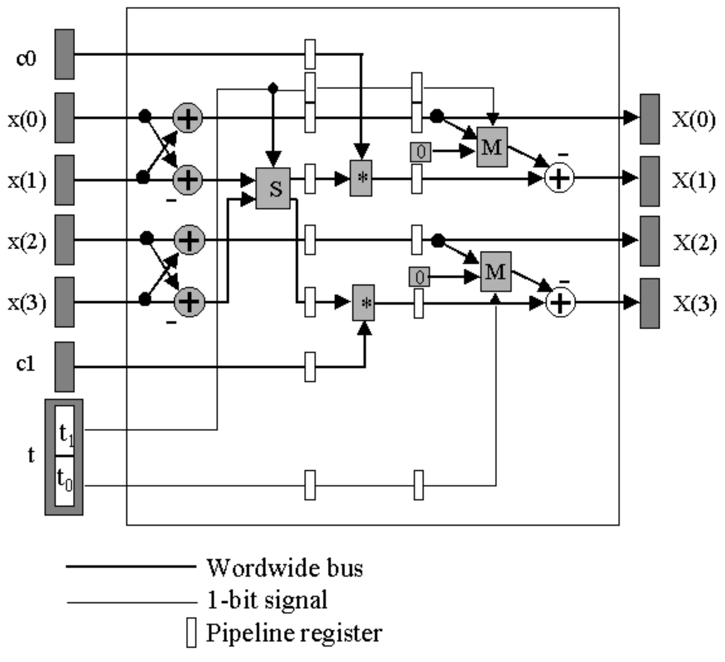


Figure 38. Pipelined special function unit.

to first intermediate stage registers to where also the coefficients and the control signals are stored. On the next cycle the multiplications are performed and the results and the control data are stored to second intermediate stage registers. During the third clock cycle multiplexing, according to signals t_0 and t_1, and last summations are performed. The results are stored to output registers. Since the function unit has two intermediate stages, the outputs are available in the output registers in a clock cycle i+2 if the operation was initialized in clock cycle i. The latency of the SFU is thus three.

5.4 MOVE Processor Implementation with DCT-SFU

A design of a special function unit for discrete cosine transform was explained in previous section. A MOVE processor design including the DCT-SFU was created to test the operationality of the hardware of the DCT-SFU. After the processor design was obtained, it was simulator using the vhdldbx simulator of Synopsys. Section 5.4.1 explains the implementation of the hardware of the DCT-SFU and adding it to a MOVE processor design. Section 5.4.2 explains the hardware simulations performed with Synopsys vhdldbx simulator program.

5.4.1 Adding the DCT-SFU to MOVE Processor Design

To add an SFU into a processor design requires that the architecture of the SFU is specified and the SFU is added to the architecture description of the processor, as was described in Section 5.1. The VHDL description of the processor is then generated according to the architecture description.

The VHDL code of the architecture of DCT-SFU was implemented in a CHDL file called “dctfua.chdl”, inside a C-function. “dct” at the beginning of the filename identify the name of the special function unit. The end of the filename, “fua.chdl”, identifies that the file describes architecture of an SFU. The CHDL file, “dctfua.chdl” is illustrated in appendix C. The DCT-SFU was specified to use 16-bit values as inputs, coefficients and outputs. After specifying the architecture of the DCT-SFU, it was added to the architecture description of the processor design. Required information of the DCT-SFU was added to the tables “fudefs[ ]” and “fuinsts[ ]” in file “fu_def.h” describing the processor configuration. In addition to the DCT-SFU, no other function units except the control unit and the load-store unit were added to the processor design. Since the DCT-SFU has six inputs, six move buses were defined on account to be able to transport all six operand values to the inputs of the DCT-SFU simultaneously and thus be able to start new operation each clock cycle. After the modifications to the input files of the MPG were made, MPG was run and the VHDL-code of the processor design was obtained.

5.4.2 Testing MOVE Processor Design with DCT-SFU

For the simulations a move-assembly test program was written. The move-assembly code specifies explicitly the data transports on each move bus each clock cycle. Each line of the assembly code specifies one instruction, which specify the data transport performed during one clock cycle. Each instruction begins with a memory address to where the instruction is to be stored. After the address the moves on all the buses are specified starting from the most significant bus number. Moves are separated with semicolons. After the move-assembly code is written, it is assembled into binary format with the assembler of the MPG. In addition to the move-assembly code, "assembler_info.txt" file is given for the assembler. The "assembler_info.txt" contains the parameters and the addresses of each source and destination IDs used in the move-assembler code. The binary code is then generated according to this information by replacing each source and destination ID with their corresponding address in binary format. The binary code thus contains all the instructions in binary format separated into memory locations. The binary code thus specifies the contents of the instruction memory. The vhdldbx simulator takes this binary file as input. When running the processor design in the simulator the instruction fetch unit fetches instructions from this virtual instruction memory. The processor then operates according to these instructions.

For simulating the DCT-SFU the move-assembly code was written. It was written to test all three possible operations of the SFU. The internal pipeline of the DCT-SFU was also tested by assigning six input values each cycle to the SFU in a way that new operation was started each cycle. After the latency of the SFU, the results were moved to register file. The assembly-code is presented in Appendix D.

The assembly code was then assembled and the vhdldbx simulator was run. The vhdldbx simulator shows the signals of the processor design in a waveform window. Part of the waveform window of the DCT-SFU simulation is depicted in Figure 39. From the signals shown in the waveform window the execution of the instructions can be followed. The *databus* signals show the contents of each move bus during the execution of each instruction. Figure 39 shows that during the first three cycles input values specified in the move-assembly code are transported on the move buses to the DCT-SFU. Since the

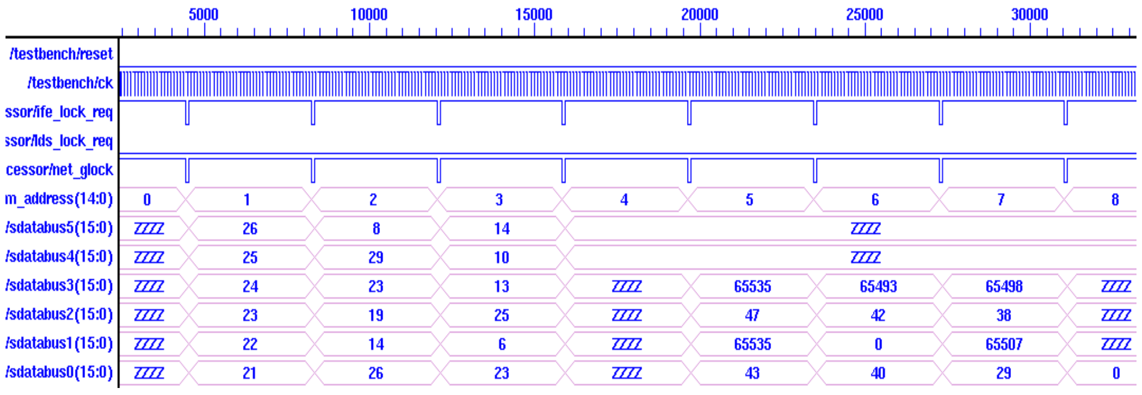


Figure 39. Simulation waveform.

DCT-SFU has latency of three, result values of operations issued in clock cycle i are obtained in clock cycle i+3, e.g., the results of the first operation issued in clock cycle one are obtained in clock cycle four. By comparing the moves specified in the move-assembly code and the realized moves in the waveform window the operationality of the DCT-SFU was verified to be correct.

6. Conclusions

The objective of this thesis was to evaluate the applicability of TTA in DSP applications. Applications realizing fast Fourier transform and discrete cosine transform algorithms were used in the evaluation. Furthermore, objective was to design a special function unit for DCT and to implement it on a processor design using the hardware subsystem of the MOVE framework. At first, background information of the concept of TTA and the MOVE framework, a tool set generated to design TTAs was given. Next the performance evaluations of FFT and DCT applications were explained and the results analyzed. Last, the implementation of a special function unit for DCT was discussed.

The performance of processors using TTA programming model, called MOVE architectures, was measured against a commercial DSP processor, Texas Instruments TMS320C62x DSP. The performance was measured as the number of clock cycles. In addition, the code density was compared. A more accurate performance measure would have been to evaluate the execution time instead of the number of clock cycles. Due to outdated technology to which the cycle time estimations of MOVE architecture are based on, evaluating the number of clock cycles was found more reasonable estimate of the performance.

The simulations performed showed inconsistent results. In FFT applications, the number of clock cycles for MOVE architecture was more than two times smaller than the number of clock cycles for C62x, but opposite results were obtained from DCT simulations. Reasons for this were found from the way the applications were written and how the two compilers work. The FFT application was written without putting much effort on writing the application. The DCT applications were written more carefully in order to let the compilers exploit software pipelining and other DSP techniques to produce efficient code. The compiler of MOVE framework is not designed for DSP applications, rather for general-purpose application. On the other hand, the compiler of C62x is designed specifically for DSP-applications. Since the FFT application was closer to general-purpose application than DSP application, the compiler of MOVE framework was able to compile it better than the compiler of C62x and MOVE processor executed the FFT application faster. In DCT application, the compiler of C62x was able to exploit software pipelining and other DSP techniques and thus resulted in faster execution. However, exploiting the fine-grain parallelism by explicitly programming the data transports between the function units and register files instead of programming operations offers a possibility for better performance by utilizing the hardware resources efficiently. The cycle count for MOVE architectures could be further reduced by reducing the latency of the load operation, since load operation could be performed in MOVE architecture with less delay cycles than was defined in the architecture configurations.

The code density of MOVE architecture was found poor compared to C62x. Programming data transports explicitly was found to be the reason for this. For example, C62x requires 32 bits per instruction whereas MOVE architecture requires approximately 60 bits per instruction. In addition, the efficiency of the binary executables was notably worse for TTA resulting in more empty moves. Effective code sizes of MOVE architecture are approximately 50 percent of the total code sizes. By exploiting variable instruction widths and entropy encoding this fact can be exploited and the code size of MOVE architecture binary executables can be reduced considerably.

Simulations performed to evaluate the resistivity against different coding styles showed that the compiler of MOVE framework was slightly more resistive than the compiler of C62x when loops structures were under consideration. However, as was found in the case of 512-point FFT and DCT simulations, the coding style can have a major effect on the performance. Since only loops were under consideration, dramatic changes in performance did not occur.

In addition to performance evaluations, special function units were discussed. An SFU for DCT was successfully implemented to a MOVE processor design. By utilizing SFUs the performance can be improved. It would have been interesting to evaluate the performance of MOVE architecture in DCT by using DCT-SFUs but due to limitations of the compiler of MOVE framework this could not be done and was left as future work.

The concept of TTA can be useful in DSP applications. Since the tools are still under development, all the features could not be used, e.g., software pipelining or utilization of SFUs. Due to this, there is still a lot of work to do with the tools before they can be used effectively in DSP applications. The work for exploiting software pipelining should be finished. To be used in DSP applications, the compiler of the MOVE framework should be modified to use DSP optimization techniques, e.g., pattern detection, for better compilation. In addition, concern should be made to produce more dense code. This could be done, e.g. by using variable instruction width and by using entropy encoding. Furthermore, the technology libraries should be updated to more recent ones. With these, the execution time could be directly compared instead of number of clock cycles.

Finally, it can be concluded that the results of the evaluations provide useful information about the applicability of MOVE architectures in DSP applications. No serious defects in the architecture were found though the MOVE framework is not a commercial product. It was found that MOVE compiler is insensitive to changes in coding style. It was also found that the compiler is not optimized for DSP applications. However, the architecture can provide comparable performance with the modifications proposed to the compiler. Code density of the architecture was found poor but it can be even better than the code density of C62x if advanced code compressions techniques are utilized. In addition, due to fact that MOVE processor can be tailored, one can also use entropy encoding, which improves the code density even further. As a conclusion, based on the results obtained in this thesis work, it can stated that MOVE framework is a promising design environment for designing cost-critical embedded system products used in DSP applications.

References

[1] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.

[2] P. Pirsch, *Architectures for Digital Signal Processing*, John Wiley & Sons, UK, 1998.

[3] A. L. DeCegama, *The Technology of Parallel Processor: Parallel Processing Architectures and VLSI Hardware*, vol.1, Prentice-Hall, NJ, U.S.A., 1989.

[4] H. Corporaal, M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 19-38, 1998.

[5] A. Capitanio, N. Dutt, A. Nicolau, "partitioned register files for VLIWs: a preliminary analysis of tradeoffs," in *Proceedings of IEEE Symposium on Microarchitecture*, 1992, pp. 292-300.

[6] J. Zalamea, J. Llosa, E. Ayguade, M. Valero, "Two-level hierarchical register file organization for VLIW processors", in *Proc. of IEE/ACM Symposium on Microarchitecture*, 2000, pp. 137-146.

[7] A. Smit, *The MPG manual*, M Sc. Thesis, Delft Univ. Tech., Delft, The Netherlands, Jul. 2000.

[8] A. Cilio, J. Janssen, "The move software framework," Technical Report, Delft Univ. Tech., Delft, The Netherlands, Jun. 2000.

[9] J. Hoogerbrugge, *Code Generation for Transport Triggered Architectures*, Dr. Tech. Thesis, Delft Univ. Tech., Delft, The Netherlands, Feb. 1996.

[10] A. Cilio, "Code generation for TTAs: the MOVE software framework", Seminar on Transport Triggered Architectures, Tampere, Finland, Oct. 2000.

[11] M. Arnold, *Instruction Set Extensions for Embedded Processors*, Dr. Tech. Thesis, Delft Univ. Tech., Delft, The Netherlands, Mar. 2001

[12] TMS320C6203 *fixed-point digital signal processor*, Data Sheet, Texas Instruments, Inc., Houston, TX, U.S.A., 2000.

[13] N. Seshan, “High velociTI processing,” *IEEE Signal Processing Magazine*, vol. 15, no. 2, pp. 86-101, 117, Mar. 1998.

[14] A. V. Oppenheim, R. W. Schafer, *Discrete-Time signal Processing*, Prentice Hall Int., Inc., Englewood Cliffs, NJ, U.S.A., 1989.

[15] H. Corporaal, “Transport triggered architectures examined for general purpose applications,” Sixth Workshop on Computer Systems, Delft, The Netherlands, Jan. 1993.

[16] C. Loeffler, A. Ligtenberg, G. S. Moschytz, “Practical fast 1-D DCT algorithm with 11 multiplications,” in *Proc. IEEE ICASSP*, vol. 2, pp. 988-991, Feb. 1989.

[17] Z. Wang, "Pruning the fast discrete cosine transform," *IEEE Trans. Communications*, vol. 39, no. 5, pp. 640-643, May 1991.

[18] J. Kwak, J. You, “One- and two-dimensional constant geometry fast cosine transform algorithms and architectures,” *IEEE Transactions on Signal Processing*, vol. 47, no. 7, pp. 2023-2034, Jul 1999.

[19] J. Takala, D. Akopian, J. Astola, J. Saarinen, “Simplified ‘architecture-oriented algorithms for discrete sine and cosine transform’,” *IEEE Trans. Signal Processing, 1999.*

[20] J. Takala*,* *Real-Time Digital Signal Processing Systems: Parallel Algorithms and Architectures*, Dr. Tech. Thesis, Tampere Univ. Tech., Tampere, Finland, 1999.

Appendices

Appendix A: Connectivity Optimized Architecture Configuration (mach-0616)

```
MoveBusses{
      m1          32, 8, signed;
      m2          32, 8, signed;
      m3          32, 8, signed;
      m4          32, 8, signed;
      m5          32, 8, signed;
      m6          32, 8, signed;
      m7          32, 8, signed;
      m8          32, 8, signed;
}

Sockets{
      fu1_o       input,      {    m2                        };
      fu1_t       input,      {                m5,         m8};
      fu1_r       output,     {                    m6,     m8};
      fu2_o       input,      {        m3, m4                };
      fu2_t       input,      {        m3,             m7    };
      fu2_r       output,     {                    m6, m7    };
      fu3_o       input,      {                m5            };
      fu3_t       input,      {        m3,         m6        };
      fu3_r       output,     {m1,                     m7, m8};
      fu4_o       input,      {    m2                        };
      fu4_t       input,      {            m4, m5, m6        };
      fu4_r       output,     {        m3,     m5            };
      fu5_o       input,      {                    m6,     m8};
      fu5_t       input,      {            m4,     m6, m7    };
      fu5_r       output,     {        m3, m4                };
      fu6_o       input,      {        m3, m4                };
      fu6_t       input,      {m1,             m5,         m8};
      fu6_r       output,     {    m2,         m5            };
      fu7_o       input,      {            m4, m5            };
      fu7_t       input,      {m1, m2                        };
      fu7_r       output,     {m1,                 m6        };
      fu8_o       input,      {                        m7    };
      fu8_t       input,      {        m3,             m7    };
      fu8_r       output,     {                m5,     m7, m8};
      fu9_o       input,      {                              };
      fu9_t       input,      {                              };
      fu9_r       output,     {                              };
      fu10_o      input,      {                              };
      fu10_t      input,      {                              };
      fu10_r      output,     {                              };
      ri_i1       input,      {m1,                         m8};
      ri_i2       input,      {                m5,         m8};
      ri_i3       input,      {            m4                };
      ri_i4       input,      {                m5            };
      ri_ii1      input,      {m1,                 m6        };
      ri_ii2      input,      {                              };
      ri_ii3      input,      {                              };
      ri_ii4      input,      {        m3,             m7    };
      ri_o1       output,     {                              };
      ri_o2       output,     {            m4,             m8};
      ri_o3       output,     {                        m7    };
      ri_o4       output,     {                m5,     m7    };
      ri_o5       output,     {                              };
      ri_o6       output,     {    m2,         m5            };
      ri_o7       output,     {        m3                    };
      ri_o8       output,     {    m2                        };
      ri_oo1      output,     {                              };
      ri_oo2      output,     {                    m6        };
      ri_oo3      output,     {                m5,     m7    };
      ri_oo4      output,     {    m2                        };
      ri_oo5      output,     {        m3                    };
      ri_oo6      output,     {            m4,             m8};
      ri_oo7      output,     {                            m8};
      ri_oo8      output,     {                              };
      b_i1        input,      {    m2                        };
      b_i2        input,      {            m4                };
      pc          input,      {m1, m2                       };
      ra          bidir,      {    m2                        };
      ir_1        output,     {m1                           };
      trap        input,      {                              };
}

FunctionUnits{
      fu1         always, 2, {fu1_o}, fu1_t, {fu1_r}, {mul};
      fu2         always, 2, {fu2_o}, fu2_t, {fu2_r}, {mul};
      fu3         always, 3, {fu3_o}, fu3_t, {fu3_r}, {ld, ldb, ldd,
                              ldh, lds, st, stb, std, sth, sts};
      fu4         always, 3, {fu4_o}, fu4_t, {fu4_r}, {ld, ldb, ldd,
                              ldh, lds, st, stb, std, sth, sts};
      fu5         always, 1, {fu5_o}, fu5_t, {fu5_r}, {add, sub, eq, gt,
                              gtu, and, ior, xor, sxbh, sxbw, sxhw, not,
                              min, max, abs};
      fu6         always, 1, {fu6_o}, fu6_t, {fu6_r}, {add, sub, eq, gt,
                              gtu, and, ior, xor, sxbh, sxbw, sxhw, not,
                              min, max, abs};
      fu7         always, 1, {fu7_o}, fu7_t, {fu7_r}, {add, sub, shl,
                              shr, shru, and, ior, xor, sxbh, sxbw,
                              sxhw, not};
      fu8         always, 1, {fu8_o}, fu8_t, {fu8_r}, {add, sub, shl,
                              shr, shru, and, ior, xor, sxbh, sxbw,
                              sxhw, not};
      fu9         always, 1, {fu9_o}, fu9_t, {fu9_r}, {add, sub, eq, gt,
                              gtu, and, ior, xor, sxbh, sxbw, sxhw, not,
                              min, max, abs};
      fu10        always, 1, {fu10_o}, fu10_t, {fu10_r}, {add, sub, eq,
                              gt, gtu, and, ior, xor, sxbh, sxbw, sxhw,
                              not, min, max, abs};
}
```

```
LongImmediate{
      Registers:
      i1          32, signed, ir_1;
      Control:
      i1 32:{8};
}

RegisterUnits{
      Integer     14, {ri_i1, ri_i2, ri_i3, ri_i4}, {ri_o1, ri_o2,
                       ri_o3, ri_o4, ri_o5, ri_o6, ri_o7, ri_o8};
      Integer     14, {ri_ii1, ri_ii2, ri_ii3, ri_ii4}, {ri_oo1, ri_oo2,
                       ri_oo3, ri_oo4, ri_oo5, ri_oo6, ri_oo7, ri_oo8};
      Boolean     4, {b_i1, b_i2};

}

InstructionUnit{
      JumpLatency 3;
      BoolLatency 1;
      BoolExprSize      1;
      ProgramCounter    pc;
      TrapRegister      trap;
      ReturnAddress     ra, ra;
}
```

Appendix B: Fully Optimized Architecture Configuration (mach-0245)

```
MoveBusses{
      m10         1, 8, signed;
      m11         32, 8, signed;
      m12         32, 8, signed;
      m13         32, 8, signed;
      m14         32, 8, signed;
      m15         32, 8, signed;
      m16         32, 8, signed;
}

Sockets{
      fu4_o       input,       {     m11,      m13,           m16};
      fu4_t       input,       {          m12,      m14, m15     };
      fu4_r       output,      {     m11, m12,      m14,      m16};
      fu12_o      input,       {     m11,                m15     };
      fu12_t      input,       {          m12,                m16};
      fu12_r      output,      {m10,                m14, m15     };
      fu16_o      input,       {m10,                m14, m15, m16};
      fu16_t      input,       {          m12,           m15, m16};
      fu16_r      output,      {     m11,      m13,      m15     };
      fu20_o      input,       {          m12, m13               };
      fu20_t      input,       {               m13,      m15     };
      fu20_r      output,      {          m12, m13,      m15     };
      ri_i1       input,       {                                 };
      ri_i2       input,       {                    m14          };
      ri_i3       input,       {          m12, m13, m14          };
      ri_i4       input,       {               m13               };
      ri_o7       output,      {m10, m11,      m13,      m15, m16};
      ri_o8       output,      {m10,      m12,      m14, m15     };
      ri_ii3      input,       {               m13, m14          };
      ri_ii4      input,       {     m11,                m15     };
      ri_oo1      output,      {                                 };
      ri_oo2      output,      {                    m14, m15     };
      ri_oo3      output,      {          m12                    };
      ri_oo4      output,      {                                 };
      ri_oo5      output,      {                                 };
      ri_oo6      output,      {                         m15     };
      ri_oo7      output,      {                              m16};
      ri_oo8      output,      {     m11,      m13               };
      b_i2        input,       {m10, m11                         };
      pc          input,       {     m11,      m13,           m16};
      ra          bidir,       {m10,                          m16};
      ir_1        output,      {m10,      m13                    };
      ir_2        output,      {     m11, m12                    };
      trap        input,       {m10                              };
}

FunctionUnits{
      fu20        always, 1, {fu20_o}, fu20_t, {fu20_r}, {ld, ldb, ldd,
                              ldh, lds, st, stb, std, sth, sts};
      fu16        always, 1, {fu16_o}, fu16_t, {fu16_r}, {add, sub, eq,
                              gt, gtu, and, ior, xor, sxbh, sxbw, sxhw,
                              not, min, max, abs};
      fu12        always, 1, {fu12_o}, fu12_t, {fu12_r}, {add, sub, shl,
                              shr, shru, and, ior, xor, sxbh, sxbw,
                              sxhw, not};
      fu4         always, 2, { fu4_o }, fu4_t, { fu4_r }, {mul};
}

LongImmediate{
      Registers:
      i1          32, signed, ir_1;
      i2          32, signed, ir_2;
      Control:
      i1 32:{7}, i2 32:{8};
}

RegisterUnits{
      Integer     16, {ri_i1, ri_i2, ri_i3, ri_i4},{ri_o7, ri_o8};
      Integer     16, {ri_ii3, ri_ii4}, {ri_oo1, ri_oo2, ri_oo3, ri_oo4,
                       ri_oo5, ri_oo6, ri_oo7, ri_oo8};
      Boolean     3, {b_i2};

}

InstructionUnit{
      JumpLatency       3;
      BoolLatency       1;
      BoolExprSize      1;
      ProgramCounter    pc;
      TrapRegister      trap;
      ReturnAddress     ra, ra;
}
```

Appendix C: Implementation of DCT-SFU (dctfua.chdl)

```
/*---------------------------------------------------------------------
DCT-block  FU architecture generation function.

BLock takes six inputs(4 samples and 2 coefficients). The transform
executed is dependent on the opcode.

The FUA_HEAD macro (defined in fua_head.h) will substitute the
required function header.  Always use this macro to ensure source
compatibility with future MPG versions.

A struct with FU parameters that you may want to use (Fu_par, see
g_types.h) is passed as argument 'p' to this function.  See CHDL
documention on how to use this.
---------------------------------------------------------------------*/
FUA_HEAD(dctfua)
{
$[

architecture arch of dctfua is

    signal sum1   : std_logic_vector(7 downto 0);
    signal sum2   : std_logic_vector(7 downto 0);
    signal sum3   : std_logic_vector(7 downto 0);
    signal sum4   : std_logic_vector(7 downto 0);
    signal coeff1 : std_logic_vector(7 downto 0);
    signal coeff2 : std_logic_vector(7 downto 0);
    signal opcode1: std_logic_vector(1 downto 0);
    signal opcode2: std_logic_vector(1 downto 0);
    signal mul1   : std_logic_vector(7 downto 0);
    signal mul2   : std_logic_vector(15 downto 0);
    signal mul3   : std_logic_vector(7 downto 0);
    signal mul4   : std_logic_vector(15 downto 0);

begin

  process(ck,T1, O1, O2, O3, O4, O5, T1opc, glock)

  begin

    if (ck'event and ck = '1') then

      if (glock = '0') then

        -- first stage
        coeff1 <= O5;
        coeff2 <= T1;
        opcode1 <= T1opc;

        sum1 <= conv_std_logic_vector(signed(O1) +
                                      signed(O2),sum1'length);
        sum3 <= conv_std_logic_vector(signed(O3) +
                                      signed(O4),sum3'length);

        case T1opc(1) is

          when '0' =>
            sum2 <= conv_std_logic_vector(signed(O1) -
                                          signed(O2),sum2'length);
            sum4 <= conv_std_logic_vector(signed(O3) -
                                          signed(O4),sum4'length);
          when '1' =>
            sum2 <= conv_std_logic_vector(signed(O3) -
                                          signed(O4),sum2'length);
            sum4 <= conv_std_logic_vector(signed(O1) -
                                          signed(O2),sum4'length);

          when others =>
            sum2 <= (others => 'X');
            sum4 <= (others => 'X');
        end case;

        -- second stage
        opcode2 <= opcode1;
        mul1 <= sum1;
        mul2 <=conv_std_logic_vector(signed(sum2)*
                                     signed(coeff1),mul2'length);
        mul3 <= sum3;
        mul4 <= conv_std_logic_vector(signed(sum4)*
                                      signed(coeff2),mul4'length);

        -- third stage
        case opcode2 is

          when "00" =>
            R1 <= mul1;
            R2 <= mul2(14 downto 7);
            R3 <= mul3;
            R4 <= mul4(14 downto 7);

          when "01" =>
            R1 <= mul1;
            R2 <= mul2(14 downto 7);
            R3 <= mul3;
            R4 <= conv_std_logic_vector(signed(mul4(14 downto 7))-
                                        signed(mul3),R4'length);

          when "11" =>
            R1 <= mul1;
            R2 <= conv_std_logic_vector(signed(mul2(14 downto 7))-
                                        signed(mul1),R2'length);
            R3 <= mul3;
            R4 <= conv_std_logic_vector(signed(mul4(14 downto 7))-
                                        signed(mul3),R4'length);
          when others =>

            R1 <= (others => 'X');
            R2 <= (others => 'X');
            R3 <= (others => 'X');
            R4 <= (others => 'X');

        end case;

      end if;

    end if;

  end process;

end arch;

]$
}
```

Appendix D: Move Assembly Code (dct_6bus.asm)

```
0      G0:T1fu10.opc0 <- #26s; G0:O5fu10 <- #25s;   G0:O4fu10 <- #24s;
       G0:O3fu10 <- #23s;      G0:O2fu10 <- #22s;   G0:O1fu10 <- #21s;

16     G0:T1fu10.opc1 <- #25s; G0:O5fu10 <- #22s;   G0:O4fu10 <- #23s;
       G0:O3fu10 <- #24s;      G0:O2fu10 <- #25s;   G0:O1fu10 <- #26s;

32     G0:T1fu10.opc3 <- #24s; G0:O5fu10 <- #24s;   G0:O4fu10 <- #28s;
       G0:O3fu10 <- #26s;      G0:O2fu10 <- #24s;   G0:O1fu10 <- #22s;

48     G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop <- Rnop;
       G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop <- Rnop;

64     G0:T1fu10.opc0 <- #23s; G0:O5fu10 <- #21s;   G0:O4fu10 <- R4fu10;
       G0:O3fu10 <- R3fu10;    G0:O2fu10 <- R2fu10; G0:O1fu10 <- R1fu10;

80     G0:T1fu10.opc1 <- #22s; G0:O5fu10 <- #23s;   G0:O4fu10 <- R4fu10;
       G0:O3fu10 <- R3fu10;    G0:O2fu10 <- R2fu10; G0:O1fu10 <- R1fu10;

96     G0:T1fu10.opc3 <- #21s; G0:O5fu10 <- #25s;   G0:O4fu10 <- R4fu10;
       G0:O3fu10 <- R3fu10;    G0:O2fu10 <- R2fu10; G0:O1fu10 <- R1fu10;

112    G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop <- Rnop;
       G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tjmp <- #0s;

128    G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop <- Rnop;
       G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop <- Rnop;

144    G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop <- Rnop;
       G0:Tnop <- Rnop;        G0:Tnop <- Rnop;     G0:Tnop  <- Rnop;
```