TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Electrical Engineering

JAAKKO SERTAMO

PROCESSOR GENERATOR FOR TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Subject approved by Department Council
20th Aug, 2003

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 2002-2003 as a part of the Flexible Design Methods for DSP Systems (FlexDSP) project funded by the National Technology Agency.

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 my workmates at institute of digital and computer systems for their company and assistance during the last two and half years I have been working at the institute.

Finally, I wish to thank my parents for their support throughout my studies.

Tampere, September 15, 2003

Jaakko Sertamo

Männikönkatu 3 A 12
33820 Tampere
p. 040 5937266
jaakko.sertamo@tut.fi

TABLE OF CONTENTS

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Degree Program in Electrical Engineering

Institute of Digital and Computer Systems

Sertamo, Jaakko Ilmari: Processor Generator for Transport Triggered Architectures

Master of Science Thesis: 73 pages, 5 appendix pages

Examiners: Prof. Jarmo Takala and Prof. Markku Kivikoski

Funding: The National Technology Agency

Department of Electrical Engineering

October 2003

Keywords: transport triggered architecture, electronic design automation

Application-specific instruction set processors can be used as building blocks of modern system-on-chips, increasing the design flexibility with programmability. Compared to fixed-processor cores, instruction-set optimized processors also provide significantly increased computational performance and energy efficiency. A wide range of architectural, software and implementation skills are necessary in the design of an application-specific CPU. The MOVE framework, a set of non-commercial software tools provide a design environment for fast semi-automatic design of custom processors. The MOVE framework consists of three components. Design space explorer automates the search of the optimal configuration of the processor for a given application. Hardware subsystem is responsible for estimating the cost of the processor configuration and generating hardware description language representation of the processor design. Software subsystem compiles high-level language application code to binary executables.

The MOVE framework utilizes the transport triggered architecture (TTA) as a processor template. TTA is VLIW-like instruction level parallelism processor architecture in which data transports between function units and register files are programmed explicitly instead of programming operations. Operations occur a as side effect of these explicit transports. Transport triggered architecture is simple, extremely scalable and flexible and therefore it is an attractive choice for embedded processors.

In this thesis, a processor generator for the MOVE framework that translates the high-level structural information of a target TTA processor into register transfer level VHDL description was designed. The designed processor generator improved the usability and reliability of the of original processor generator of the MOVE framework as the description of the target processor is obtained from specification files common to the rest of the tools of the MOVE framework. A clear interface specification and a functional unit

template for user-defined functional units was designed. An extensive library of integer functional units was designed utilizing the template.

Using the VHDL code obtained from the processor generator as a design entry, a large set of processor configurations were synthesized and evaluated. The estimated clock frequencies of the implemented processor designs were found comparable to the fastest synthesizable processor cores reported. Different transport bus demultiplexing structures were compared and AND-OR composition was found the most promising alternative to replace the tristate bus traditionally employed in TTA processors. Furthermore, the effect of clock gating was evaluated and significant reduction in power consumption was achieved.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Sähkötekniikan koulutusohjelma

Digitaali- ja tietokonetekniikan laitos

Sertamo, Jaakko Ilmari: Processor Generator for Transport Triggered Architectures

Diplomityö: 73 sivua, 5 liitesivua

Tarkastajat: Prof. Jarmo Takala ja Prof. Markku Kivikoski

Rahoitus: Teknologian kehittämiskeskus (TEKES)

Sähkötekniikan osasto

Lokakuu 2003

Avainsanat: transport triggered architecture, elektroniikan suunnitteluautomaatio

Sovelluskohtaisia käskykantaprosessoreita voidaan käyttää nykyaikaisten järjestelmäpiirien rakennusosina. Näin pystytään parantamaan järjestelmäsuunnittelun joustavuutta, koska piirin toimintaa on mahdollista jälkikäteen muuttaa ohjelmakoodia päivittämällä. Tämän lisäksi järjestelmän toiminnan kuvaaminen korkean tason ohjelmointikielillä on nopeampaa ja vähemmän virhealtista kuin perinteinen piirisuunnittelu. Valmiisiin suoritinytimiin verrattuna optimoidun käskykannan prosessorit tarjoavat huomattavasti paremman suorituskyvyn ja energiatehokkuuden. Räätälöityjen suorittimien suunnittelu vaatii kuitenkin paljon asiantuntemusta prosessoriarkkitehtuureista, ohjelmistokehityksestä ja laitteistosuunnittelusta, minkä vuoksi niiden käyttö ei ennen ollut realistista. Tätä ongelmaa helpottamaan on Hollannissa, Delftin teknillisessä yliopistossa, kehitetty MOVE framework. Se on joukko ei-kaupallisia suunnittelutyökaluja, jotka muodostavat suunnitteluympäristön sovelluskohtaisten suorittimien nopeaan, osittain automatisoituun suunnitteluun. MOVE framework koostuu kolmesta komponentista. Design space explorer etsii tietylle sovellukselle optimaalisen prosessorikonfiguraation. Hardware subsystem vastaa laitteistokustannusten arvioinnista ja laitteistokuvauskielisen kuvauksen generoinnista. Software subsystem kääntää ja optimoi korkealla tasolla kuvatun sovelluksen prosessorilla ajettavaksi binäärikoodiksi.

MOVE frameworkissa hyödynnetään transport triggered -suoritinarkkitehtuuria (TTA) suunnittelualustana, jota muuntelemalla sovelluskohtaiset suoritinytimet kehitetään. TTA suoritinarkkitehtuuri perustuu VLIW-arkkitehtuuriin ja kuuluu siten niin kutsuttuihin käskytason rinnakkaisuutta hyödyntäviin mikroprosessorityyppeihin. TTA-prosessorien ohjelmointimalli perustuu operaatioiden määrittelemisen asemesta siirtojen tarkkaan ohjelmointiin laskentayksiköiden välillä. Datansiirrot liipaisevat laskentaoperaation käynnistymisen ja siirron kohdeosoite määrää operaation tyypin. Tällainen ohjelmointimal-

li laskee konekielisen ohjelmoinnin abstraktiotason vielä perinteistä assembler-koodia matalammalle tasolle. Tästä seuraa, että konekielisen koodin kirjoitus käsin on hyvin vaivalloista, mutta toisaalta siirtojen tarkka ohjelmoitavuus antaa kehittyneelle aikatauluttavalle kääntäjälle mahdollisuuden käyttää hyvin kehittyneitä optimointimenetelmiä, jotta sovelluksen sisältämä rinnakkaisuus saataisiin mahdollisimman tehokkaasti hyödynnettyä. TTA-prosessorien perusrakenne on hyvin yksinkertainen. Prosessori muodostuu laskentayksiköistä ja rekisteritiedostoista, jotka liitetään toisiinsa kytkentäverkoston kautta. Kytkentäverkko koostuu väylistä ja laskentayksiköiden ja väylien välisistä kytkennöistä. Koska siirrot verkossa ovat ohjelmoitavissa, voidaan kytkentöjen määrä sovittaa sovelluksen vaatimusten mukaisiksi, mikä yksinkertaistaa prosessorin rakennetta. Laskentayksiköiden ja verkon liityntärajapinta on yksinkertainen ja säännöllinen. Siirtoverkon kapasiteetti, samoin kuin laskentayksiköiden määrä, on kasvatettavissa ilman, että kompleksisuus kasvaa räjähdysmäisesti. Näiden ominaisuuksien takia transport triggered -arkkitehtuuri on mielenkiintoinen vaihtoehto suurta laskentatehoa tarjoavaksi sovelluskohtaisten, sulautetuissa järjestelmissä käytettävien suorittimien alustaksi.

Tässä diplomityössä esitellään MOVE frameworkin kanssa käytettäväksi suunniteltu prosessorigeneraattori, joka muuntaa suunnitteluympäristön käyttämän yksinkertaisen korkean tason prosessorikuvauksen rekisterisiirtotasoiseksi, VHDL-kieliseksi esitykseksi. Uusi prosessorigeneraattori katsottiin tarpeelliseksi, koska MOVE frameworkiin kuuluva prosessorigeneraattori, MOVE processor generator, todettiin rajoittuneeksi sekä hankalaksi ja epäluotettavaksi käyttää. Suunniteltu prosessorigeneraattori haluttiin paremmin yhteensopivaksi muiden MOVE-suunnittelutyökalujen kanssa. Tärkein parannus tässä suhteessa oli, että prosessorin rakenne ja käskyjen koodaustapa luetaan samoista tiedostoista, joita muutkin MOVE-työkalut hyödyntävät. Nämä tiedostot voidaan usein luoda automaattisesti ja samalla niiden rakenteellinen ja sisällöllinen oikeellisuus tarkastetaan. Koska haluttiin selvittää, onko TTA-prosessoreissa perinteisesti käytössä ollut kolmitilapuskureihin perustuva väylä korvattavissa uusiin valmistusteknologioihin paremmin soveltuvalla ratkaisulla, lisättiin prosessorigeneraattoriin tuki myös AND-OR -rakenteeseen perustuvalle ja multiplekseripohjaiselle väylälle. Suorittimessa käytettäville laskentayksiköille määriteltiin yksityiskohtainen ja selkeä ulkoinen rajapinta ja prototyyppi. Niiden avulla käyttäjä pystyy suunnittelemaan omia laskentayksiköitään laitteistonkuvauskielien avulla ja hyödyntämään niitä kirjastokomponentteina prosessorigeneraattorilla suunniteltavissa prosessoreissa. Tämän prototyypin pohjalta kirjoitettiin kokonaislukuaritmetiikan tarjoavien laskentayksiköiden VHDL-mallit.

Seuraavaksi design space explorer -työkalulla luotiin kolmelle testisovellukselle joukko suoritinkonfiguraatioita, jotka sijoittuivat eri kohtiin kustannus-suorituskyky -tasoa. Kehitetyn prosessorigeneraattorin avulla luotiin näitä prosessorikonfiguraatioita vastaavat VHDL-kuvaukset. Simuloimalla VHDL-kuvauksia varmennettiin generoidun kuvauksen ja prosessorigeneraattorin oikea toiminta. TTA-prosessorien laitteistotason suorituskyvyn mittaamiseksi VHDL-kuvaukset syntesoitiin 0.13 μm:n vakiosoluteknologialle ja näin saaduista vetolistoista analysoitiin prosessorien suurin saavutettavissa oleva kellotaajuus, pinta-ala ja tehonkulutus. Havaittiin, että suunniteltujen suorittimien kellotaajuudet olivat vertailukelpoisia nopeimpien vakiosoluilla toteutettujen suorittimien kanssa. Lisäksi todennettiin, että kytkentäverkon yksinkertaistaminen parantaa merkittävästi TTA-suorittimen suorituskykyä ja energiatehokkuutta ja pienentää prosessoriyti-

men viemää pinta-alaa mikropiirillä. Seuraavaksi vertailtiin prosessorigeneraattorin tukemia väylärakenteita. Havaittiin, että AND-OR -rakenne oli lähes kaikissa testitapauksissa edullisin ratkaisu ja sitä voidaankin suositella jatkossa vakiosoluteknologioilla toteutettavien TTA-prosessorien väyläratkaisuksi. Lopuksi kokeiltiin kellojen portitusta TTA-prosessoreilla. Se vähentää suorittimen tehonkulutusta katkaisemalla kellosignaalin pääsyn rekistereihin niiden kellojaksojen aikana, joissa rekisteriä ei tarvitse päivittää. Havaittiin, että tällä menetelmällä TTA-prosessorien tehonkulutus pieneni merkittävästi.

LIST OF ABBREVIATIONS AND SYMBOLS

α	Constant reflecting the importance of cost
β	Constant reflecting the importance of performance
V_{DD}	Operating voltage
2-D	Two-Dimensional
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Set Processor
C62x	TMS320C62x
CHDL	A combination of C and VHDL code
CMOS	Complementary Metal-Oxide Semiconductor
DC	Decode stage
DCT	Discrete Cosine Transform
DSP	Digital Signal Processor or Digital Signal Processing
EX	Execute stage
FIR	Finite Impulse Response
FU	Function Unit or Functional Unit
GCC	GNU Compiler Collection
GNU	Gnu’s Not Unix
GPR	General-Purpose Register

HLL	High-Level Language
ID	Identifier
IF	Instruction Fetch stage
ILP	Instruction-Level Parallelism
MPG	MOVE Processor Generator
MV	Move stage
PC	Program Counter
RA	Return Address
RF	Register File
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register Transfer Level
SVTL	Semi Virtual Time Latching
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 complexity of system-on-chips for embedded wireless devices is increasing while, at the same time, greater flexibility of the implementation is being demanded. In addition, the cost penalty and probability of design errors in traditional application-specific integrated circuits (ASIC) are steadily rising. These factors are the key drivers for heterogeneous platforms consisting of programmable processor cores combined with dedicated hardware modules.

Heterogeneous platforms provide high computational performance for runtime-critical dataflow-dominated tasks, combined with high flexibility for complex mixed control and dataflow-oriented tasks. Furthermore, the software part of these platforms allows bug fixes and adaptations to changing requirements at low cost, as well as design reuse leverage for several product cycles.

Essentially, application-specific instruction set processors (ASIP) can be used as modules for platform-based design, increasing design flexibility with programmability. Compared to fixed processor cores, instruction-set optimized ASIPs also provide significantly increased computational performance and energy efficiency. Traditionally, such an approach would have been a last resort because of the range of architectural, software and implementation skills necessary to complete an application-specific CPU design. The emergence of design environments specifically for generating application-specific processors and their language tools makes the actual CPU creation considerably more attractive. One of such design tools is MOVE framework [1], a set non-commercial software tools aimed at computer aided design of application-specific processors.

The MOVE framework utilizes a subset of transport triggered architecture (TTA), a class of very long instruction word (VLIW) architecture as a flexible design template. TTAs have properties that are favorable as a template for application-specific processors. The modularity and regularity of TTA enables retargetable application code compilation and automatic hardware generation for a target TTA processor. The latter means essentially that it is possible transform the internal representation of a TTA processor, optimized for given application, into a general hardware description, which is independent of the

MOVE framework and TTA. Tool that performs this transform is called processor generator and the produced hardware description is typically hardware description language code, e.g., VHDL or Verilog.

In this thesis a streamlined processor generator to replace the existing processor generator in the MOVE framework is presented. In addition for better usability and improved interfacing to other components of the framework, the designed processor provides support for different realization of transport buses and a different hierarchical view to the structure of a TTA processor. Secondly, this thesis presents various implementation results, obtained through synthesis of the hardware description language code produced by the designed processor generator. The implementations were used to evaluate the hardware characteristics of TTA processors adopted to simple signal processing tasks, implemented in modern standard cell technology.

The structure of the thesis is the following. In Chapter 2, transport triggered concept is derived from the VLIW architecture and the hardware and software aspects of TTA processors are discussed. Moreover, two silicon implementations of TTA processors are presented.

Chapter 3 describes the MOVE framework and its components. Design space explorer, a tool for finding the best architecture configuration for a given application is described first. Secondly, software subsystem responsible for generating instruction level parallel object code for the processor designs is discussed. Hardware subsystem is responsible for estiamting the hardware cost of processor designs and generating VHDL description of the target processor. The latter is, more specifically, the task of the MOVE processor generator which is discussed and evaluated in more detail.

The new processor generator is described in Chapter 4. First, the requirements for a processor generator for TTAs and the MOVE framework are presented. Secondly, the implementation of the designed processor generator is presented. The main focus in this chapter is in the structure of the generated TTA core described in VHDL, rather than the internal software implementation of the processor generator. Lastly, the design principles of a functional unit library incorporated with the new processor generator are described.

Chapter 5 describes various implementation experiments of TTA processors designed using the MOVE framework and the new processor generator. At first the performance of the transport triggered architecture template of the MOVE framework is evaluated by analyzing a variety of different processor configurations optimized for small DSP benchmark cores. Moreover, the effect of connectivity optimization of the transport

network on the performance and cost of the processor cores is evaluated. In addition to performance analysis, three interconnection structures supported in processor generator were compared. Lastly, the results of applying clock gating on a number of processor designs are presented. Finally, Chapter 6 presents the summarized conclusions of this thesis.

2. TRANSPORT TRIGGERED ARCHITECTURES

The possibility to overlap the initiation and execution of multiple instructions within a single processor is referred to as instruction level parallelism (ILP). Exploitation of instruction level parallelism has been an important method for improving the performance of the latest microprocessors.

In superscalar processors such as Intel Pentium, processor hardware detects and resolves the dependencies between the operations in sequential instruction stream at runtime. This approach provides binary compatibility with previous architecture generations but suffers from hardware complexity and long design cycle, making superscalar architectures an unattractive choice from the embedded systems viewpoint.

VLIW processors issue only one instruction per cycle. Such an instruction typically consists of several RISC-like operations, such as addition, memory load, or jump. The operation dependencies, however, are managed at compile-time by a sophisticated compiler performing the instruction scheduling. The compiler for VLIWs has a very fine control over the machine resources and plays an important role in exploiting the resources and enhancing the performance. To achieve this goal, the compiler must use a detailed model of the target processor and keep track of the machine status and resource usage. Consequently, VLIWs are simpler, more flexible, and offer good scalability. Therefore, they are favorable candidates for low-cost high performance systems. Some notable commercial VLIW microprocessors are TMS320C6x of Texas Instruments and Trimedia of Philips.

Despite their good properties, the datapath of VLIWs may become too complex, in particular, when they are scaled to very high performance. Fortunately, the concept of replacing hardware complexity with compiler effort can be elaborated even further. This leads us to transport triggered architectures.

Section 2.1 discusses how TTA can be derived from VLIW architecture. In Section 2.2 the hardware aspects of TTAs are discussed. Section 2.3 describes the software aspects of TTAs. Finally, in Section 2.4 two TTA realizations are presented.

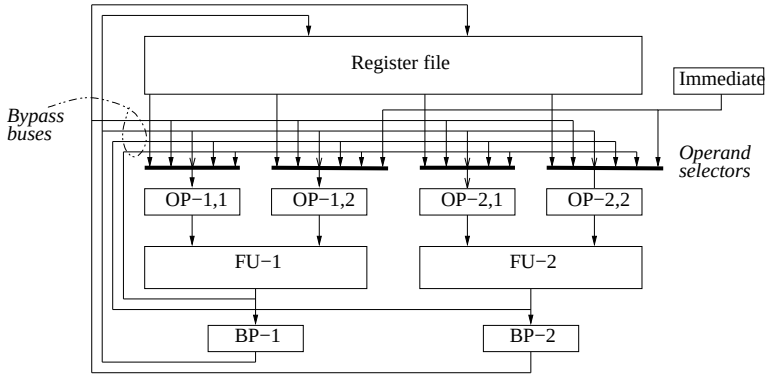

Figure 1. *Possible data path of VLIW processors with two functional units*

2.1 From VLIW to TTA

The datapath of a VLIW processor consists of functional units (FU), which are connected to a register file (RF) through a bypassing network as illustrated in Fig. 1. The bypass network is required to forward a result from the output of a functional unit to the input latch of another unit when a read-after-write data hazard occurs on a pipelined machine. As can be seen, bypassing the values from all FU outputs to all FU inputs requires crossbar connectivity, which results in quadratically growing complexity with number of functional units. However, the full bandwidth of this network is seldom utilized, not even when all the units are busy. [2]

The register file complexity may also become a bottleneck in VLIW processors. For each functional unit two read ports and one write port are required. This is only for the worst case situation when each FU needs to perform two reads and one write on the RF. In majority of VLIW realizations there are four or less functional units connected to a single register file. Unfortunately the performance tends to degrade when the number of ports is very high. Clustered VLIW architectures have been introduced to lower the requirements for the register file and thus improve scalability. [3] [4]

Transport triggered architecture was developed to reduce the complexity of VLIW by placing the register traffic under program control. In other words, the data transports become visible at the architectural level and they can be controlled and optimized by the compiler. TTAs are organized as a set of FUs and register files, which are connected by an interconnection network. Some functional units may have connections outside of CPU, e.g., to memory. This organization is illustrated in Fig. 2.

TTAs remind VLIW architectures in that they can perform multiple operations per cycle. The principal difference is the way, in which operations are programmed and executed.

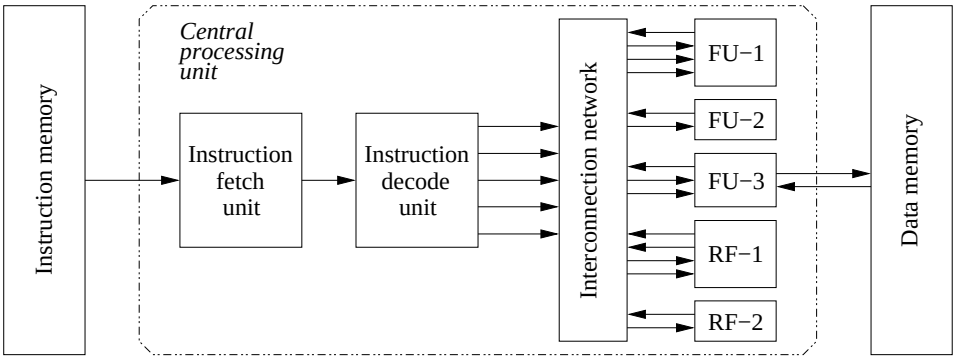

Figure 2. *Organization of TTA*

In VLIWs, instructions specify RISC type operations, while in TTAs, they specify data transports or moves. Operations are triggered as a side effect of these data transports: the destination of a transport implicitly specifies the operation performed on the data.

2.2 Hardware Aspects

TTAs are constructed using a restricted number of building blocks. Essentially, TTAs are built by a proper connection of FUs and transport/interconnection network. Register files can be considered as special kind of FUs (implementing the identity operation). FUs are completely independent of each other and interconnection network; they only need to realize the network interface. FUs can, therefore, be designed separately, pipelined independently, and they can support any type of operations. The modularity of TTAs allows the hardware design process to be automated. Different TTAs can easily be configured by assembling different combinations of such blocks. [1]

2.2.1 Interconnection Network

The interconnection network allows FUs and RFs to exchange data. There are two simple components to the interconnection network: buses and sockets. Buses not only provide data transport capability but they also perform the distribution of the signals that control the transports: source and destination register IDs and signals for locking the processor.

A TTA instruction for a processor with *M* buses, depicted in Fig. 3, typically consists of *M* fields, which each specify an independent, concurrent transport from a source to a destination. Source and destination refer to a functional unit output and input.

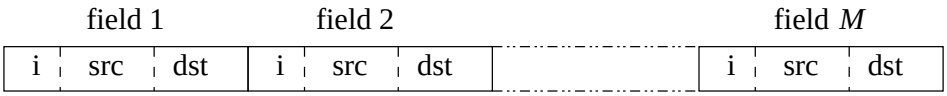

Figure 3. *General instruction format for TTA processor*

The interface between buses and functional units is provided by sockets, which implement programmable connections between functional units and buses. Each socket is connected to one or more buses and to one or more registers of one functional unit. Input sockets are basically input multiplexers, which select a value on one of the buses and write it into the destination register. Output sockets are output demultiplexers and load the contents of the source register into one or more of the connected buses. Source and destination fields are sent through the control path of the buses into all the connected sockets. The socket connected to a requested register is activated by the register-id and passes the data in the wanted direction. An interconnection network composed of three input sockets and two output sockets is illustrated in Fig. 4.

A fully connected interconnection network, where each socket is connected to all the buses, simplifies the task of assigning transports to buses. However, this is not an efficient design because connections increase the capacitive load on the bus, lengthen the

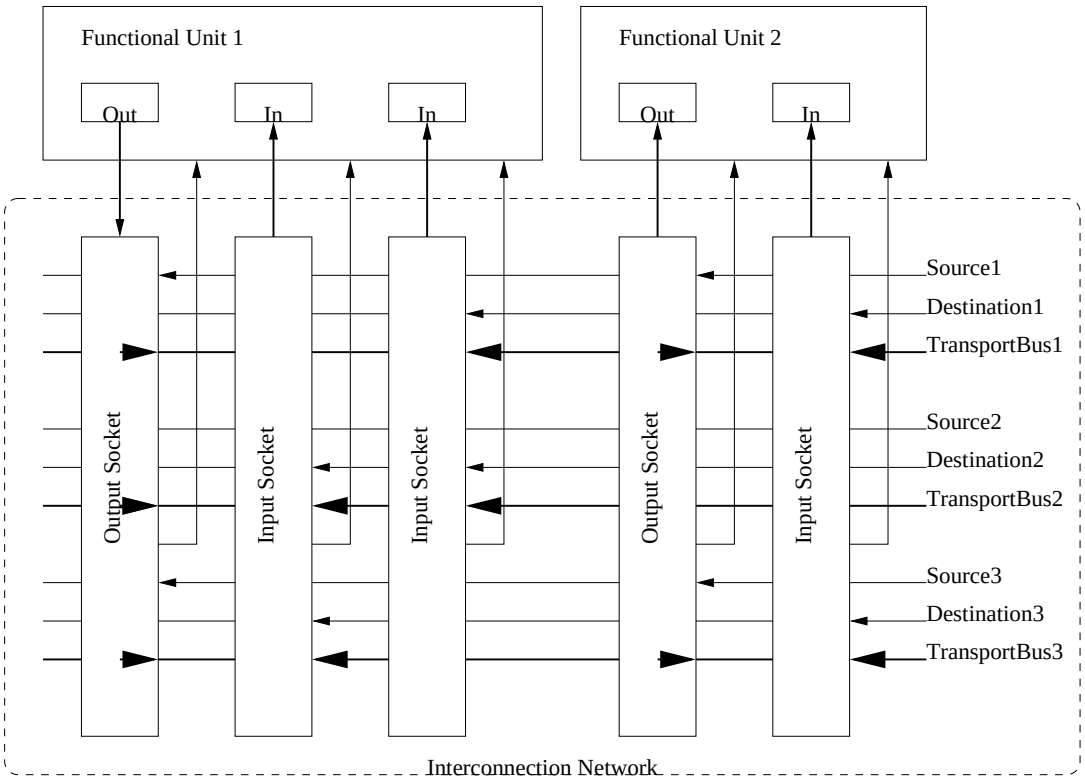

Figure 4. *Interconnection network*

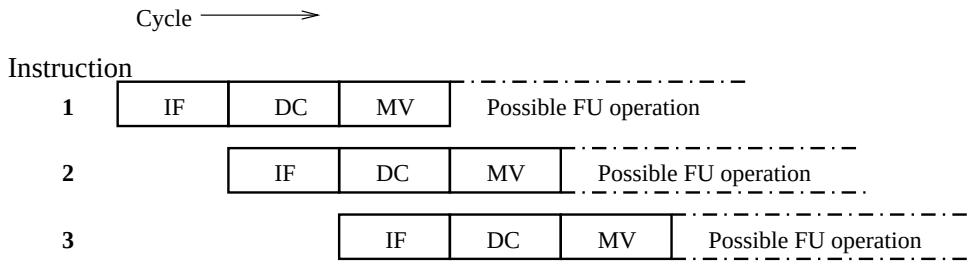

Figure 5. *Three-stage transport pipeline. IF: Instruction fetch. DC: Instruction decode. MV: Move stage.*

overall cycle time, and increase the power consumption. In ASIPs, the connectivity of the interconnection network should match the communication requirements of the applications and the cost constraints. Since the transports are explicitly programmed, a TTA compiler can take care of routing the data transports across a partially connected network effectively.

2.2.2 Transport Pipelining

The execution of instructions can be efficiently pipelined in TTAs. This is called transport pipelining. Functional unit pipelining, described in Section 2.2.3, is also supported and can be designed independently of transport pipelining. Typically data transports are executed using a three stage pipelining mechanism, which consists of instruction fetch, decode, and move stages. Fig. 5 illustrates this pipelining scheme from the instruction stream point of view. The decode and move stages can be combined for two stage pipelining.

During the instruction fetch stage instruction memory or cache is accessed for reading the next instruction. This access takes one cycle. In the decode stage, the source and destinations fields are extracted from the instruction word and forwarded to sockets, which activates the data transport to functional units. The actual data transport takes place in the move stage in which data is copied from output of a functional unit to input register of another FU. Register file read/write and bypassing data values between FUs are performed in the move stage. The operations have to be programmed explicitly.

2.2.3 Functional Units and Register Files

FUs are the components that perform the computation and communicate with the external environment. Units that are usually present in a TTA processor are instruction fetch,

guard generation, and load-store units. The instruction fetch unit reads instructions from the memory and to controls the flow of the program. This is the only unit that can write the control path of the processor. A load-store unit provides an access to data memory, external to CPU, in which variables with long lifetimes can be stored.

A functional unit contains one or more input and output (result) registers. Furthermore, the input registers can be distinguished as trigger and operand registers. The operand registers provide storage for the input operands of the FU. Trigger registers provide storage for the input operands but also fulfill two other important functions. First of all, a transport to trigger register initiates (or triggers) a new operation. Second, if an FU supports more than one operation, an opcode, which selects the operation to be performed, is received from the socket concurrently with data to be latched to the trigger registers. Hence, functional unit input registers have to be identified by the compiler. A programmed transport to operand and trigger register are called an operand move and trigger move, respectively. Similarly, a transport that reads the output of the FU is result move for a given functional unit.

Supporting more inputs and outputs requires multiple instruction formats, which significantly complicates the instruction decoding hardware of simple RISC instruction set architectures. TTAs are much more flexible in this respect. Each operand is encoded separately and written independently, thus there is no constraints on the number of operands that are used by an operation. This characteristic is especially useful in the context of ASIP design because it allows to add custom operations to be created with any number of input and output operands, without having to modify the general organization of the instruction decoder. [5]

When execution of an operation takes more than one machine cycle, it may be subject to pipelining. The execution stages are local to each FU pipeline and independent from other FU pipelines. In [2], several pipeline control disciplines are presented, which determine when pipeline latches are allowed to accept new data. Hybrid and virtual-time latching are two commonly used latching methods.

FUs employing virtual-time latching (VTL) run synchronously to the instruction stream as specified by the compiler. Each time an instruction is issued, FU pipeline progresses one step. Assuming that the functional unit is fully pipelined, i.e., the number of pipeline stages are equal to its latency, stages accept new data each time an instruction is issued. It is the responsibility of the compiler to ensure that no data is unintentionally overwritten. Pipeline is locked only when a stall occurs, e.g., for a cache miss in instruction fetch. Two different versions of virtual-time latching can be characterized, which differ

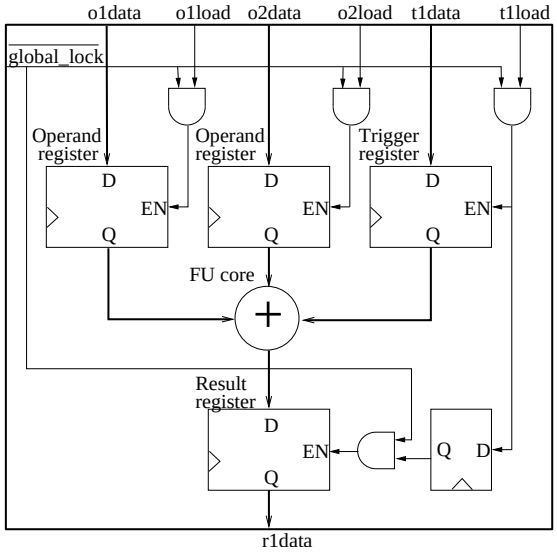

Figure 6. *3-input adder unit utilizing SVTL pipelining discipline*

on how the transports to operand registers are treated.

For true virtual-time latching (TVTL), move to any of the input registers will start a new operation. This makes hardware simple but limits the scheduling freedom of the compiler. In semi virtual-time latching (SVTL), only a trigger move will activate FU operation. For each operand register, a shadow register has to be added to temporarily store a pre-fetched operand value. However, the shadow registers can be omitted if FU has extra register stages in addition to trigger and operand registers. Fig. 6 depicts a 3-input adder unit utilizing SVTL pipelining. The input registers latch the data from the corresponding input socket if load signal is active, indicating incoming data from the socket. FU core, in this particular case a 3-input adder, performs the operation on the latched values of two operand and one trigger register. The result register is updated if control signal `t1load` was active on the previous clock cycle. The latency of the adder unit is thus two clock cycles meaning that the result can be read from the result register two clock cycles after the operation was triggered. The result value remains available until it is overwritten by the result of another operation. Since the functional unit is pipelined, new operations can be triggered every clock cycle. When `global_lock` is active the FU pipeline is halted.

A hybrid pipeline latches data as long as the pipeline register does not contain valid data from previously started operation, which is not proceeding in the pipeline. This is accomplished by attaching a stage controller to each pipeline registers of the FU. The last pipeline stage proceeds only when functional unit is accessed by a result move. For this reason FU pipeline has to be flushed if speculative operations have been triggered.

In [6], the two pipelining alternatives were compared in terms of clock cycle count. Even though the hybrid pipelining offers greater scheduling freedom, compared to virtual-time-latching pipelines, the required flush moves degrade its performance when speculative execution is applied. No significant scheduling advantage for the hybrid pipelined TTA was found for a given set of benchmark applications. Therefore, VTL pipeline is, in general, preferred due to its simple control logic.

Regardless of their programming methodology, the program controlled register traffic, also TTA microprocessors require general-purpose registers (GPR) to store the intermediate values with short lifetimes. The GPRs are arranged as one or more register files, which are connected to the interconnection network via input and output sockets just like ordinary FUs. In TTAs, the number of register file ports can be reduced significantly in comparison to VLIWs. Moreover, the GPRs and the register file ports can be efficiently partitioned into multiple register files partitions without notable degradation in performance. [7] [8]

2.3 Software Aspects

For traditional operation triggered architectures (OTA), such as RISCs and VLIWs, the executable program consists of an ordered set of operations which are performed by the processor. As has been described, transport triggered architectures are programmed by specifying data transports between the functional units and register files through an interconnection network. For this reason only one type of operation, move, is required. Hence, TTAs are also called MOVE architectures and their realizations MOVE processors.

To illustrate TTA programming, a correspondence between RISC-type add operation and data transports of a MOVE processor can be illustrated as

```
                  r1 -> add.o;
add r3,r2,r1  ⇒   r2 -> add.t
                  add.r -> r3;
```

The destinations add.o and add.t are the operand and trigger registers of the adder, and add.r denotes the result (output) of this adder. TTAs can easily be made fully programmable. This requires support for control flow operations and conditional execution. Control flow operations, such as jumps and calls, can be implemented by making the program counter a visible source and destination location of the instruction fetch unit. Conditional execution can be supported by guarding move operations; all data

transports become then conditional on a boolean expression.

The TTA programming method may first seem clumsy. Instead of one dyadic FU operation three moves have to be specified. However, this approach provides opportunities for many compile-time optimizations that are not available for the traditional architectures. Because all transports can now be controlled by the programmer or compiler, unnecessary transports, which occur quite frequently in current architectures, can be avoided. This is the key to solving the scalability problem of VLIWs. [2]

Programming and optimizing code for transport triggered processors is significantly more complex compared to OTAs. It would be extremely slow and error prone to perform it by hand. Due to this, applications should be described in high-level languages (HLL), such as C or Java. A compiler transforms the high level code to data transports between hardware resources, and an instruction scheduler optimizes the code trying to minimize the execution time and code size. Details of code generation and optimization are discussed in [6] and [9]. A software toolset for code generation is described in chapter 3.

2.4 Realizations

A few prototype implementations have been designed and manufactured. Two of such experimental processors are presented in the next sections.

2.4.1 32-bit General-Purpose Processor

In order to evaluate the specific design and implementation tradeoffs, an instance of a transport triggered architecture, called MOVE32INT, was designed at Delft University of Technology. [10] The architecture mainly consists of a transport network, controlled by the network controller, and several functional units. The network contains 4 busses. Each bus contains a data bus, which is capable of transporting one data value of 32 bits, an ID-bus, which transports one move operation specifier of 16 bits (as shown in Fig. 8), and a control bus, which contains a few control signals. This means that four data transports (moves) can be handled in parallel.

Fig. 7 shows the operational view of MOVE32INT, including operand (O), trigger (T) and result registers (R). As indicated, the processor uses a Harvard architecture, i.e., there are separate memory interfaces for the instructions and data. MOVE32INT contains 10 FUs.

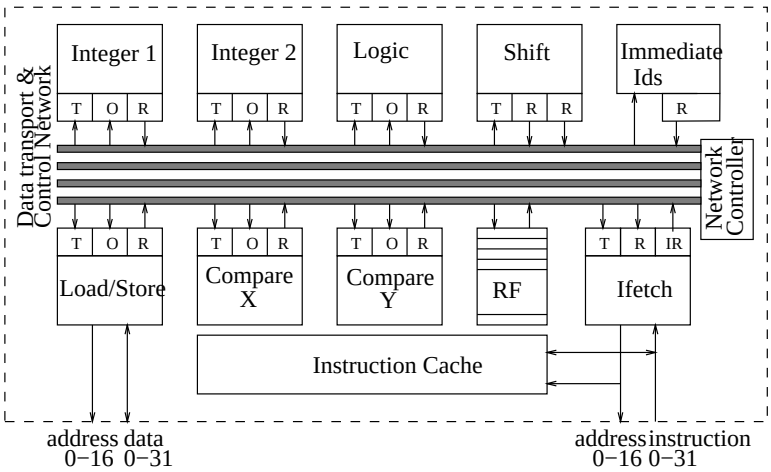

Figure 7. *Operational view of MOVE32INT.*

Integer units perform addition and subtractions and the logic unit boolean operations, AND, OR, and XOR on two operands. Shift unit performs 1-bit and 2-bit shifts on the data fetched to trigger register and stores each result to separate registers. There is one immediate unit for each bus. Reading from this unit has the effect of putting the 6-bit source identifier of the move operation as an unsigned integer on one of the data transport busses. Compare units (X,Y) produce boolean result values which can be read from the result registers. Their output is not connected to the data bus, but to the control bus, and is used to guard the data transports. Load & Store unit contains a trigger and result register for the load part, and a trigger and operand register for the store. This unit is connected to the external data memory. Instruction fetch (Ifetch) unit is basically a load unit, where the address (the program counter) is auto-incremented. However, writing to the trigger register forces a jump. Reading from the result register gives the address of the current instruction plus 2. The Ifetch unit is connected to a small internal instruction cache. FUs are implemented using the hybrid pipelining mechanism. The transport pipeline of MOVE32INT employs the three stage scheme presented in Section 2.2.2.

Each instruction for MOVE32INT contains 64 bits and specifies four transports. The format for the instruction and the move are shown in Fig. 8. It contains 3-bit guard specifier, 1-bit immediate flag, and 6-bit source and destination fields. The immediate flag determines the interpretation of the source field. This source field contains either a short 6-bit immediate, or a specification of a source register.

MOVE32INT supports a very general way of conditional execution; each move is conditionally executed. The condition is specified by a 3-bit guard specifier in each move.

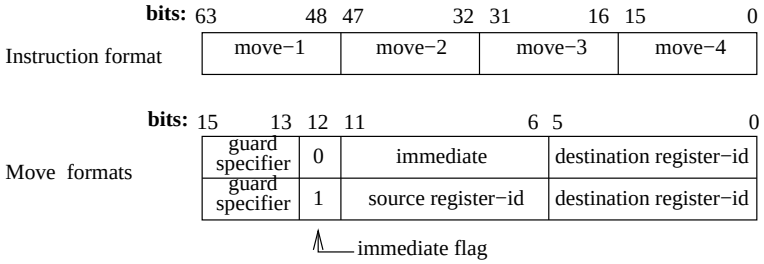

Figure 8. *MOVE32INT instruction and move formats*

During each cycle 4 guards are produced, one for each transport bus. The guard determines if a move has to be performed or has to be squashed. The guard specifiers indicate how to evaluate the guards; many boolean expressions of the boolean results of the compare units (Rx and Ry) can be specified.

MOVE32INT was realized and fabricated in a 2 μm (minimal gate length 1.6 μm, 2 metal layers) CMOS Sea of Gates (SoG)[11] technology. The SoG image contains 88 rows of 1088 transistor pairs per row resulting in 191k transistors. The total die size is 1×1 cm^2. MOVE32INT can achieve a relatively high clock rate, 80 MHz, despite the modest technology used.

2.4.2 Application-Specific Processor for Navigation Receiver

An application-specific processor for a multi-system navigation receiver is presented in [12]. Since various computation tasks, such as real-time digital filtering, Fourier transforms, and tracking loop algorithms, has to be performed in the receiver, an ASIP was chosen instead of custom hardware. The flexibility of transport triggered architectures allowed the designers of the processor to optimize the processor configuration for the real-time signal processing, specifically for a digital finite impulse response (FIR) filter. The programmability of this architecture makes it possible to adapt the receiver for new navigational algorithms.

The navigation processor contains multiplier, adder, accumulator, and two vector register units. The vectors register units contain 128 registers which are organized as FIFO queue with feedback from head to tail through a multiplexer. The queue advances one position when the vector register is read, and the head value is written back to the tail and passed to transport bus. When a write to the vector register occurs, the FIFO does not advance and the tail element gets replaced by the value from the bus. The vectors register is used in navigation processor to store FIR filter coefficients.

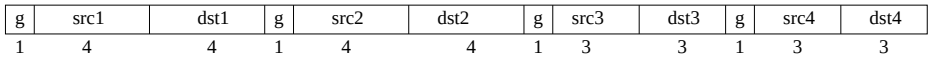

Figure 9. *Instruction format of the navigation processor.*

The transport network of the navigation processor consists of four 16-bit transport buses, which is sufficient for fast FIR filter computation. The buses are implemented with precharged wired-OR technique. Each instruction for the processor contains 32 bits and specifies four transports, one transport for each bus. The instruction is composed of source and destination fields as depicted in Fig. 9. A guard bit is specified for each transport. If the guard bit is true the move is always executed. If it is false, the move is disabled if the predicate most recently computed by the of the guard unit is false.

The navigation processor was implemented on 180k transistor 1.6 μm CMOS sea-of-gates chip. The transport network was found to be the critical path in the processor, limiting the maximum clock frequency to 125 MHz.

3. MOVE FRAMEWORK

As the discussion in the previous chapter shows, transport triggered architectures have properties that are valuable for an application specific processors. They are constructed from a limited number of building blocks and are thus modular. The performance of TTAs can be scaled to meet the application requirements by adding more functional units. The communication resources, i.e., buses and sockets, can be carefully tailored for a given application independently of the FUs. Moreover, to enhance performance, application-specific functionality can be easily added without any kind of modification to the interconnection network interface or the instruction decoding. Finally, a TTA processor can be designed to be fully programmable so that it can execute any high-level language program.

To exploit these advantages, an automated design framework, called the MOVE framework, has been developed in Delft University of Technology. The MOVE framework aims at automated design of TTA-based computing systems. As shown in Fig. 10, it consists of three interoperative parts: hardware subsystem, software subsystem, and design space explorer. The hardware subsystem is aimed at the automatic generation of MOVE processor hardware designs, implemented in a specific technology. The software subsystem is aimed at generating machine code for one or more target applications, written in a HLL. The third component, design space explorer, automates the search of suitable TTA configuration for a given application.

The MOVE framework is based on a shared architecture description called machine description file, a complete textual specification of the target processor architecture, which contains a set of architectural parameters. These parameters fully describe the essential characteristics of the target processor, such as the number and type of FUs, transport buses, RFs, etc. The machine description file is used as a communication method between the three subsystems of the MOVE framework.

The design space explorer is discussed in Section 3.2. The software subsystem, which is further consist of front-end and back-end is detailed in Section 3.3. Finally, the hardware subsystem is subject of Section 3.4.

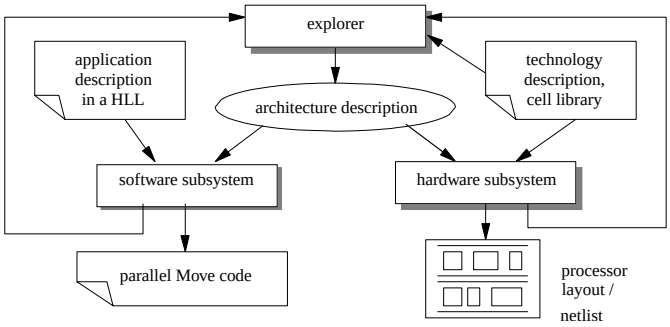

Figure 10. *General organization of the MOVE framework.*

3.1 Architecture Template

The architecture template of the MOVE framework (the MOVE architecture template) defines the set of architectural parameters that uniquely specify the properties of the target TTA processor. A processor is designed by instantiating the user-definable architectural parameters of the template, specified in the machine description file. The MOVE architecture template is a subset of the general TTA concept, although extremely flexible one. Use of such a template enables automated design space exploration, retargetable code generation, and automated processor hardware synthesis.

The interconnection network can contain any number of buses and the connectivity between sockets and buses can be freely chosen. However, the width of the move buses is currently limited to discrete values of 1, 32, and 64 bits. Consequently, this defines the minimum and maximum input and output data widths of the FUs, because if a functional unit is connected to 32-bit bus it is assumed that the FU supports operations on 32-bit input data.

The MOVE architecture template does not set limitations on the number of functional units or register files. Moreover, a functional unit may contain any type of operations on condition that they have equal latencies. The functional unit can either fully pipelined, which means that a new operation is allowed to trigger every cycle, or completely unpipelined. The latter means that there has to be at least *latency* cycles between two consecutive operations. For pipeline control, the MOVE architecture template supports hybrid-latching pipelining and semi-virtual time latching pipelining (Section 2.2.3). In functional units, the latency is user-definable architectural parameter, whereas the latency of register files is fixed to one clock cycle. This means that value stored in a general-purpose register, located in the register file, must be accessible by a result move next cycle it was written by a trigger move. The register files of the MOVE architecture

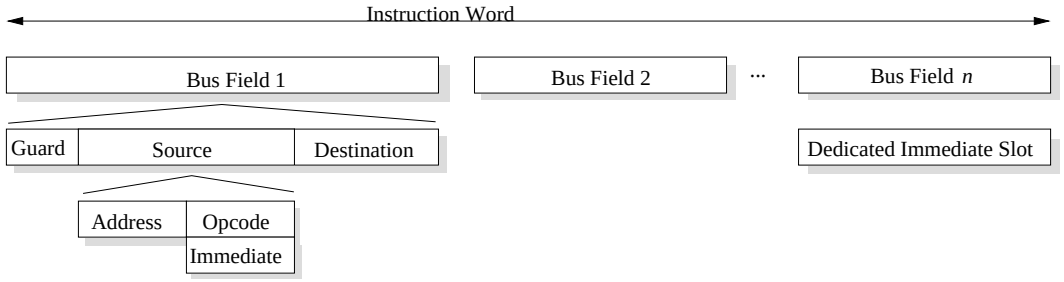

Figure 11. *Instruction format of MOVE architecture.*

can have any number of read ports and write ports. The number of input and output ports of functional units is restricted to eight.

The generic instruction word of the MOVE architecture is depicted in Fig. 11. Apart from the source and destination fields the instruction word contains a guard field for each bus. The guard field for a given bus contains a binary coded guard expression that specify a condition whether the move on the bus should be executed or not. The guard expression are expressions of boolean variables, stored in boolean registers. Contents of the guard field for a processor with two boolean registers are shown in Table 1. If the guard expression is false, the move on the corresponding bus is blocked. The boolean registers can be written through interconnection network, but their contents can only be used in guard expressions.

Constant values that are known at compile-time do not need to be stored in registers or memory; instead, their binary representation can be encoded into the instruction stream. Small constants can be encoded in the source fields. For longer constants, dedicated immediate slots can be used.

In the MOVE architecture, a jump operation is performed by writing the destination address into the program counter, and conditional branches are simply guarded jump operations. Depending on the instruction pipeline, a jump can have one or more delay slots. For example, on an instruction pipeline consisting of three stages IF-DC-MV (instruction fetch, instruction decode, move) a jump takes two cycles (after being fetched from memory) to update the program counter with its target address. If the latency of instruction memory is greater than one clock cycle, it adds delay slots to jumps. A subprogram call is special kind of jump in which the previous value of the program counter is written to return address register. The contents of return address register can be read through the interconnection network and saved if nested jumps are required. Only absolute addressing of the instruction memory is supported in the MOVE architecture.

Table 1. *Guard encodings for processor with two boolean register*

binary word	expression	interpretation
“000”	true	always execute the move
“001”	b0	execute if boolean register 0 contains ’1’
“010”	b1	execute if boolean register 1 contains ’1’
“100”	false	never execute the move
“101”	¬b0	execute if boolean register 0 contains ’0’
“110”	¬b1	execute if boolean register 1 contains ’0’

3.2 Design Space Explorer

Designing an ASIP by means of templated TTA consists of finding a proper configuration for the given application, where the configuration corresponds to a set of architectural parameters. For TTAs, the architectural parameters are the number and type of FUs, the configuration and size of RFs, and the capacity of the interconnection network. The design objectives for an embedded ASIP are to minimize execution time and cost, where the latter is proportional to the processor area. The objectives are conflicting, a fast processor is typically also an expensive one. It is very unlikely that examining a non-trivial application is sufficient to find a suitable processor configuration. For this reason the design process should based on quantitative feedback from the compiler and the hardware estimator.

Although the TTA template is used in the design process, the design space is still extremely large. Selecting a proper solution from this design space requires analysis of many design points. Design space explorer of the MOVE framework automates this search process [6].

The design space of TTAs is infinite, discrete, and has a large number of dimensions. An evaluation function of the Design space explorer maps the design space to a two dimensional cost/execution time space. A configuration is evaluated by invoking the compiler and hardware estimator. Information on cycle count is obtained from the compiler whereas the hardware estimator approximates the silicon area of the target processor configuration being evaluated.

Only a subspace of the cost/execution time space is interest to the designer. These points are called Pareto points. A configuration is a Pareto point if it is realizable and there are no other realizable configurations that are both faster and cheaper.

Design space exploration is composed of the two separate tasks, resource optimization

and connectivity optimization. Resource optimization consists of finding the right match of resources, such as which FUs to use, how many buses, how many registers, and how many ports on the register files. The goal of resource optimization is to reduce costs. Connectivity optimization determines the connectivity between the buses and the sockets. The resulting connectivity is based on the communication requirements between the FUs and the register files. The primary goal of connectivity optimization is to reduce the bus load and thus the cycle time. Details of resource optimization is presented in the Section 3.2.1. The Section 3.2.2 discusses connectivity optimization.

3.2.1 Resource Optimization

The objective of resource optimization is to find present a large set of Pareto points to the designer from which he/she can make a choice. The Pareto points are found by the means of a local search algorithm [6]. The starting point for resource exploration is a oversize architecture configuration. From this configuration the exploration proceeds to the next configuration by removing one of the resource. The resource to be removed is determined according to the following quality function that evaluates a design space point:

$$quality = \frac{1}{ExecutionTime^{\alpha} \times Costs^{\beta}} \quad (1)$$

where α and β are constants, which express the relative importance of performance and costs. The influence of the high bus load on the execution time is ignored at resource optimization phase since connectivity optimization, which takes care of the bus load, is performed after resource optimization has been completed.

After the local search algorithm has reached the minimum configuration the process is reversed and the resources are added until the initial configuration is reached. With this sweep back, it is possible to find new Pareto points. During the remove-sweep it is necessary to stay as close as possible to the cost axis. This is achieved by $\beta > \alpha$. Similarly, during the add-sweep it is necessary to stay close to the execution time axis which is achieved by $\alpha > \beta$. A few remove/add-sweeps are performed with different values for the α and β constants. For each configuration found in resource optimization, a machine description file is generated.

3.2.2 Connectivity Optimization

Connectivity optimization transforms the fully connected configuration found by resource optimization into a partially connected configuration that has less load on the

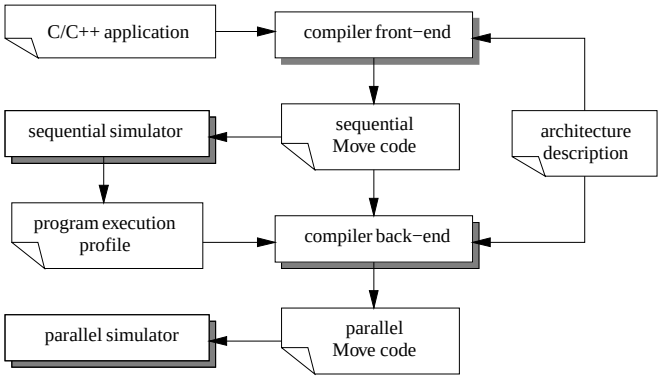

Figure 12. *Software subsystem of MOVE framework.*

buses and therefore a shorter cycle time. This is accomplished by removing bus socket connections from the buses in round robin fashion to balance the bus load. In addition, removing connections also results in smaller area due to simpler multiplexers(demultiplexers) in the sockets. Furthermore, the instruction size may decrease since the number of addressable locations per bus is lower. The bus socket connection that is removed from a bus is the first connection that has no influence on the cycle count. If no such connection exists, the connection with the lowest influence on the cycle count is taken. This process is repeated until the cycle time remains constant and the cycle count starts to increase. [6] As with resource optimization, a configuration found in connectivity optimization phase corresponds to a unique machine description file.

3.3 Software Subsystem

The purpose of the software subsystem is to generate machine code that can be run on a given target processor for an application specified by the user. Also, the software subsystem produces execution statistics and allows to verify that the generated code is correct. The various tools that form the MOVE software generation subsystem and their relations are illustrated in Fig. 12. First, a front-end compiler based on GCC, accepts applications written in a HLL, e.g., C or C++, and generates unscheduled, sequential MOVE code, which can be simulated with the sequential simulator to obtain profiling information. The back-end reads in the sequential code, the architecture description and, if present, the execution profile. The main task of the back-end is to generate instruction-level parallel code, optimized for the target processor specified by the architectural parameters. A flexible, retargetable instruction scheduler, capable of region scheduling and software pipelining performs the task. The resulting code can be simu-

lated with the parallel simulator. The output of the parallel simulator can be examined and compared to the output of the sequential simulator in order to verify the correctness of the generated code. Performance statistics such as schedule length, hardware resource utilization and execution time give the ASIP designer valuable feedback and are also used by the design space explorer, described in previous section. [5]

3.4 Hardware Subsystem

The hardware subsystem of the MOVE framework is responsible for generating and evaluating the hardware of the processor design. The two components of the hardware subsystem are MOVE estimator, detailed in Section 3.4.1, and the MOVE processor generator which is the subject of Section 3.4.2.

3.4.1 MOVE Estimator

The MOVE estimator is a tool responsible for characterizing a given target processor in terms of attainable clock frequency and silicon area. As was presented in Section 3.2.1, a large number of different architecture configurations has to be characterized during design space exploration process. Complete synthesis process and netlist analysis for each design point for obtaining area and timing information on a target processor would make the exploration runtimes extremely long. Therefore, the MOVE estimator uses a model of higher abstraction level to characterize a move configuration corresponding a design point. As with other parts of the MOVE framework toolset, the machine description file is the primary input to the estimator tool.

The area estimation of a configuration is fairly straightforward. It is the sum of the area of the needed for FUs, sockets, register files, transport buses, and bonding pads. Optionally, the area of a on-chip program RAM can also be added to this number. There are separate expressions for each type of FU. The area of the FUs is function of operand width. Also the number of pipeline stages is taken into account. The area of a socket depends on the number and the width of the buses to which the socket is connected. Using the area estimates for the sockets and functional units the area required for the transport buses is calculated. Assuming that channel routing is used, the area of buses increases almost quadratically with number of wires (bits) 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 units whereas the number of power/ground

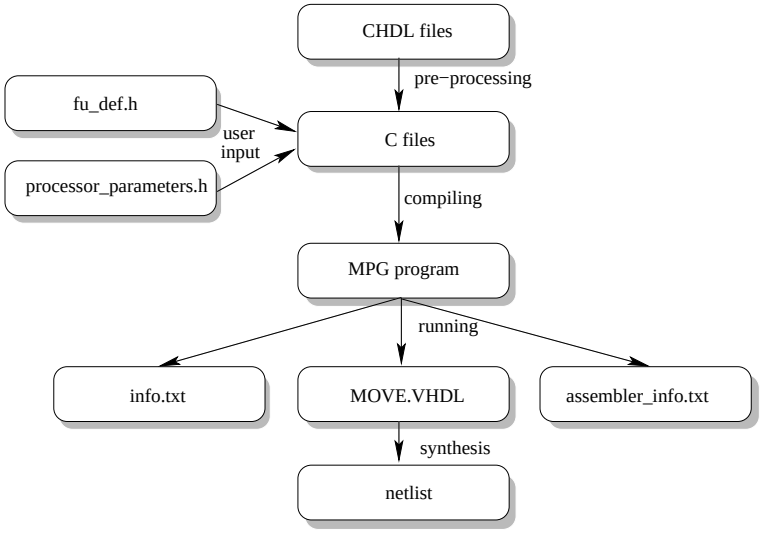

Figure 13. *MOVE processor generator design flow*

pads is related to the total area. [13]

The minimum cycle time is the maximum of the longest FU stage delay and the longest bus transport delay of the entire processor if three stage transport pipelining is employed. In case of two stage pipelining, the final stage of the FU is directly connected to the bus. This means that path dictating the cycle time is the slowest final-FU-stage/transport bus combination. [13]

3.4.2 MOVE Processor Generator

The MOVE processor generator (MPG) generates VHDL description of a MOVE processor according to parameters given by the user. The produced VHDL code can be simulated at register transfer level (RTL) with aid of a generated testbench. Any standard computer aided design tools that accept VHDL as entry language, e.g., logic synthesis software, can be used to process the design further. Complete design flow using the MPG is illustrated in Fig. 13. [14]

Before being able to generate the VHDL description, the MPG needs information on the target processor configuration. This information is divided between two textual files, “processor_parameters.h” and “fu_def.h”. In the latter one, the functional units that are available to be used in the processor configuration are defined in C language arrays, illustrated in Fig. 14. Global properties of the processor, such as number and width of the transport buses are defined in “processor_parameters.h”. It also contains attributes of some function units like the instruction fetch unit and the load-store unit.

```
Fu_def fudefs[] =
{
  {
    "int",              /* FU type */
    "integer_unit",     /* FU id */
    intfua,             /* FUC generation function */
    {
      1, 1, 1,          /* no. of T, O, and R ports*/
      {1}, {0}, {0},    /* no. of id bits used for above */
      {32}, {32}, {32}, /* datawidth of the sockets */
    },
          false,        /* FU uses clock signal */
    false,              /* FU uses reset signal */
    false,              /* FU uses lock request signal */
    true,               /* FU uses result register */
    true,               /* FU uses SVTL of TVTL    */

    0,                  /* number of off-chip signals */
    {0},                /* width of each off-chip signal */
    {0},                /* type of each off-chip signal */
    {""},               /* name of each off_chip signal */
  }

                              ...
```

Figure 14. *Functional unit definition in fu_def.h*

For example, support for virtual memory can be enabled and the size of the generated caches can be specified.

The user can define the combinatorial logic for his own units but the pipeline control mechanism and FU interface, detailed in Section 2.2.3, are always automatically generated. As well as the FU definitions, also the FU instantiations and their connections to the buses are specified in “fu_def.h”. Again the information is organized in C language arrays. Functional unit parameters and the connectivity between FUs and transport network, defined in the arrays, has to match the parameters and the connectivity defined in machine description file in which the architecture configuration is defined for the rest of the tools of the MOVE framework. For example, an excerpt entry of a machine description file, shown in Fig. 15, defines a functional unit fu0 to be instantiated and connected to buses one and two, of three transport buses contained in the processor. The corresponding section of “fu_def.h” shown in Fig. 16.

The user has to take care that the instantiated, functional unit can perform the necessary operations which in example of Fig. 15 are integer addition and subtraction. A drawback is the fact that the connectivity information has to be copied manually from the section where the sockets are defined in machine description file. No automatic tool is provided with the MPG for this conversion. The interface via C language arrays also degrades the

```
                        ...
 Sockets
 {
         as0_o         input,  { bus1,       bus3 };
         as0_t         input,  {       bus2       };
         as0_r         output, { bus1, bus2, bus3 };


                        ...
FunctionUnits
{
        as0          always, 2, {as0_o}, as0_t,
                                {as0_r}, {add, sub};
}

                        ...
```

Figure 15. *Functional unit instantiation in machine description file*

usability of MPG. The user has to define the width of the field, in which, for example, the bus to which a socket in connected to. The MPG does not perform any checks if the number of elements in the field matches the defined width. The user has no way to validate that the specification in the in“fu_def.h” actually matches the definitions in the machine description file.

The instruction fetch unit template applied by the MOVE processor generator is presented in [15]. It contains support for features such as exceptions, virtual memory, and instruction cache. Even though these functions could be useful in some implementations, organization of the instruction unit template is not flexible enough so that these additional features could be removed or added individually.

MPG and the binary code generator of the MOVE framework back-end do not use the same algorithm for managing the addresses (IDs) for the sockets. For this reason, the processor generated by MPG cannot run the binary code generated with the software subsystem of the MOVE framework as such. The hardwired identifiers in the generated VHDL description have to be changed manually, which is tedious and error-prone process.

MPG is written in CHDL, which is a mixture of C and VHDL languages. The CHDL pre-processor creates C files of all CHDL files. The C files generated by the CHDL pre-processing are be compiled and the executable file MPG is generated.

The MPG generates three output files, info.txt, assembler info.txt and move.vhdl. info.txt contains information about the generated processor architecture, for example, an overview of the chosen parameter values and some values calculated by the MPG. Information required for code generation is available on “assembler_info.txt”. All the generated

```
{
  "as0",           /* name of FU instantiation */
  "integer_unit",  /* id */
  {                /* tsocks configuration */
    {
      1,           /* number of buses */
      {1}          /* buses tsock is connected to */
    }
  },
  {                /* osocks configuration */
    {
      2,           /* number of buses */
      {0,2}        /* buses osock is connected to */
    }
  },
  {                /* rsocks configuration */
    {
      3,           /* number of buses */
      {0,1,2}      /* buses rsock is connected to */
    }
  }
}

                          ...
```

Figure 16. *Functional unit instantiation in fu_def.h*

VHDL code for this particular processor design is included in “move.vhdl”.

4. NEW PROCESSOR GENERATOR

When the MOVE framework is used for ASIP design, the designer first optimizes the target processor configuration to suit the performance requirements of the target application while simultaneously trying to minimize the costs. After an adequate configuration is found, it has to be transformed into a format that represents physical implementation. A processor generator can be applied to automate this transform process. Requirements for such a processor generator are discussed in Section 4.1. Implementation and design of a new processor generator is described in Section 4.2.

4.1 Requirements

The defects found in the MOVE processor generator, described in Section 3.4.2, encouraged to design a new processor, which has simpler user interfaces and better support for the inherent modularity of transport triggered architecture, and which can be easily incorporated to the rest of the tools of the MOVE framework. General requirements for a processor generator for ASIPs are discussed in Section 4.1.1. In Section 4.1.2 the interfacing between the MOVE Framework and the processor generator is defined. Section 4.1.4 describes interconnection structures that need to be supported by the processor generator.

4.1.1 General Requirements

Ideally, processor generator would operate as a silicon compiler that creates the very large scale integration (VLSI) layout image of the processor, representing the actual devices processed to silicon wafer. However, such a direct transform is not very feasible. There exists already numerous design automation tools for layout design and thus there is no need to include such capabilities to the processor generator. Instead, generating an intermediate design specification with HDL is more beneficial alternative.

Hardware description languages, such as VHDL or Verilog, allow structural and behavioral design specification at different abstraction levels. Although neither VHDL nor Verilog support description on technology specific physical level, both of them can be used to present a netlist, which describes the interconnectivity of blocks or cells. An equivalent layout image can exist for these blocks, or only a behavioral description for such a block can be given. By describing the behavior of the processor and its building blocks, the specification is less prone to errors and the verification process is alleviated. The simulation run-times of a behavioral description are shorter by many orders when compared to the run-times of circuit simulation on transistor-level netlist, extracted from layout. Different abstraction levels as well as structural and behavioral description style can be easily mixed, even inside a block. Another benefit for using HDLs as an intermediate format is that the HDL description can be technology independent but it can still contain the exact definitions of registers, buses, and off-chip ports that the physical implementation requires. Both VHDL and Verilog, being standardized languages [16][17], are accepted as design entry format by the majority of design automation tools. A special class of tools, logic synthesis software, can automatically transform a behavioral description of the processor to a netlist consisting of technology specific cells. [11]

The simplicity and regularity of TTAs enables automatic generation of complete, bit accurate, hierarchical HDL description of a TTA processor according to configuration description given by the user. The generated HDL is targeted to be used as an design entry for logic synthesis with standard cell technologies. During the synthesis, the design is optimized according to timing and silicon area constraints specified by the designer. However, the quality of results of the optimizations strongly depends on the coding style applied in the HDL description. Consequently, HDL code produced by the processor generator has to follow the guidelines and rules set for efficient HDL code so that there is no need to manually tune the processor design for better synthesis results.

4.1.2 Interfaces

The new processor generator has to obtain the information on the organization of the target processor directly from the machine description file. Machine description file has a well defined lexical structure and compact, accessible organization. A complete syntax and semantic check is performed when this file is accessed by the tools of the software subsystem. Therefore, majority of inconsistencies and syntax errors are eliminated if the file is processed in advance by the tools of the software subsystem, making the error checking a secondary design criterion for the new processor generator.

```
                                ...
MoveSlot  0:150,13
{
        GuardEncoding  11,2
        {
                inv: 1, 0x1;
                true:   0x1;
                b.0:    0x0;
        }

        SrcIDEncoding  0,7
        {
                Immediate:  6,1,0x0;
                Sockets:
                        ri4_o2  : 1,6,0x40      ri4_o2-0;
                        ri6_o2  : 1,6,0x42      ri6_o2-0;
                        fu15_r  : 0,7,0x44      fu15-4;
                        ri1_o2  : 0,7,0x45      ri1_o2-0;
                        ir_1    : 0,7,0x46      ir_1-0;
        }

        DstIDEncoding  7,4
        {
                Sockets:
                        fu15_t  : 2,2,0x0       fu15-0;
                        fu6_t   : 1,3,0x4       fu6-0;
                        fu27_t  : 1,3,0x6       fu27-0;
                        fu4_o   : 0,4,0x8       fu4-1;
                        fu4_t   : 0,4,0x9       fu4-0;
                        fu6_o   : 0,4,0xa       fu6-1;
                        b1_i    : 0,4,0xb       b1_i-1;
        }

}

                                ...
```

Figure 17. *Section of binary mapping file describing socket ID encoding*

The address generation method used in the back-end of the software subsystem does not have to be duplicated in the processor generator. For the binary code generation, information on the socket addresses and opcodes delivered to FUs is read from a specific binary mapping file. This file is a structured textual database that defines an unique ID for each addressable location (socket) and an unique opcode for each supported operation. The binary mapping file can be generated automatically for a given target processor configuration described in machine description file. The task of the new processor generator is to parse the information from this text file and generate the hardware description accordingly. A section of the mapping file where the encodings for destination and source IDs for sockets connected to a given bus is depicted in Fig. 17.

4.1.3 Modularity

In the new processor generator, there has be a clear partitioning which of the modules of the processor can be provided by the user and for which parts the HDL description is created by the processor generator. For transport triggered architectures this distribution of tasks if fundamentally straightforward: building blocks, which are to be modified when the number of computation resources and communication resources is varied, have to be generated. Primarily, this means the interconnection network and the top level binding of the components. Parts that have a regular interface to interconnection network, i.e., functional units and register files, can be provided as library components, which can be plugged in to the interconnection network at the top level description, independently of other components. This does not of course exclude that HDL generation for complete processor design can be supported. But an application programming interface (API) for utilizing user defined submodules should be given when it is possible. Design of a library of functional units is detailed in Section 4.3.

A simplistic instruction fetch unit is sufficient for the new processor generator. The fundamental task of the this instruction fetch unit is to retrieve instructions from the memory address pointed by the program counter and implement the control flow instructions, such as jumps and calls, supported by the software subsystem.

4.1.4 Support for Different Interconnection Structures

In Chapter 2, it was stated that the interconnection network of a TTA processor does not require full crossbar connectivity. Nevertheless, even a partially connected network may be responsible for a significant portion of the costs of a TTA processor. Interconnection network with a given architectural properties (buses, connections) can have numerous lower-level implementations, which appear identical from the compiler viewpoint. These implementations can differ significantly in terms of delay, silicon area, and power consumption depending on the used target technology. One interconnection structure may be more suitable for a given application and processor configuration. Therefore, the new processor has to able to generate HDL code for different interconnection structures.

To be precise, the design freedom on different interconnection network structures on the behavioral level consist mostly on specifying the design hierarchy and the detailed structure of demultiplexer in the datapath of the result sockets. In fact, as will be shown later, the result sockets are not necessary a concrete hardware component, but merely a

concept for the compiler and the designer to define connections between functional unit outputs and buses. The transport triggering concept is not dependent on the existence of the sockets. It mainly defines that a processor is programmed by specifying moves that are conducted by an interconnection network.

4.2 Implementation

Based on the guidelines set in the previous section, a new processor generator was designed and implemented. Fundamentally, a processor generator is a software tool that performs a transformation from one source hardware description language to another. In this case, the transformation means converting the description presented in internal format of the MOVE framework to a generic standardized hardware description language code that represents the processor design in format that is not dependent of the MOVE framework.

For statically scheduled processors such as TTAs, all the latencies, i.e., the depths of internal pipelines in the processor, have to be visible to the compiler. If the latency is an architectural parameter that can be varied, then it is specified in machine description file. Latencies that are not specified cannot be changed; otherwise the compiler cannot schedule the code correctly. Therefore, there is virtually no opportunities to optimize the processor at the HDL generation phase of the design process. For this reason, the HDL generation is fairly straightforward and consists mostly on string processing and formatting. Also the complexity of feasible processor designs is limited and, therefore, it is not necessary to pay too much attention on performance and the memory traces of the internal data structures of the processor generator, specially when the processor generator is run on a modern computer workstation.

Due to the low performance requirements, the processor generator was written in Python [18]. Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines powerful text manipulation features with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. Finally, Python is portable: it runs on many brands of UNIX, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2.

The processor generator is a command line script-like tool that obtains the name of the machine description file and corresponding binary mapping file as well as the bus struc-

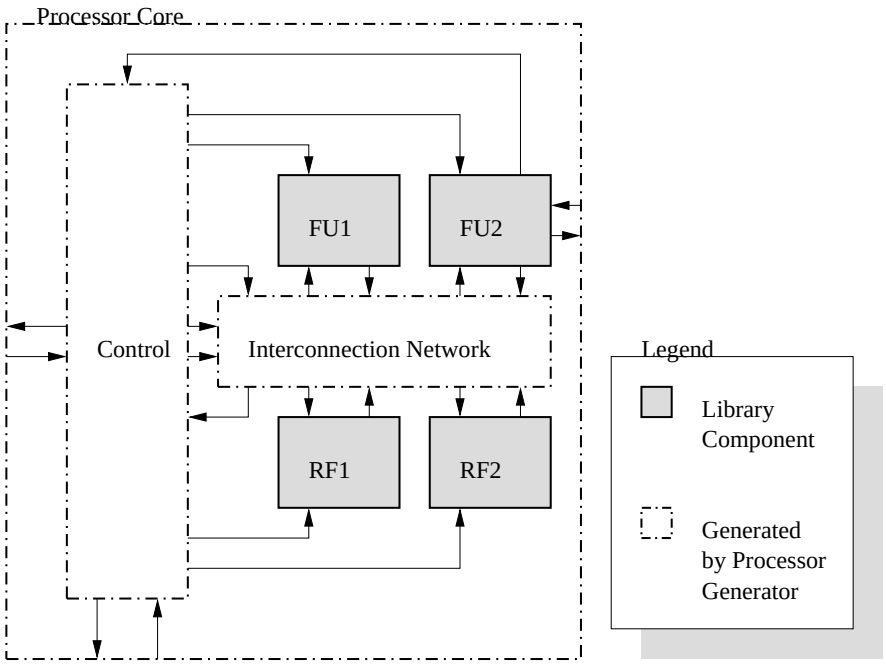

Figure 18. *Processor Organization*

ture to be used as command line parameters when it is invoked. Information from the files are parsed and stored to internal data structures. Additional information, which is not available on machine description file, is provided with separate input files. An input file to define external ports of the processor can be given as well as a file that specifies how the external ports are mapped to ports of the functional units. The processor generator produces a separate file of VHDL code for each generated submodule of the processor. The organization of the generated processor explained in next section.

4.2.1 Processor Organization

The control of the TTA processors has traditionally been distributed to the sockets that have been responsible of performing both the instruction decoding and the data flow control. A different point of view to the organization of the processor has been taken for this processor generator. This structure is illustrated in Fig. 18. All the hardware allocated for control path of the processor is placed to a control unit. The interconnection network, being exclusively a datapath component, consists of multiplexers from input sockets and demultiplexers from output sockets. The instruction decoding from the sockets is centralized to the control unit. In the instruction decoding, the pipeline control signals opcodes are generated for functional units and register files as well as control signals for multiplexers and demultiplexers of the interconnection network.

The task of the processor generator at top-level of design hierarchy is to create the top-level structural VHDL description, i.e., a netlist, of the target processor. The netlist is composed of set of instantiated components and connections between them. Two of the components, the control unit and interconnection network are generated by the processor generator. Functional units and register files are instantiated from a pre-designed library of components. The detailed structure and HDL code generation for interconnection and control unit are discussed in Sections 4.2.2 and 4.2.3, respectively.

To be able to exploit user defined functional units and register files, a clear and consistent interface specification for a FU/RF is required. Assuming this interface specification is met, the processor generator can automatically instantiate components from a library of units and generate the necessary wiring to connect the unit to interconnection network and control unit.

There has to be a convention how a functional unit defined in the machine description file corresponds to a HDL object such as entity (VHDL) or a module (Verilog). For this processor generator this binding is realized through the module/entity name. The architectural parameters for a unit are combined into a character string which is the name of entity/module expected to be found in the library.

The general interface for a functional unit/register file is illustrated in Fig. 19. The interface ports are divided into classes as follows. Ports of the standard interface carry data from unit to interconnection network and vice versa. The opcodes indicate (a) the operation to be executed and (b) the register that is written. For each FU, there has to be one (and only one) trigger data input (`t1data`), but for register files with I input ports, I trigger data inputs (`t1data..t`I`data`), and thus I opcode inputs (`t1opcode..t`I`opcode`) are required. If the FU supports only one operation or a register file has only one general purpose register, the interface does not contain the opcode port. The bit width of trigger data is either same as bus width or a subrange of that (specified by the user). The bit width of opcode is $\lceil \log_2 K \rceil$, where K is either (a) the number of operations supported by FU or (b) the number of GPRs in the register file. Only FUs can have operand data inputs (`o1data..o`N`data`). The bit width of operand data is either the same as bus width or a subrange of that (specified by the user). Each read port in a RF and each output port in a FU corresponds to a result data port (`r1data..r`M`data`).

Only register files need result opcodes, which indicate the GPR that drives data to an output port. A register file with M write ports has thus exactly M result opcode ports (`r1opcode..r`M`opcode`) in its interface. Again, the bit width of result opcodes is

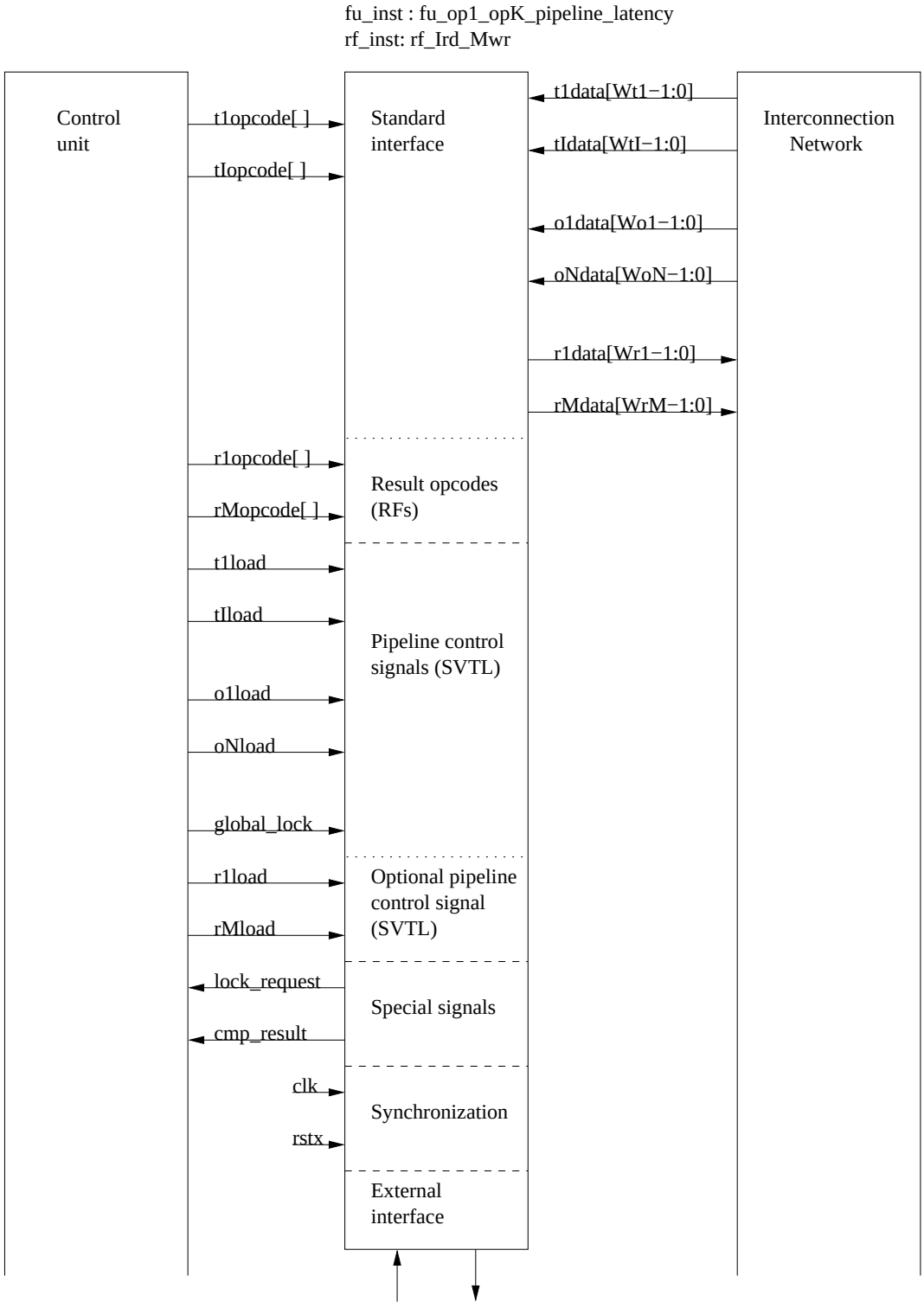

Figure 19. *General interface of functional unit / register file.*

$\lceil \log_2 K \rceil$, where K is the number GPRs in the register file. If a register file has only one GPR, the interface does not contain the result opcode port.

Pipeline control ports are required for signals which control the FU pipelining. The

```
as0 : fu_add_sub_always_2
port map (
  t1data   => as0_t_data,
  t1load   => as0_t_load,
  t1opcode => as0_t_opcode,
  o1data   => as0_o_data,
  o1load   => as0_o_load,
  r1data   => as0_r_data,
  global_lock =>  global_lock,
  clk      => clk,
  rstx     => rstx);
```

Figure 20. *FU instantiation in top-level VHDL description.*

interface shown in Fig. 19 is for semi virtual-time-latching pipelining (Section 2.2.3). For each input (trigger or operand) a `load` signal is required to indicate that new data is to be loaded into the unit. If different pipelining scheme is used the interface is subject to change.

Special signals class contains ports for special signals that have effect on processor control. For example, a comparator result, `cmp_result`, can be bypassed to control unit. Also, a unit that can halt the processor execution by requesting a lock. This is indicated by a `lock_request` signal. An input port for active low reset (`rstx`) and clock (`clk`) must be provided for each unit.

A port of a functional unit, which is not member of any class described above, has be connected to an external port of the generated processor core. The external ports of the processor core, and the connections from functional unit ports to the external ports have to be explicitly specified by the user in input files called “external_ports” and “fu_port_map”, respectively.

In order to be correctly used with the processor generator, input and output ports of a functional unit/register file in the library must be designed according to this specification. For example a generated component instantiation corresponding to the machine description file entry from Fig. 15 is depicted in Fig. 20. A module/entity with name fu_add_sub_always_2 has to be available in the library when the generated code is compiled. The ports on the left side of operator “`=>`” have to exist in the library component fu_add_sub_always_2. The signals on the right side of “`=>`” are the wires, generated by the processor generator, to which the ports are mapped. These wires are furthermore connected to ports of control unit and interconnection network. A similar component instantiation is performed for all the components of the processor core.

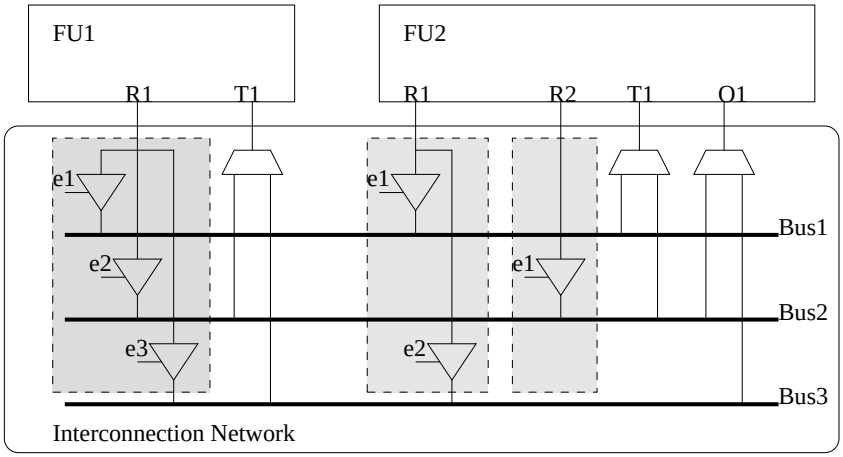

Figure 21. *Datapath realized with tristate drivers*

4.2.2 Interconnection Network

Traditionally, bus demultiplexing in MOVE processors is realized by means of tristate drivers. Support for two other interconnection network structures, which are more suitable for modern technologies, were selected for further evaluations.

Fig. 21 shows the detailed datapath of an interconnection network composed of three transport buses, three result sockets and three input sockets. Result port (socket) R1 of FU1 is connected to all three buses whereas other ports (sockets) have more limited connections. Shaded regions in the interconnection network represent the demultiplexers contained in the result sockets. For each bus connection a tristate driver of bus bitwidth is required in the demultiplexer. Each tristate driver is controlled by enable signal, which is generated at control logic. When the enable signal for a particular tristate driver is high, the data from the functional unit is driven to the bus. If the enable signal is low, the output of the tristate driver is in high impedance state or floating, i.e., there is no low impedance path from buffer output to power lines. As a result, outputs of several tristate buffers can be connected together, as long as only one driver is enabled at a time. Tristate buses are commonly used in full custom designs, but in standard cell based ASICs the use of tristate buses is not recommended. [19] Tristate buses require specific production test structures in order to get stuck'at 1 and stuck'at 0 faults detected. In addition, a standard cell library usually contains only a limited selection of buffer sizes for tristate drivers. Too strong buffers has to be selected to drive the a bus which typically has large capacitance due to fanout and wire length. This may result in implementation that has exceedingly large area and power consumption. Tristate buses reduce routing congestions as they can be driven from multiple physical locations. The

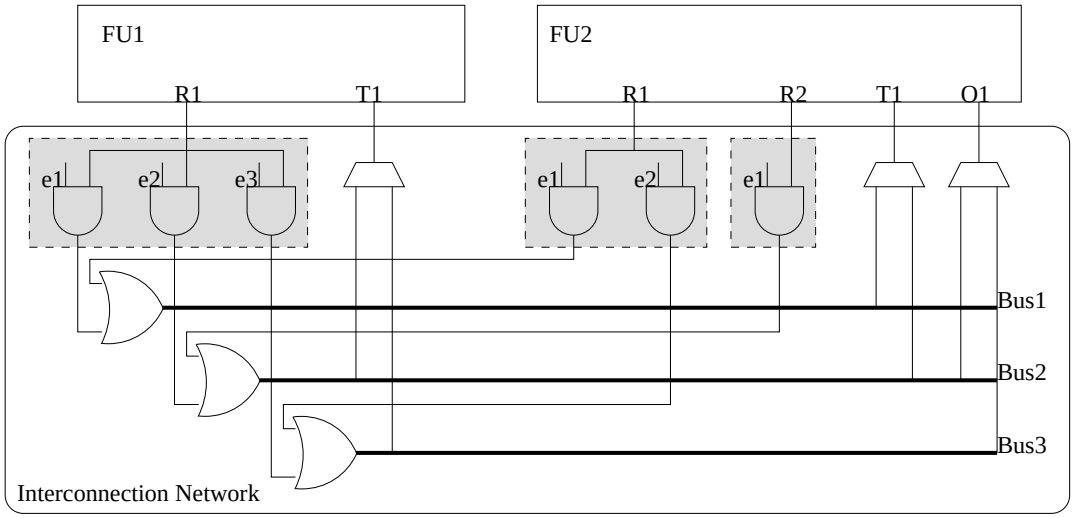

Figure 22. *Datapath realized with AND-OR network*

routability, however, is not typically a severe problem in modern process technologies which have more than five metal layers available for wiring.

Demultiplexing can also be realized with AND-OR network [20]. Fig. 22 depicts a datapath that is architecturally identical with one illustrated in Fig. 21. For each bus connection an AND gate of bus bitwidth is required in the demultiplexers which are presented as shaded regions in Fig. 22. AND gates, corresponding to FU outputs connected to a given bus, are connected to the OR gate driving the bus. Each AND gate is controlled by enable signal as was the case with the tristate drivers. If the enable is low for a particular AND gate the output of the AND gate is pulled low. When the enable signal for a particular AND gate is high, the logic value from FU output is propagated to the output of the AND gate. Provided that only one AND gate per bus is enabled, the logic value is further transmitted to the bus.

The datapath of the transport network of a TTA processor can be viewed as an interconnection of multiplexers. The input sockets contain multiplexers to select from which bus the functional unit trigger input or operand input reads data. For each bus there is multiplexer that selects the FU output writing the data on the bus. The control signal for a multiplexer is decoded from the corresponding source field of the instruction word. Fig. 23 illustrates a datapath that is architecturally identical with ones illustrated in Fig. 21 and 22. As can be observed, the result sockets are no longer components of the datapath.

Interconnection network is the communication medium for all the programmed data traffic in the processor core. The network has an input port for every addressable source

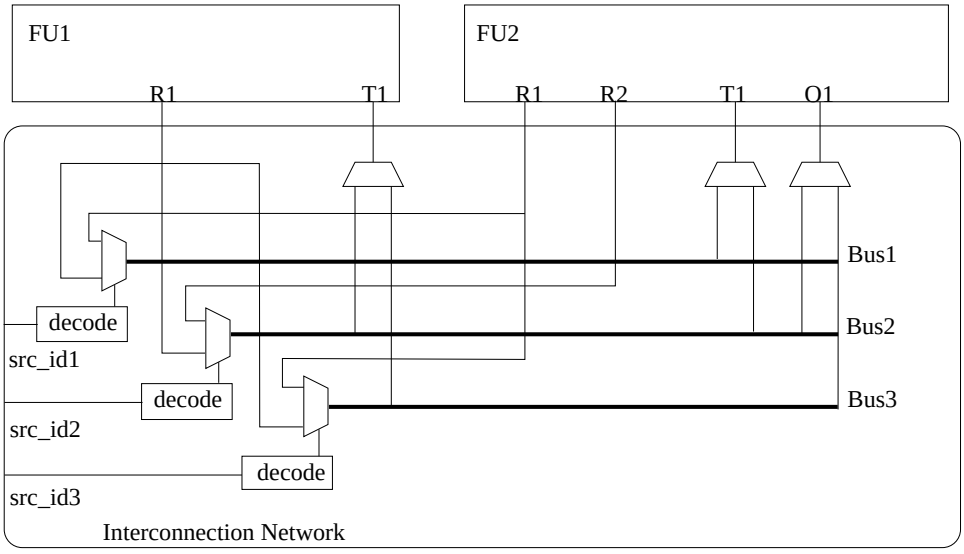

Figure 23. *Datapath realized with interconnection of multiplexers*

location and an output port for every addressable destination location. These locations are mainly trigger, operand, and result ports of functional units and register files but also the control unit can access the datapath, for instance to read the jump address. In addition to input and output ports for data, the interconnection network has inputs for control signals of multiplexers and demultiplexers. For an input socket, connected to n buses, a multiplexer control signal of $\lceil \log_2 n \rceil$ bits is required. A demultiplexer, connected to m buses requires m-bit control signal, because each AND gate and tristate buffers has to independently controllable. For an interconnection network realized with multiplexers the source field from the instruction word has to be provided to control the logic of bus write selector.

A fragment of generated VHDL code for the interconnection network, corresponding to an input socket is presented in Fig. 24. The input socket is connected to three buses (1st, 3rd and 4th) and requires thus a 2-bit control signal. Each input socket in the target processors is mapped into a similar process where the behavior of a multiplexer can be described in VHDL.

Fig. 25 illustrates VHDL code corresponding to a result socket. The result socket `as0_r` is connected to three buses (1st, 2nd and 4th) and requires a control bit for each bus. Demultiplexer in this particular example is realized by a AND-OR network. For each bus the result socket is connected, each bit of the result data from the functional unit is ANDed with control bit corresponding to the bus connection and the output of AND is assigned to intermediate data signal/wire (`transport_bus`*n*`_conn`*m*). The intermediate data wires for a given bus are ORed and the output of the multiple input

```
as0_t : process (as0_t_cntrl , transport_bus1,
                 transport_bus3, transport_bus4)
  begin  -- process as0_t
    case as0_t_cntrl is
      when "00" =>
        as0_t_data <= transport_bus1;
      when "01" =>
        as0_t_data <= transport_bus3;
      when others =>
        as0_t_data <= transport_bus4;
    end case;
  end process as0_t;
```

Figure 24. *VHDL code for input socket.*

OR is propagated on the transport bus. On the example depicted in Fig.25, the transport bus 1 has five result sockets, including `as0_r`, connected to it.

4.2.3 Control Unit

The control unit of TTA processor generated by the processor generator is composed of three functional units and instruction decoding logic, as shown in schematic presented in Fig. 26. An instruction fetch unit to load instruction words from memory must be present in every TTA processor. It contains a program counter (PC) register that is incremented at each clock cycle. This auto-incrementing is overridden when a new value, from `if_t1data` (See Fig. 26) is loaded to on jump or subprogram call. On jump, only the program counter register is affected, whereas on call the contents of the PC is loaded into a return address (RA) register. The PC write is indicated by `if_t1load` and the jump and call are distinguished by `t1_opcode` signal. The contents of the return address register can be read though `if_r1data` if the RA value needs to be saved on nested subprogram calls. A new value is written to RA through `if_t2data`. Hence, the instruction fetch unit contains three addressable locations, which can be accessed by a move. The instruction fetch unit must communicate with external memory or cache where the instructions are loaded. This complicates automated generation of the unit, because different memory interfaces has to be supported. At the moment, the instruction fetch unit contains only simple interface to a read-only memory (ROM) and no cache. The contents of the program counter is used as an address when the memory is accessed. The retrieved instruction word is propagated to other modules of the control unit. For all practical purposes two additional functional units, immediate unit and guard unit, are required in the control path to enhance the programmability of the processor.

```
-- Demultiplexer for result socket as0_r
transport_bus1_conn1 <= ext(
  as0_r_data and
  sxt(as0_r_cntrl(0 downto 0), as0_r_data'length),
  transport_bus1_conn1'length);
transport_bus2_conn1 <= ext(
  as0_r_data and
  sxt(as0_r_cntrl(1 downto 1),as0_r_data'length),
  transport_bus2_conn1'length);
transport_bus4_conn2 <= ext(
  as0_r_data and
  sxt(as0_r_cntrl(2 downto 2),as0_r_data'length),
  transport_bus4_conn2'length);

                      ...
    -- Write to transport_bus 1
    transport_bus1 <= transport_bus1_conn1
                or transport_bus1_conn2
                or transport_bus1_conn3
                or transport_bus1_conn4
                or transport_bus1_conn5
```

Figure 25. *VHDL code for result socket (AND-OR).*

Some specific hardware must be allocated to evaluate the guard expressions, decoded as binary words in the guard fields of the instruction word. The guard fields of the instruction are delivered to the guard unit, in which the hardware resources required to support conditional execution are located. If the guard expression is false, the move on the corresponding bus is blocked, which is indicated to instruction decoding unit by a squash signal. Since the boolean registers have to be directly accessible when the guard expressions are evaluated, they are placed in the guard unit. The guard unit depicted in Fig. 26 has an interface for one input socket for the boolean registers. A boolean value from `grd_t1data`, typically a result of a comparison, is loaded into the boolean register pointed by `grd_opcode` when `grd_t1load` is active.

The source field of the instruction word is composed of result socket address and opcode part. The opcode part can be used as an immediate value (see Section 3.1). When three stage instruction pipelining is employed, a register is required for each immediate value so that the immediate can be specified in the same instruction, which it is transported. These registers are in the immediate unit. The structure of the immediate unit, as well as guard unit, is dependent on the configuration of the target processor and thus the processor generator is responsible for generating the hardware description for these units.

In the instruction decoding unit, the contents of the instruction word is decoded into pipeline control signals and opcodes for functional units and register files. Additionally,

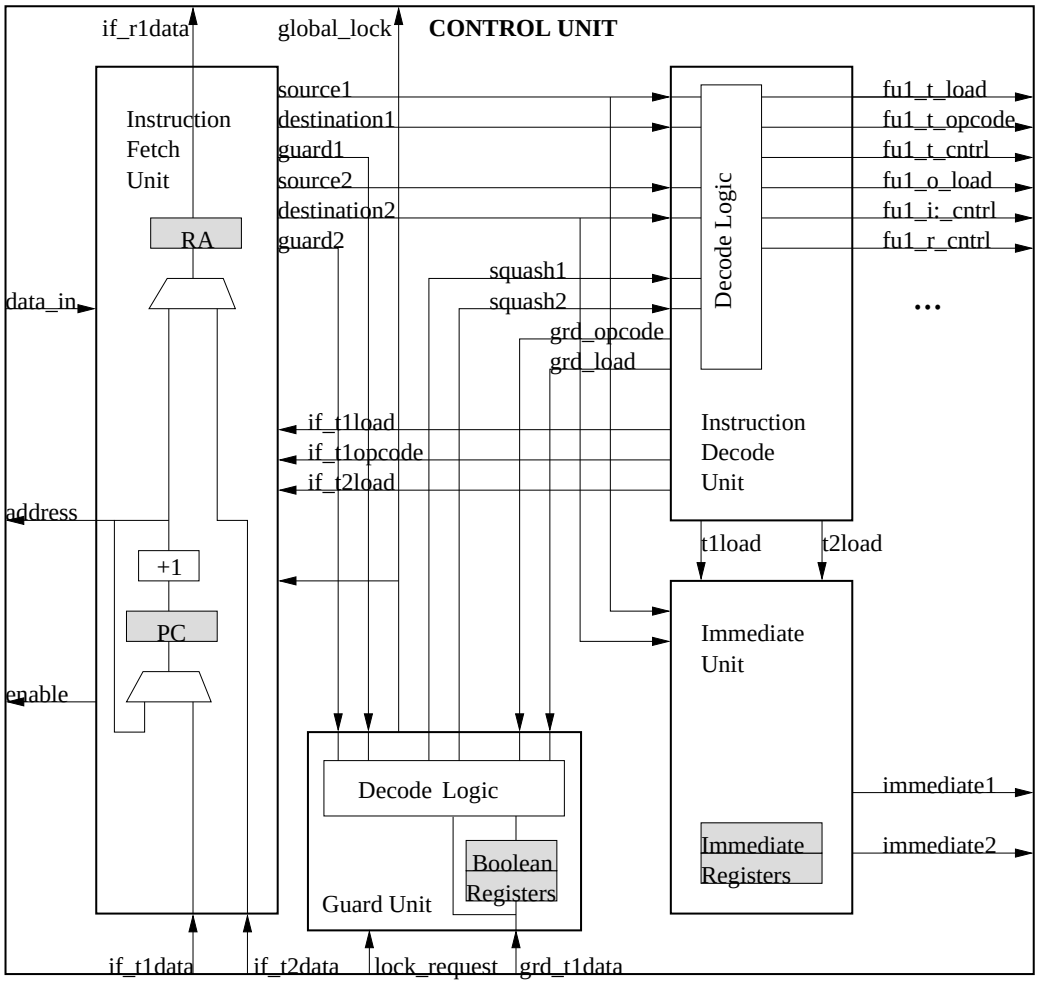

Figure 26. *Control unit of processor with two buses.*

the control signals for multiplexers and demultiplexers of the interconnection network are generated at the instruction decoding. Fig. 27 illustrates a section of VHDL code for instruction decoding, corresponding to an input socket. The address of the destination field is compared against the hardwired socket ID for each bus the socket is connected to. If the contents of the destination field matches with the ID the control registers of the interconnection network as well as the opcode registers are updated accordingly when the instruction to be decoded is at decode stage of the instruction pipeline. If there is no match for any of the connections these registers retain their value from the previous clock cycle, which reduces power consumption. The (semi virtual-time latching) pipeline control register, `as0_t_load_reg`, has to be always inactive if none of the IDs match. The pipeline control signals and opcodes are taken from the corresponding registers to functional units and register files, i.e., outside the control unit. Similarly, the control signals to multiplexer and demultiplexer are redirected from the control unit to the interconnection network. Due to the lack of space, a complete set of control signals

```
if ("00"&dst_id1(5 downto 2)&"00" = X"20"
    and squash1 = '0') then
  as0_t_cntrl_reg  <= "00";
  as0_t_opcode_reg <= dst_id1(1 downto 0);
  as0_t_load_reg   <= '1';
elsif ("00"&dst_id3(5 downto 2)&"00" = X"1c"
       and squash3 = '0') then
  as0_t_cntrl_reg  <= "01";
  as0_t_opcode_reg <= dst_id3(1 downto 0);
  as0_t_load_reg   <= '1';
elsif ("00"&dst_id4(5 downto 2)&"00" = X"18"
       and squash4 = '0') then
  as0_t_cntrl_reg  <= "10";
  as0_t_opcode_reg <= dst_id4(1 downto 0);
  as0_t_load_reg   <= '1';
else
  as0_t_load_reg <= '0';
end if;
```

Figure 27. *VHDL code for instruction decoding unit*

is shown only for one functional unit, fu1, in Fig. 26.

4.3 Functional Unit Library

In order to utilize the designed processor generator in automated hardware design of TTA processors, functional units and register files, instantiated in the generated top-level netlist, had to be designed and implemented. Since the processor generator is targeted for standard cell design methodology, also the functional units and register files were designed from that aspect. Therefore, the functional units were described in register transfer level behavioral VHDL. This enables design of relatively large set of units within relatively short period of time. When a functional unit containing a datapath element is designed in full custom techniques the unit has to be distinguished not only by its architectural parameters but also on physical design constraints such as minimum delay. The RTL code is more flexible in this respect. As well as portable to different process technologies, the same RTL code can be used to describe a unit with different design goals like high speed or small silicon area. It is up to the synthesis tool to determine the specific architecture of datapath components. For example, a multiplier is presented by a "*" operator in RTL code. Whether a Wallace tree or a carry-save array multiplier is selected during synthesis, depends on the design constraints given to the synthesis tool.

The designed functional units and register files are organized as a library, meaning that

they are stored in common database which can shared by several designers who work with the processor generator. The library can be physically a directory in file system or a compiled, tool specific object file. Once designed and placed or compiled in the library the designed FU can be used in several processor designs. Since the RTL hardware description language code for the functional units is intended to be used by various designers and design automation tool and with different target technologies, the coding style had to follow the guidelines set for good, reusable HDL code. A clear naming conventions for the VHDL constructs in the functional unit designs was used consistently throughout the library. A header was included in each source VHDL source code file where the filename, version history and information about functional unit is given. Moreover, generics and constants were used extensively instead of hardcoded literals. [21] discusses the coding practices for reusable RTL code in more depth.

The designed library contains functional units that cover all integer operations that are supported by the front-end of the software subsystem of the MOVE framework. With this set of units, a TTA processor capable of executing any C language application can be designed. Most of the designed functional units support more than one operation. This enables operations to share sockets and input registers. For some operations, it is also possible to share the core logic, on condition that the logic is not pipelined. For example, addition and subtraction can be performed with one adder and some extra logic and therefore these two operations can share a functional unit without any additional cost. Only semi virtual-time latching pipelining was utilized designed functional units due to its simple and efficient hardware structure.

A VHDL template for a generic SVTL functional unit with latency of three cycles is presented in Appendix A. Fig. 28 depicts a block diagram corresponding to the functional unit template. At default, the template corresponds to a following entry in the machine description file:

```
always, 3, { fu1_o }, fu1_t, { fu1_r}, {op1,op2};
```

The VHDL template understandably respects the interface requirements set by the processor generator. As was described in Section 2.2.3, the functional unit contains an input register for each trigger input and operand input. In the VHDL template the signals that correspond to trigger register, the first operand register and the first result register are `t1_reg`, `o1_reg` and `r1_reg`, respectively. If the interface of FU is modified the input and output registers described in the VHDL has to be modified accordingly. A separate sub-block can be designated for the core functional logic. This logic can have internal pipeline stages, given that the pipelines for each opera-

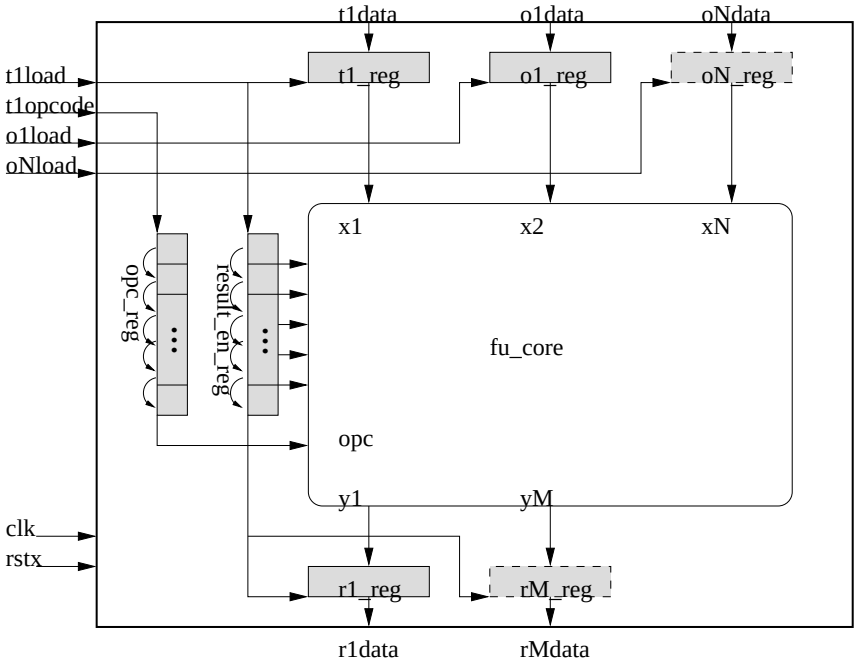

Figure 28. *Functional unit block diagram.*

tions performed at the logic have equal depths. The latency of the functional unit is 2+#*internal pipeline stages*. The core logic has to designed be the user, or alternatively a suitable Synopsys DesignWare[22] component can be applied. If the functional unit supports more than one operation, a shift register chain of length *latency*-1, named `opc_reg`, is needed to store the opcodes of last *latency*-1 cycles. The last register in the chain is used to select the result of operation presented by the opcode, to written to the result registers and further to FU output. The selection should be done in the core logic if possible. Another shift register chain, `result_en_reg`, is needed to store the status of `t1load` signal, i.e. the activity of the FU for the last *latency*-1 cycles. The last bit of the chain controls the latching of the data from the functional core logic to the result register(s), so that new data is latched precisely *latency*-1 cycles after a operation was triggered and otherwise the old value is retained.

There is less design freedom on the register files compared to the functional units. The compiler of the MOVE framework assumes that the register files have fixed latency of one. The only architectural parameters of the register files are thus the number read and write ports and the number of GPRs. The HDL code for such register file is extremely regular. Due to this, a simple generator script, again in Python, was written to generate the VHDL code for a register file. The number of read and write ports are given to the script as command line parameters. Register file size is a generic parameter, which is fixed when the register file is instantiated in the processor design. As an example,

generated VHDL code for a register file with two write ports and two read ports is given in Appendix B.

5. IMPLEMENTATION EXPERIMENTS

The developed processor generator, discussed in chapter 4, was used together with the MOVE framework to design a bundle of transport triggered architecture ASIPs with different implementation and architectural parameters. The implemented processors were evaluated in terms of delay, silicon area and power consumption, depending on evaluation objectives. This chapter describes the implementations and the obtained results. Section 5.1 discusses the general performance of the TTA template. In Section 5.2, an important power saving method, clock gating, is applied on TTAs. Bus structures supported in the processor generator are compared in Section 5.3.

In principal, the implementation flow is similar to conventional a standard cell ASIC design flow, which is thoroughly explained, for example, in [23]. The VHDL code obtained from the processor generator and predesigned libraries were used as design entry. Functional verification and register transfer level simulation of the VHDL code was performed using ModelSim mixed-language simulator (version 5.7c, 32-bit Linux-platform). Synopsys Design Compiler (version 2002.05, 32-bit Linux-platform) was used to translate the VHDL code to a netlist, composed of cells from modern (0.13 μm, low-K dielectric, six layers of copper wiring, V_{DD} 1.5 V) standard cell library. All the presented implementation characteristics are acquired from the netlist-level designs using Design Compiler as an analysis tool. Switching activities required in power analysis were obtained from gate-level simulation run on ModelSim. All delay calculations are based on the worst case operating conditions, i.e., weak process, 125°C temperature, and operating voltage of 1.35 V.

5.1 Performance Evaluation

Several application-specific processors were implemented to give insight on the hardware aspects of standard cell based TTA processors. Motivation for the implementations was to (a) verify that TTA architecture is actually a feasible choice for high performance low-cost, embedded processor, (b) find the critical components or bottlenecks of the

TTA processor in terms of cost and performance. Three small DSP applications were used as benchmarks, for which the processors were optimized using the design space explorer of the MOVE framework.

The first benchmark is an 8×8 discrete cosine transform (DCT) realized with row-column approach, i.e., the entire two-dimensional (2-D) transform is computed with the aid of 1-D transforms. Here the constant geometry algorithm proposed in [24] has been used. Constant geometry algorithms being regular and modular allow better exploitation of the inherent parallelism.

The second benchmark is a 32-point DCT, where DCT algorithm described in [25] is used. The created C-code contains five functions, one for each processing column of the signal flow graph of the algorithm. Each processing column is written totally unrolled, i.e., no iterations are used. This way the MOVE compiler was able to detect and exploit the inherent parallelism of the algorithm. On the other hand, this sort of code results in larger program code. Both the DCT applications were described in C language using fractional data type, i.e., fixed-point representation where the number range is normalized. Such a data type is often used in DSP realizations but it is difficult to exploit in C compilers because the ANSI C does not contain predefined data type for fractional representation.

The third benchmark is Viterbi decoding [26], an algorithm widely used in many decoding and estimation applications in the communications and signal processing domain. The algorithm decodes 256-state 1/2-rate convolutional codes and, contains path metric computation and survivor path search. This algorithm, also written in C, contains more complex control flow; conditional statements are also needed.

For each application, three configurations, high-performance, medium-size, and cost-efficient, were manually selected from the set of Pareto points found in resource optimization phase of the design space exploration. The high performance configuration corresponds to processor, which has a large pool of hardware resources and thus the instruction scheduler can exploit almost all the instruction level parallelism available in application. In this configuration, the number of clock cycles is minimized but attainable clock cycle may be reduced due to the hardware complexity. In the cost-efficient configurations the amount hardware resources are is lowered to minimum so that code scheduling can be barely performed. This kind of configuration suits for applications that do not have very high throughput requirements. The medium configuration is a compromise between the high-performance and cost-efficient configurations. On the medium-size configuration, the hardware resources were reduced significantly com-

Table 2. *Processor configurations*

Application	**32-point DCT**		
Configuration	High-performance	Medium	Cost-efficient
Functional Units	2 Adders 1 Multiplier 2 Shifters 1 Load-Store	1 Adder 1 Multiplier 1 Shifter 1 Load-Store	1 Adder 1 Multiplier 1 Shifter 1 Load-Store
Interconnection Network	9 Buses	7 Buses	2 Buses
Register Files	2×1, 5×2 12 GPRs in total	2×1, 5×2 12 GPRs in total	3×1, 4×2 11 GPRs in total
Application	**8×8 2-D DCT**		
Configuration	High-performance	Medium	Cost-efficient
Functional Units	4 Adders 1 Multiplier 4 Shifters 1 Comparator 1 Sign-extend 2 Load-Stores	2 Adders 1 Multiplier 1 Shifter 1 Comparator 1 Sign-extend 2 Load-Stores	1 Adder 1 Multiplier 1 Shifters 1 Comparator 1 Sign-extend 2 Load-Stores
Interconnection Network	9 Buses	5 Buses	4 Buses
Register Files	7×12, 1×16 100 GPRs in total	4×4, 4×6 40 GPRs in total	6×2, 2×3 18 GPRs in total
Application	**Viterbi Decoding**		
Configuration	High-performance	Medium	Cost-efficient
Functional Units	2 Adders 2 Logic 4 Shifters 2 Comparators 1 Sign-extend 2 Load-Stores	1 Adder 1 Logic 1 Shifter 1 Comparator 1 Sign-extend 1 Load-Store	1 Adder 1 Logic 1 Shifters 1 Comparator 1 Sign-extend 2 Load-Stores
Interconnection Network	6 Buses	5 Buses	3 Buses
Register Files	8×8 64 GPRs in total	4×6, 4×8 56 GPRs in total	6×4, 2×6 40 GPRs in total

pared to the high-performance configuration, but the number of clock cycles is within 20–30 percents of clock cycles of the high-performance configuration. In general, the medium-size configuration is expected to offer the best cost/performance ratio.

The three configurations optimized for the three testbench applications, are described in Table 2. The processor configurations for the two DCT applications have very similar set of functional units, as both of the algorithms can be presented as signal flow graphs composed of additions/subtractions and multiplications. The shifter is required for scaling the numbers in fractional representation to avoid overflows in additions. The comparator is required in 8×8 DCT to test the loop conditions. In Viterbi decoding, a multiplier is not needed at all but the comparator is used in add-compare-select operations in addition to loop status checks. Furthermore, a logic unit and shifter are needed because bit-level manipulations are performed. It is notable that for none of the configurations have particularly large set of functional units. On the composition of the register files, the deviation between the processor configurations is more distinguishable. The Viterbi decoding seems to have the largest register pressure as even the very cheap configuration needs a large number of general-purpose registers.

5.1.1 Full Connectivity

First, the nine processor configurations with fully connected interconnection network were implemented and analyzed. Full connectivity guarantees reprogrammability, i.e., a processor with an interconnection network where all the sockets are connected to all the buses can execute any application, given that the functional units provide the required operations. As can be seen from Table 2, all the configurations of 32-point DCT are subsets of all the configurations of 8×8 DCT. Reprogrammability can be an advantageous property, for example, if a processor is used as a hardware accelerator on a multimedia chip. Such a processor can be adopted to DSP tasks of future audio and video standards by means of a software update. The results obtained from the analysis of the implementations of the configurations of full connectivity are presented in Table 3.

For all the processor configurations, the design goal was to achieve the highest possible clock frequency in order to find the maximum throughput for the testbench applications on the MOVE architecture. This may result in lower energy efficiency and increased area (gate count). The achieved clock frequencies range from 147 to 268 MHz. The critical path of all the evaluated processor designs starts from the output register of the instruction decode logic where a control signal for output demultiplexer is generated. Then follows a route from the output of a functional unit or register file corresponding to the demultiplexer through the interconnection network to the input of guard unit in the control block. Since the bit written into a boolean register in a previous cycle has to be accessible in guard evaluation of the next instruction, bypassing logic is needed in

Table 3. *Implementation results (full connectivity)*

Application	**32-point DCT**		
Configuration	High-performance	Medium	Cost-efficient
Gate Count	71443	60585	26337
Clock Frequency / MHz	201	207	268
Throughput / Samples/s	16.9	16.2	9.5
Power / mW	45	40	20
Power / μA/MHz	150	130	50
Energy Efficiency / MSamples/J	380	410	490
Application	**8$\times$8 2-D DCT**		
Configuration	High-performance	Medium	Cost-efficient
Gate Count	174075	77347	55438
Clock Frequency / MHz	147	205	245
Throughput / MPixels/s	8.53	9.20	5.90
Power / mW	97	59	40
Power / μA/MHz	440	190	110
Energy Efficiency / MPixels/J	90	160	150
Application	**Viterbi Decoding**		
Configuration	High-performance	Medium	Cost-efficient
Gate Count	121624	98036	54151
Clock Frequency / MHz	181	191	184
Throughput KBits/s	85.4	74.3	39.1
Power / mW	67	62	32
Power / μA/MHz	250	220	120
Energy Efficiency / MBit/J	20	19	20

guard unit. This bypassing logic is also part of the critical path, followed by the logic used in guard expression evaluation and finally the instruction decoding logic. Due to the required bypassing the MOVE architecture contains a path that is actually spanned over two stages of the transport pipeline, namely the move stage and decode stage (of next instruction). It needs to be noted that both, the path through the actual interconnection network and the path through the instruction decoding logic get more complex as connections between sockets and buses are added. To conclude, the processor architecture, not the performance of the arithmetic units limits the lowest attainable cycle time. The obtained clock frequencies were somewhat lower compared to state of the art standard cell processors [19].

For 32-point DCT and Viterbi decoding, the best overall throughput for each application

was obtained with the largest configurations although they cannot run with as high clock frequencies as the smaller configurations. For 8×8 DCT the highest megapixels-per-second ratio was achieved with the medium configuration. On all applications, the low-cost configurations resulted in the lowest throughput.

5.1.2 Optimized Connectivity

Full connectivity in the interconnection network results in complex hardware; specially in processors, which have many transport buses. If a processor is designed for strictly application-specific purposes, it is profitable to tailor the interconnection network for the given application. Little, if any, modifications is allowed in the application code if it is expected to be compiled to a processor with reduced connectivity between buses and sockets. The connectivity exploration (Section 3.2.2) of the design space explorer was used to optimize the interconnection networks of the nine processor configurations (Table 2).

Again, the resulting configurations from the connectivity optimization were synthesized and optimized for the maximum clock frequency. After that, the synthesized processor designs were analyzed. The essential characteristics of the implementations are collected to Table 4. Connectivity optimization provided significant improvement on the performance of the processor designs.

As with the processors with full connectivity, the critical path of the all the processors with optimized connectivity runs from a FU output through the interconnection network and guard evaluation logic to the instruction decoding. On processors optimized for the DCT benchmarks the unpipelined 32-bit multiplier with 32-bit output was also optimized close to the limits of the technology. Less connections in datapath of transport network results in faster and smaller multiplexers and demultiplexer. Similarly, control logic is simplified because the range of addresses identified in instruction decoding is smaller. Due to this, the delay of on critical path of the was reduced and higher clock frequencies, ranging from 216 to 293 MHz, could be attained. On average, removing unnecessary connection from the interconnection network resulted in gain of 33 percents to clock frequencies. Consequently, the throughput of the applications run on the processors, was equally increased as the number of clock cycles spent on the computation was not changed.

In five out of nine configurations, the dynamic power consumption was reduced compared to their fully connected counterparts, despite the higher clock frequencies. Signifi-

Table 4. *Implementation results (optimized connectivity)*

Application	**32-point DCT**		
Configuration	High-performance	Medium	Cost-efficient
Gate Count	39345	31061	23474
Clock Frequency / MHz	293	277	270
Throughput / Samples/s	24.7	21.7	9.8
Power / mW	38	43	18
Power / μA/MHz	86	78	44
Energy Efficiency / MSamples/J	650	670	540
Application	**8×8 2-D DCT**		
Configuration	High-performance	Medium	Cost-efficient
Gate Count	157560	61082	43018
Clock Frequency / MHz	240	290	272
Throughput / MPixels/s	13.8	13.5	6.6
Power / mW	107	62	35
Power / μA/MHz	298	141	85
Energy Efficiency / MPixels/J	129	219	190
Application	**Viterbi Decoding**		
Configuration	High-performance	Medium	Cost-efficient
Gate Count	104362	83643	49187
Clock Frequency / MHz	256	216	269
Throughput KBits/s	116.3	83.7	59.3
Power / mW	75	52	37
Power / μA/MHz	195	161	92
Energy Efficiency / MBit/J	25	26	25

cantly lower normalized power figures (mA/MHz) and greatly improved energy efficiencies illustrate the benefits of the reduced interconnection network even more distinguish ably.

5.2 Clock Gating Results

In many VLSI chips, power dissipation of the clocking system is often the largest portion of total chip power consumption because the switching activity of clock networks are equal to one and the total node capacitance is high due to the large number of clocked nodes.

Clock gating is a design strategy that allows to mitigate the switching activity of the

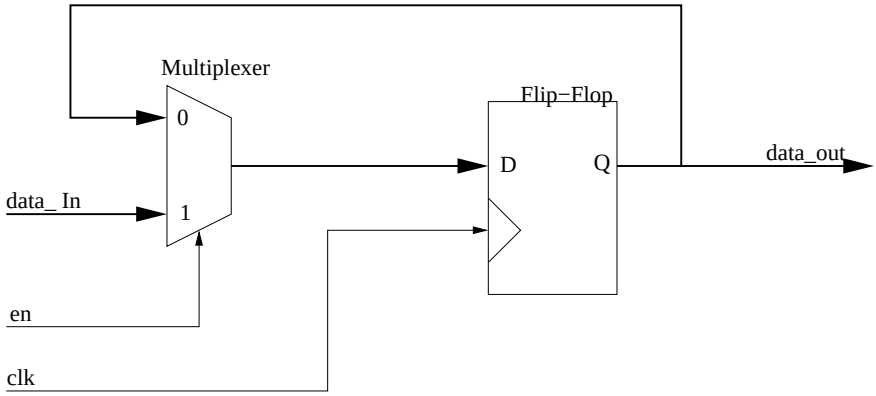

Figure 29. *Register bank with load enable using multiplexer*

clock tree and its leaf registers. In order to reduce the activity of the clock node of a register bank the clock node is enabled only when the register bank has to sample new input. [27]

Without clock gating register banks are implemented by using a feedback loop and a multiplexer. When such registers maintain the same value through multiple cycles, they use power unnecessarily. Fig. 29 shows a simple register bank implementation using a multiplexer and a feedback loop.

Clock gating means inserting a specific circuitry into the clock network of the register bank and creating the control to eliminate unnecessary register activity. This principle is depicted in Fig. 30. Clock gating also reduces the clock network power dissipation, relaxes the data path timing, and reduces routing congestion by eliminating feedback multiplexer loops. For designs that have large multi-bit registers, clock gating can save power and reduce the number of gates in the design. However, for smaller register banks, the overhead of adding logic to the clock tree might not compare favorably to the power saved by eliminating a few feedback nets and multiplexers. The insertion of the clock gating circuitry can be performed automatically by Synopsys Power Compiler tool during the logic synthesis process and thus it does not have to be considered in the HDL code. [22]

The effect of clock gating on transport triggered architectures was evaluated on three processor configurations optimized for 8×8 DCT. The three configurations located in different points of cost-performance space, are identical to ones used in performance evaluations presented in Section 5.1. The high-performance, medium and cost-efficient have 156, 82 and 59 register banks of full word width (32-bits), respectively. The largest portion of the register banks belong general-purpose registers located in register files.

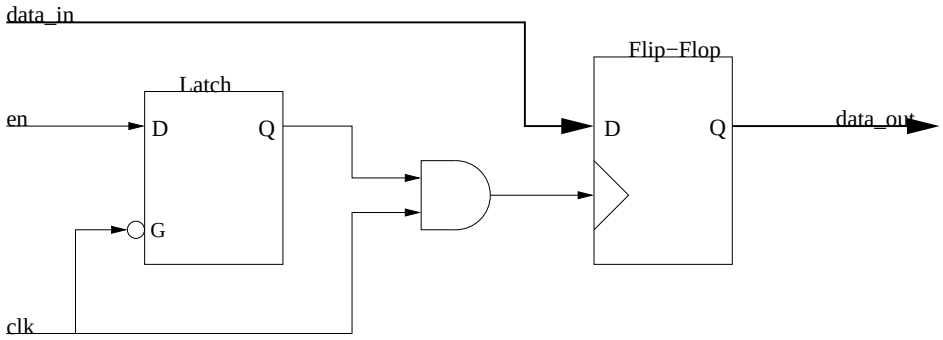

Figure 30. *Register bank with gated clock*

The rest of the register banks are trigger, operand, and result registers of the functional units. For all three configurations, a version which has unmanipulated clock tree, and a version on which clock gating was applied, was synthesized and optimized with clock frequency constraint of 100 MHz. Gate counts and power consumption figures for the processor designs are illustrated in Table 5 for comparison.

Utilizing gated clocks proved to be a very efficient method to save energy in the processor cores as power consumption was 34–37 percents lower when clock gating was applied. As was predictable, the largest energy savings were achieved on the register files. On each processor configuration their power consumption was lowered more than 65 percent by deactivating the clock signal when none of the registers in a RF needs to be refreshed. As was the case with register files, also the power consumption of the functional units was reduced considerably. However, for energy efficiency of the control unit the clock gating did not give such a large improvement. This can be explained by the fact that control unit, specifically the instruction decoding contains large number of independent registers that cannot be composed into a bank controlled by single enable signal. Moreover, also the silicon area of the cores, which is directly proportional to gate count, was reduced.

5.3 Bus Structure Comparison

In Chapter 4, it was discussed that demultiplexing in the interconnection network established using tristate drivers is not a feasible solution in modern standard cell designs. The demultiplexer structures supported in the processor generator were evaluated in order to determine whether the two new interconnection structures introduced in Section 4.2.2 are suitable candidates to replace the tristate bus.

Table 5. *Clock gating results*

High-performance		
	No Clock Gating	Gated Clock
Power / mW	36	23
Gate Count / kgate	76.4	66.5
Medium		
	No Clock Gating	Gated Clock
Power / mW	19	13
Gate Count / kgate	37.5	32.0
Low-Cost		
	No Clock Gating	Gated Clock
Power / mW	13	8
Gate Count / kgate	26.0	22.6

As before, the three processor configurations adapted for the application realizing 8×8 DCT were used in evaluation of the bus structures. For all three configurations, a revision for each demultiplexing method was implemented. Furthermore, each configuration was realized with fully connected as well as optimized transport network using VHDL code obtained from the processor generator and the library of functional units as a design entry. Clock frequency of 100 MHz was used as an optimization constraint in logic synthesis step where the HDL code was transformed into a technology specific netlist. By analyzing and simulating the gate-level netlist the area and power characteristics, shown in Table 6, were obtained.

On each realized processor configuration using both AND-OR network and multiplexer based bus write selection resulted in smaller gate counts compared to configurations that employed tristate buses. On the fully connected configurations, AND-OR demultiplexing resulted on average 50 percent reduction in gate count of interconnection network and almost as much on configurations with reduced connectivity. Both AND-OR network and tristate bus have identical instruction decoding logic as similar enable signals are used to control both tristate buffers and AND-gates. Therefore, there was practically no difference in the gate count of control logic of tristate buses and AND-OR network — all the area saving was achieved in datapath. Processor designs with AND-OR network as demultiplexing structure resulted in average gate count reduction of 20 percent on the fully connected designs and 8 percent in the connectivity optimized designs.

Also the multiplexer based bus write resulted in smaller gate counts in comparison to the tristate bus. The area reduction was not, however, quite as large as with AND-OR

Table 6. *Bus structure comparison results*

Full Connectivity			
High-performance			
Bus Structure	AND-OR	Tristate	Multiplexer
Power / mW	37	38	47
Gate Count / kgate	87.3	107.3	99.2
Medium			
Bus Structure	AND-OR	Tristate	Multiplexer
Power / mW	17	17	19
Gate Count / kgate	37.9	47.9	38.7
Low-Cost			
Bus Structure	AND-OR	Tristate	Multiplexer
Power / mW	10	10	11
Gate Count / kgate	26.2	32.6	26.1
Optimized connectivity			
High-performance			
Bus Structure	AND-OR	Tristate	Multiplexer
Power / mW	23	21	25
Gate Count / kgate	66.5	68.9	67.9
Medium			
Bus Structure	AND-OR	Tristate	Multiplexer
Power / mW	13	12	12
Gate Count / kgate	32.0	34.7	31.6
Low-Cost			
Bus Structure	AND-OR	Tristate	Multiplexer
Power / mW	8	8	8
Gate Count / kgate	22.6	25.9	22.5

network. This is due to the fact that logic that selects, which functional unit output drives the bus using the source id as a control signal could not be to generated as effectively at the logic synthesis. The instruction decoding logic was reduced, as a part of the instruction decoding is performed at interconnection network. Nevertheless, this did not compensate larger area originating from more complex transport network.

There is virtually no difference in power consumption of the processors utilizing AND-OR demultiplexing and tristate buffers. In general, multiplexer based bus resulted in slightly higher power consumption but differences were not significant compared to the other two bus structures, except on the high-performance configurations.

5.4 Discussion

To obtain even higher clock frequencies than presented in Section 5.1.2 the delay on the critical path through the transport network can be reduced by assuming that the 1-bit boolean values to the guard unit are always transported directly from the comparator unit(s) and never from the functional units and register files processing and storing data of full word width. To alleviate the bottleneck the boolean values can now be transported through dedicated 1-bit buses, which have only a small number of connections as the number of the comparators and guard unit input sockets is typically very small. Sometimes even a direct connection from the result register of the comparator unit to input of guard unit is sufficient. Due to the very low number of connections the capacitive load on these 1-bit buses is low and therefore the delay is decreased. The effect of the 1-bit dedicated bus for the boolean values was tested on the medium and cost-efficient configuration for the Viterbi decoding benchmark. The implementations resulted in clock frequencies of 286 MHz for the medium configuration and 308 MHz for the cost efficient configurations. The improvement to the processor designs where boolean values are transported on normal transport buses was 24 percent and 17 percent, respectively. The comparisons are not, however, completely justifiable because the configurations where the 1-bit bus was utilized were obtained through new connectivity exploration form the fully connected configuration as the compiler was not able to generate code for the original connectivity optimized processors, on which a bus from the output of the comparator to guard unit was added afterwards. A point worth mentioning is that the number of clock cycles was also decreased when dedicated bus for the comparator results was available.

As CMOS process geometrics shrink, the delay attributable to interconnection, i.e., the wiring between the active devices, is starting do dominate over the gate delay. Designers have had to take into account the effect of wire on the large chip level designs. When the effective gate length is decreased to 90 nm and below the effect of wiring has to be considered even on designs of gate count under 100 kgates, such as the MOVE processors cores discussed in this chapter. [28] [29]

During the synthesis the capacitances and thus the delays associated with final wiring are unknown. Models of interconnect, known as wire-load models [29], attempt to predict the amount of capacitance in a wire by reducing it to function of fanout and block size. In this approach a single capacitance value is used for all nets in a block with the same fanout. Obviously, the capacitance values for nets in a block will vary and so the wire-load model is necessarily only an approximation of the actual future capacitance. If the

capacitance in wires increase, then wire-load models become increasingly inaccurate.

Another problem rises on defining the blocks or hierarchy during the synthesis. A logical block, such as the transport network of TTA processor may not result in an uniform region of placed gates on silicon. For this reason, the wire load model of the interconnection network of the processor designs presented in this chapter was enlarged to cover the entire core area. It is likely that many of the buses span over the entire core but for some of the buses such assumption may be too pessimistic. Such an oversizing of wire-load model can skew the results of bus comparison as one interconnection structure may be more sensitive to the increase of bus load than other.

On the other hand, assuming that the resulting placement of cells for some other building blocks, e.g., register files, reside on a restricted region may have been too optimistic. These problems can get even worse due to increased ratio of coupling capacitances to substrate capacitances caused by reduction of wire aspect ratios. In general, the effect of wiring undermines the reliability of the implementation results presented in this chapter.

6. CONCLUSIONS

In this thesis, design of a processor generator for transport triggered architecture template of the MOVE framework was described. Important design goal for the developed processor generator was to improve the usability and reliability of the MPG, the original processor generator of the MOVE Framework. This was achieved by reading the properties of the target processor from the machine description file and map file. These files respectively contain information the on the structure and binary encoding of the target processor, and are used as information exchange format by rest of the tools of the MOVE framework. Support for three different demultiplexing structures was also incorporated to the processor generator. Furthermore, the inherent modularity of TTA is exploited by defining a clear and consistent interface for the functional units and register files instantiated by the processor generator. A prototype functional unit honoring this interface specification employing the semi-virtual time latching pipeline control discipline was also developed. Given this prototype, a set of functional units realizing all the integer operations supported by the software subsystem of the MOVE framework was designed. Moreover, a generator script for the register files was written.

Using the design space explorer of the MOVE framework a set of processor configurations for three DSP benchmark applications, 32-point DCT, 8×8 DCT, and Viterbi decoding, were obtained. To examine the hardware characteristics and performance of the TTA template of the MOVE framework, these processor configurations were implemented using the VHDL code produced by the processor generator as the design entry. VHDL code for each processor configuration was synthesized to modern 0.13 μm standard cell technology and the obtained to netlists were analyzed to gain information on the performance, silicon area, and power consumption of the processor designs. Furthermore, combining the hardware performance and the cycle count statistics from the software subsystem, the performance of the test applications on the processor designs was measured. On processor configurations with fully connected network the maximum clock frequencies were somewhat lower than clock frequencies of the fastest processors implemented with standard cell technology. By optimizing the connectivity of the interconnection network significant gain was achieved on performance and energy effi-

ciency of the processors. By analyzing the critical path of the processor, it was found that on the Viterbi decoding where no multiplier was required, the architecture rather the performance of the arithmetic units limits the attainable clock frequency. The delay of the critical path can be alleviated by dedicating 1-bit buses for boolean value transports from the comparator to the guard unit. In addition to performance analysis, three bus demultiplexing structures, tristate, AND-OR, and multiplexer-based, were evaluated and compared. Although the difference between the bus structures in terms of power consumption and gate count was found quite small, the AND-OR structure performed slightly better than the multiplexer based bus and thus it can proposed as the replacement for the tristate bus in standard cell implementations. Lastly, clock gating, a widely used power saving method in standard cell designs, was evaluated on TTAs. It was discovered that applying clock gating not only results in considerably lower power consumption but it also reduces the silicon area of the processor designs.

The developed processor generator produces functionally correct VHDL code that can be synthesized efficiently. The tool can be, however, improved and developed further in number of ways. The generated control unit currently contains very simple instruction fetch unit that has a fixed interface for a clocked ROM, from which the instructions are read. Different memory hierarchies, i.e., caches, and memory interfaces should be supported in flexible manner as well as program code decompression circuitry. There exist plenty of possibilities to enhance the usability of the processor generator. For example, user-friendly interface to add and manage user defined functional units in the library of FUs could be developed. In addition, third party design automation tools should be more tightly incorporated in the processor generator. This would mean that makefiles, synthesis scripts and setup files required to import and compile the produced VHDL code should be generated automatically.

To validate the results obtained from the analysis of the netlist level processor designs the should be elaborated closer to real physical implementations. By performing standard cell placement and initial routing more accurate information on wiring capacitances is available. The delay calculations based on physical placement of the gates provides more reliable information on the true timing bottlenecks of the processor designs. When the placed-gates and netlist-level designs are compared it is possible to evaluate whether the netlist and correctly chosen wire-load models provide sufficiently accurate information on the physical characteristics of the processor designs.

BIBLIOGRAPHY

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

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

[3] S.Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *Proc. of 6th Annual Int. Symp. on High-Performance Computer Architecture*, 2000, pp. 375–386.

[4] V. S. Lapinskii, M. F. Jacome, and G. A. de Veciana, "Application-specific clustered VLIW datapaths: early exploration on a parameterized design space," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 8, pp. 889–903, 2002.

[5] A. Cilio, "Code generation and optimization for embedded processors," Ph.D. dissertation, Delft Univ. Tech., Delft, The Netherlands, Sep. 2002.

[6] J. Hoogerbrugge, "Code generation for transport triggered architectures," Ph.D. dissertation, Delft University of Technology, Delft, The Netherlands, Feb. 1996.

[7] H. Corporaal and J. Hoogerbrugger, "Register file port requirements of transport triggered architectures," in *27th Annual Workshop on Microprogramming*, 1995, pp. 191–195.

[8] H. Corporaal and J. Janssen, "Partitioned register files for TTAs," in *28th Annual Workshop on Microprogramming*, 1996, pp. 303–312.

[9] J. Janssen, "Compilation strategies for transport triggered architectures," Ph.D. dissertation, Delft University of Technology, Delft, The Netherlands, Sep. 2001.

[10] H. Corporaal and P. A. Arend, "MOVE32INT, a sea of gates realization of a high performance transport triggered architecture," *Microprocessing and Microprogramming*, no. 38, pp. 53–60, Sep. 1993.

[11] J. M. Rabaey, *Digital integrated circuits: a design perspective*. Upper Saddle River, NJ, U.S.A: Prentice-Hall, Inc., 1996.

[12] E. Aardoom and P. Stravers, "An application specific processor for a multi-system navigation receiver," in *Proc. of International Conference on Computer Design: VLSI in Computers and Processors*, 1992, pp. pp. 128–131.

[13] M. Arnold, "Instruction set extensions for embedded processors," Ph.D. dissertation, Delft University of Technology, Delft, The Netherlands, Mar. 2001.

[14] A. Smit, "The MPG manual," Master's thesis, Delft Univ. Tech., Delft, The Netherlands, Jun. 2000.

[15] S. Roos, "Design and implementation of an advanced instruction fetch unit for the MOVE framework," Master's thesis, Delft Univ. Tech., Delft, The Netherlands, Jun. 1997.

[16] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1993, 1994.

[17] *IEEE standard Verilog hardware description language*, IEEE Std. 1364-2001, 2001.

[18] G. van Rossum, *Python reference manual*, Centrum voor Wiskunde en Informatica, 1995.

[19] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC & Custom*. Boston, MA, U.S.A: Kluwer Academic Publishers, 2002.

[20] M. D. Ercegovac, J. H. Moreno, and T. Lang, *Introduction to Digital Systems*. New York, NY, U.S.A: John Wiley & Sons, Inc., 1998.

[21] M. Keating and P. Bricaud, *Reuse methodology manual: for system-on-a-chip designs*. Norwell, MA, U.S.A: Kluwer Academic Publishers, 1998.

[22] *Synopsys Online Documentation*, Synopsys, 2002.

[23] M. Smith, *Application-specific integrated circuits*. Boston, MA, U.S.A: Addison-Wesley Longman Publishing Co., Inc., 1997.

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

[25] J. Takala, D. Akopian, J. Astola, and J. Saarinen, “Constant geometry algorithm for discrete cosine transform,” *IEEE Trans. Signal Processing*, vol. 48, no. 6, pp. 1840–1843, June 2000.

[26] A. J. Viterbi, “Error bounds for convolutional coding and an asymptotically optimum decoding algorithm,” *IEEE Trans. Inform. Theory*, vol. 13, pp. 260–269, Apr. 1967.

[27] G. Palumbo, F. Pappalardo, and S. Sannella, “Evaluation on power reduction applying gated clock approaches,” in *Proc. IEEE Int. Symposium on Circuits and Systems*, 2002, pp. IV–85–IV–88 vol.4.

[28] P. Gopalakrishnan, A. Odabasioglu, L. Pileggi, and S. Raje, “An analysis of the wire-load model uncertainty problem,” *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, no. 1, pp. 23–31, Jan. 2002.

[29] D. Sylvester and K. Keutzer, “Getting to the bottom of deep submicron,” in *Proc. of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998, pp. 203–211.

Appendix A

FUNCTIONAL UNIT TEMPLATE

```
-------------------------------------------------------------------------------
-- Title      : Functional unit template
-- Project    : FlexDSP
-------------------------------------------------------------------------------
-- File       : fu_template.vhdl
-- Author     : Jaakko Sertamo  <sertamo@vlad.cs.tut.fi>
-- Company    : TUT/IDCS
-- Created    : 2003-03-11
-- Last update: 2003-09-24
-------------------------------------------------------------------------------
-- Description: Functional unit template for SVTL pipelined FU
--
-- This template can be used as a basis when a new functional unit needs to be
-- designed. At default, this template corresponds at a following definition
-- in the machine description file:
-- fu1      always, 3, { fu1_o }, fu1_t, { fu1_r}, {op1, op2};
--
-------------------------------------------------------------------------------
-- Copyright (c) 2003
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2003-03-11  1.0      sertamo Created
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- Package declaration for op1_op2 unit opcodes
-------------------------------------------------------------------------------

package op1_op2_opcodes is

  constant OPC_OP1 : integer := 0;
  constant OPC_OP2 : integer := 1;

end op1_op2_opcodes;


-------------------------------------------------------------------------------
-- Entity declaration for the functional unit
-------------------------------------------------------------------------------

library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_arith.all;
use work.op1_op2_opcodes.all;

entity fu_op1_op2_always_3 is
  generic (
    DATAW : integer := 32;
    BUSW  : integer := 32);
  port (
    -- clock and reset
```

```
    clk  : in std_logic;
    rstx : in std_logic;

    -- FU DATA INPUTS
    -- trigger input
    t1data : in  std_logic_vector (DATAW-1 downto 0);
    -- operand input(s)
    o1data : in  std_logic_vector (DATAW-1 downto 0);
    --oNdata : in  std_logic_vector (DATAW-1 downto 0);
    -- result output(s)
    r1data : out std_logic_vector (DATAW-1 downto 0);
    --rMdata : out std_logic_vector (DATAW-1 downto 0);

    -- FU CONTROL INPUTS
    t1opcode    : in std_logic_vector (0 downto 0);
    t1load      : in std_logic;
    o1load      : in std_logic;
    --oNload      : in std_logic;
    global_lock : in std_logic);

end fu_op1_op2_always_3;

architecture rtl of fu_op1_op2_always_3 is

  component userdef_logic
    generic (
      X1WIDTH : integer;
      X2WIDTH : integer);
    port (
      clk   : in std_logic;
      rst_n : in std_logic;

      x1  : in std_logic_vector(X1WIDTH-1 downto 0);
      x2  : in std_logic_vector(X2WIDTH-1 downto 0);
      y1  : in std_logic_vector(X1WIDTH-1 downto 0);
      --yM  : in std_logic_vector(X1WIDTH-1 downto 0);
      opc : in std_logic_vector(0 downto 0));
  end component;

  type std_logic_vector_array is array (natural range <>) of
    std_logic_vector(t1opcode'length-1 downto 0);

  -- REGISTERS
  -- trigger register
  signal t1reg : std_logic_vector (DATAW-1 downto 0);
  -- operand register(s)
  signal o1reg : std_logic_vector (DATAW-1 downto 0);
  --signal oNreg   : std_logic_vector (DATAW-1 downto 0);
  -- result register(s)
  signal r1reg : std_logic_vector (DATAW-1 downto 0);
  --signal rMreg   : std_logic_vector (DATAW-1 downto 0);

  -- opcode register
  signal opc_reg       : std_logic_vector_array(1 downto 0);
  signal result_en_reg : std_logic_vector(1 downto 0);

  -- WIRES
  signal r1 : std_logic_vector (DATAW-1 downto 0);
  --signal rM      : std_logic_vector (DATAW-1 downto 0);

begin  -- rtl

  -- Instantiate user defined component (Or a DesignWare component)
  fu_core : userdef_logic
    generic map (
      X1WIDTH => DATAW,
      X2WIDTH => DATAW)
    port map (clk   => clk,             -- userdef logic is pipelined
              rst_n => rstx,            -- userdef logic needs reset
              opc   => opc_reg(opc_reg'length-1),
              x1    => o1reg,
              x2    => t1reg,
              y1    => r1
              --yM  => rM
```

```
                );

        pipeline_control : process (clk, rstx)
130     begin  -- process pipeline_control

          if rstx = '0' then
            -- reset registers
            t1reg <= (others => '0');
135         o1reg <= (others => '0');
            --oNreg <= (others => '0');
            r1reg <= (others => '0');
            --rMreg <= (others => '0');

140         result_en_reg <= (others => '0');
            for i in 0 to opc_reg'length-1 loop
              opc_reg(i) <= (others => '0');
            end loop;  --

145       elsif clk = '1' and clk'event then

            if t1load = '1' then
              t1reg <= t1data;
            end if;
150
            if o1load = '1' then
              o1reg <= o1data;
            end if;

155         --if oNload = '1' then
            --   oNreg <= oNdata;
            --end if;

            for i in 0 to result_en_reg'length-1 loop
160           if i = 0 then
                -- latch data to register chain
                result_en_reg(0) <= t1load;
                opc_reg(0)       <= t1opcode;
              else
165             -- propagate register chain
                result_en_reg(i) <= result_en_reg(i-1);
                opc_reg(i)       <= opc_reg(i-1);
              end if;
            end loop;  -- i
170
            -- update result only when new operation was
            -- triggered \latency-1 cycles ago
            if result_en_reg(result_en_reg'length-1) = '1' then
              r1reg <= r1;
175           --rMreg <= rM;
            end if;

          end if;
        end process pipeline_control;
180
        -- connect registers to output ports
        r1data <= r1reg;
        --rMdata <= rMreg;

185   end rtl;
```

Appendix B

REGISTER FILE

```
-------------------------------------------------------------------------------
-- Title      : Register File for TTA
-- Project    : FlexDSP
-------------------------------------------------------------------------------
-- File       : rf_2wr_2rd.vhdl
-- Author     : Jaakko Sertamo  <sertamo@vlad.cs.tut.fi>
-- Company    : TUT/IDCS
-- Created    : 2003-15-04
-- Last update:
-------------------------------------------------------------------------------
-- Description: 2 Write port(s)
--              2 Read port(s)
-------------------------------------------------------------------------------
-- Copyright (c) 2003
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author   Description
-- 2003-15-04  1.0      sertamo   Created
-------------------------------------------------------------------------------
library IEEE;
use IEEE.Std_Logic_1164.all;
use IEEE.std_logic_arith.all;
use work.util.all;

entity rf_2wr_2rd is
  generic (
    dataw   : integer := 32;
    rf_size : integer := 8);
  port(
    t1data   : in std_logic_vector (dataw-1 downto 0);
    t1opcode : in std_logic_vector (bit_width(rf_size)-1 downto 0);
    t1load   : in std_logic;

    t2data   : in std_logic_vector (dataw-1 downto 0);
    t2opcode : in std_logic_vector (bit_width(rf_size)-1 downto 0);
    t2load   : in std_logic;

    r1data   : out std_logic_vector (dataw-1 downto 0);
    r1opcode : in  std_logic_vector (bit_width(rf_size)-1 downto 0);
    r1load   : in  std_logic;

    r2data   : out std_logic_vector (dataw-1 downto 0);
    r2opcode : in  std_logic_vector (bit_width(rf_size)-1 downto 0);
    r2load   : in  std_logic;

    rstx : in std_logic;
    clk  : in std_logic);
end rf_2wr_2rd;

architecture rtl of rf_2wr_2rd is
  type   reg_type is array (natural range <>) of std_logic_vector(dataw-1 downto 0);
  signal reg    : reg_type (rf_size-1 downto 0);
```

```
  signal r1_reg : std_logic_vector(dataw-1 downto 0);
  signal r2_reg : std_logic_vector(dataw-1 downto 0);

begin

  outputregs : process (clk, rstx)
    variable opc : integer;
  begin  -- process ouputregs
    if rstx = '0' then                  -- asynchronous reset (active low)
      r1_reg <= (others => '0');
      r2_reg <= (others => '0');

    elsif clk'event and clk = '1' then  -- rising clock edge
      if r1load = '1' then
        if t1load = '1' and t1opcode = r1opcode then
          r1_reg <= t1data;
        elsif t2load = '1' and t2opcode = r1opcode then
          r1_reg <= t2data;
        else
          opc := conv_integer(unsigned(r1opcode));
          -- pragma synthesis_off
          if opc > rf_size-1 then
            opc := rf_size-1;
          end if;
          -- pragma synthesis_on
          r1_reg <= reg(opc);
        end if;
      end if;

      if r2load = '1' then
        if t1load = '1' and t1opcode = r2opcode then
          r2_reg <= t1data;
        elsif t2load = '1' and t2opcode = r2opcode then
          r2_reg <= t2data;
        else
          opc := conv_integer(unsigned(r2opcode));
          -- pragma synthesis_off
          if opc > rf_size-1 then
            opc := rf_size-1;
          end if;
          -- pragma synthesis_on
          r2_reg <= reg(opc);
        end if;
      end if;
    end if;

  end process outputregs;

  regfile_write : process(clk, rstx)
    variable opc : integer;
  begin
    if rstx = '0' then
      for idx in (reg'length-1) downto 0 loop
        reg(idx) <= (others => '0');
      end loop;  -- idx

    elsif clk = '1' and clk'event then
      if (t1load = '1') then
        opc      := conv_integer(unsigned(t1opcode));
        reg(opc) <= t1data;
      end if;

      if (t2load = '1') then
        opc      := conv_integer(unsigned(t2opcode));
        reg(opc) <= t2data;
      end if;
    end if;

  end process regfile_write;

  r1data <= r1_reg;
  r2data <= r2_reg;

end rtl;
```