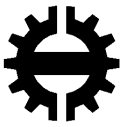


TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Electrical Engineering

RISTO MÄKINEN

FAST FOURIER TRANSFORM ON TRANSPORT TRIGGERED ARCHITECTURES

Master of Science Thesis

Subject approved by Department Council
17th Aug, 2005

Examiners: Prof. Jarmo Takala
M.Sc. Jari Heikkinen

PREFACE

This M.Sc. thesis was completed in Institute of Digital and Computer Systems of Tampere University of Technology (TUT) 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 giving me a chance to develop as a designer and for his patience as I have made myself thoroughly familiar with the transport triggered architecture concept. I want also thank him for guiding me to a fascinating area of research and for giving me the thread for my thesis. I am also extremely grateful to M.Sc. Jari Heikkinen for his help in the inspecting process and for his patience to answer numerous questions I have posed. Special thanks to M.Sc. Teemu Pitkänen for always having time to help with the hardware issues related to my work. I would also like to thank my colleagues for the inspired and motivated work atmosphere they have provided.

I am also grateful to my friends for giving me momentous, relaxing hours outside working days. Finally, I would like to thank my family, especially my deceased father, for their support throughout my studies.

Tampere, October 11, 2005

Risto Mäkinen

TABLE OF CONTENTS

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Program of Electrical Engineering
Institute of Digital and Computer Systems
Mäkinen, Risto Mikael: Fast Fourier Transform on Transport Triggered Architectures
Master of Science Thesis: 74 pages, 24 appendix pages
Examiners: Prof. Jarmo Takala and M.Sc. Jari Heikkinen
Funding: The National Technology Agency
Department of Electrical Engineering
October 2005
Keywords: transport triggered architecture, fast Fourier transform, performance optimisation

Application-specific instruction set processors are an interesting choice as processors are being selected for the needs of digital signal processing applications. These processors are programmable and their hardware can be tailored for the needs of the application to obtain both high performance and good area efficiency. However, these processors are hard to design since there can exist even hundreds of good processor configurations for a certain application in addition to which these processors have to be reprogrammed when the architecture is modified. Thus, effective design tools are needed to ease the design process of these processors. A semi-automated design environment, MOVE framework, was developed for designing application specific instruction set processors. Processors designed with the MOVE framework utilize the transport triggered architecture (TTA) paradigm where the program specifies only the data transports to be performed by the interconnection network. Actual operations occur as a side effect of these explicitly defined data transports.

In this thesis, fast Fourier transform (FFT) was implemented on TTA to evaluate TTA's performance in performing FFT. First, FFT was implemented using high-level language (HLL) code and the HLL compiler of the MOVE framework. Due to unfavourable performances of HLL implementations, FFT was implemented using assembly code in the next implementing phase. Based on the optimized assembler implementation, an effective TTA processor for performing FFT is proposed and compared with several other commercial and academic FFT processors to evaluate TTA's performance against other FFT processors.

TTA's performance in performing FFT was noticed to be extremely good when assembly code was used in programming. Even better performance indices than in the known, good application-specific integrated circuit implementations for FFT was obtained. As a conclusion, based on the results obtained in this thesis work, it can be stated that TTA is a promising programmable architecture candidate for implementing DSP applications.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Sähkötekniikan koulutusohjelma
Digitaali- ja tietokonetekniikan laitos
Mäkinen, Risto Mikael: Fast Fourier Transform on Transport Triggered Architectures
Diplomityö: 74 sivua, 24 liitesivua
Tarkastajat: Prof. Jarmo Takala ja DI Jari Heikkinen
Rahoitus: Teknologian kehittämiskeskus (TEKES)
Sähkötekniikan osasto
Lokakuu 2005
Avainsanat: transport triggered architecture, nopea Fourier-muunnos,
suorituskyvyn optimointi

Sovelluskohtainen käskykantaprosessori (Application Specific Instruction set Processor, ASIP) on mielenkiintoinen vaihtoehto, kun ollaan valitsemassa prosessoria digitaalisen signaalinkäsittelyn sovellusten tarpeisiin. ASIP-prosessorit ovat ohjelmoitavia ja niiden laitteistoresurssit voidaan räätälöidä vastaamaan sovelluksen tarpeita korkean suorituskyvyn ja tehokkaan pinta-alan käytön aikaansaamiseksi. ASIP-prosessorien suunnitteleminen on kuitenkin vaikeaa, koska tietylle sovellukselle voi olla olemassa jopa satoja hyviä prosessorikonfiguraatioita. Prosessori täytyy lisäksi ohjelmoida aina uudelleen, kun sen arkkitehtuuria on muutettu. Edellisten syiden takia tarvitaankin tehokkaita suunnittelutyökaluja helpottamaan ASIP-prosessorien suunnittelua.

MOVE-ympäristö on puoliautomatisoitu ohjelmisto ASIP-prosessoreiden suunnittelemista varten. MOVE-ympäristö koostuu kolmesta osasta. Suunnitteluavaruuden kartoitus-työkalu etsii annetulle sovellukselle optimaalisen prosessoriarkkitehtuurin. Ohjelmistotyökaluilla käännetään korkean tason ohjelmointikielellä toteutettuja sovelluksia suoritettaviksi binääriohjelmiksi käyttäen käskytason rinnakkaisuutta hyväksi. Laitteistotyökaluilla generoidaan puolestaan prosessorin kuvaus varsinaista piitoteutusta varten. MOVE-ympäristössä suunnitellut prosessorit käyttävät siirtoliipaisuperiaatetta (transport triggering), jossa ohjelma määrittelee ainoastaan datan siirrot laskentayksiköiden ja rekisterien välillä. Varsinaiset operaatiot tapahtuvat näiden datan siirtojen liipaisemina, eli päinvastaisesti kuin perinteisissä ohjelmointimalleissa. Tällä periaatteella toimivista prosessoriarkkitehtuureista käytetään nimitystä siirtoliipaisuarkkitehtuuri (Transport Triggered Architecture, TTA).

Datan siirrot suoritetaan TTA-prosessorissa ns. kytkentäverkon kautta, joka koostuu väylistä ja laskentayksiköiden sekä väylien väliin kytketyistä ns. soketeista. Sokettien tehtävä on siis siirtää dataa laskentayksiköstä/rekisteristä väylälle tai väylältä lasken-

tayksikköön/rekisteriin. TTA-prosessorin kytkentäverkon kytkentöjen määrä voidaan optimoida tiettyä sovellusta varten, mikä vähentää prosessorin valmistuskustannuksia huomattavasti.

Nopea Fourier-muunnos (fast Fourier transform, FFT) on eräs keskeinen digitaalisessa signaalinkäsittelyssä käytettävä muunnos. Se perustuu diskreettiin Fourier-muunnokseen, eli FFT-muunnoksella tarkoitetaan diskreetin Fourier-muunnoksen laskentaan kehitettyjä nopeita algoritmeja. FFT-muunnosta tarvitaan mm. tietoliikennesovelluksissa sekä digitaalisessa puheen- ja kuvankäsittelyssä. FFT-muunnos on erillisistä laskentatasoista koostuva kompleksinen muunnos. Nämä laskentatasot muodostuvat pienikokoisista diskreeteistä Fourier-muunnoksista, joiden laskenta voidaan tehdä suorittamalla yksinkertaisia kompleksisia kertolaskuja sekä summauksia. Tällaista pienikokoista diskreettiä Fourier-muunnosta, joka toimii FFT-muunnoksen perusrakenne-elementtinä, kutsutaan perhoseksi. FFT-muunnoksen laskentaan liittyy keskeisesti myös ns. indeksin generointi, jonka avulla määritetään perhosten operandien muistiosoitteet muunnosta suoritettaessa.

Tässä diplomityössä toteutettiin FFT-muunnos TTA-prosessoria käyttäen, jotta voitiin arvioida TTA-arkkitehtuurin suorituskyky nopean Fourier-muunnoksen laskennassa. FFT-muunnos toteutettiin aluksi käyttäen korkean tason ohjelmointikieltä ja kääntäjää. Korkean tason ohjelmointikielellä tehtiin yhteensä kolme toteutusta, joista kukin toteutti 1024-pisteisen muunnoksen. Aluksi sovellus kirjoitettiin käyttäen ainoastaan korkean tason kielen perusoperaatioita ja -tietotyyppejä. Tämän jälkeen sekä sovelluksen koodiin että prosessorin arkkitehtuuriin alettiin tehdä inkrementaalisia parannuksia paremman suorituskyvyn aikaansaamiseksi. Toisessa toteutuksessa arkkitehtuuriin lisättiin kompleksinen kertoja ja summain suorittamaan kompleksilukuaritmetiikkaa. Kolmannessa toteutuksessa arkkitehtuuriin lisättiin vielä osoitegeneraattori-yksikkö nopeuttamaan perhosten operandien laskennassa tarvittavaa muistiosoitearitmetiikkaa. Lisäksi ylimääräiset kertolaskut poistettiin koodista ja koodin sisemmän silmukan rakennetta optimoitiin hieman. Kukin tehty muutos paransi FFT-muunnoksen laskennan suorituskykyä. Varsinkin osoitegeneraattorin käyttö paransi suorituskykyä huomattavasti.

Viimeisen korkean tason kielellä tehdyn toteutuksen suorituskyky ei kuitenkaan ollut riittävän hyvä. Tämän takia 1024-pisteinen FFT-muunnos toteutettiin seuraavassa toteutusvaiheessa assembly-koodia käyttäen. Aluksi määritettiin, miten yksittäinen perhonen vuoronnetaan tehokkaasti. Sitten voitiin helposti määrittää, miten muunnoksessa tarvittava useiden peräkkäisten perhosten laskennan vuoronnus piti tehdä. Muunnoksen operaatioiden vuoronnuksen jälkeen määritettiin optimaalinen arkkitehtuuri FFT-muunnoksen laskentaa varten sekä kirjoitettiin assembly-kielinen kuvaus 1024-pisteiselle muunnokselle. Prosessorin kytkentäverkon kytkentöjen määrä kyettiin optimoimaan erittäin alhaiselle tasolle assembly-kielisen toteutuksen avulla, mikä vähensi prosessorin valmistuskustannuksia huomattavasti. Lopuksi generoitiin prosessorin laitteistonkuvauskielinen toteutus MOVE-ympäristön laitteistotyökaluja käyttäen, minkä jälkeen prosessori simuloitiin ja syntesoitiin. Tähän optimoituun assembler-toteutukseen pohjautuen työssä esitellään FFT-muunnoksen laskentaan optimoitu tehokas TTA-prosessori, jota verrataan myös muihin kirjallisuudessa raportoituihin FFT-prosessoreihin.

TTA-arkkitehtuurin suorituskyvyn huomattiin olevan erittäin hyvä FFT-muunnoksen laskennassa, kun ohjelmoinnissa käytettiin assembly-koodia. TTA-arkkitehtuurilla päästiin jopa parempiin suorituskykyindekseihin kuin hyvillä, FFT-muunnoksen laskentaan kehitetyillä sovelluskohtaisilla integroiduilla piireillä. Korkean tason kielellä tehtyjen toteutusten suorituskyvyt olivat huomattavasti huonompia kuin assembler-toteutuksen suorituskyky, joka oli odotettu tulos. Yhteenvetona voidaankin todeta, että tässä diplomityössä saavutettujen tulosten perusteella TTA on lupaava ohjelmoitava arkkitehtuurikandidaatti digitaalisen signaalinkäsittelyn sovellusten toteuttamiseen.

LIST OF ABBREVIATIONS AND SYMBOLS

α	Constant reflecting the importance of area
β	Constant reflecting the importance of execution time
ADF	Architecture Description File
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
BEM	Binary Encoding Map
CMOS	Complementary Metal Oxide Semiconductor
DFT	Discrete Fourier Transform
DIT	Decimation-In-Time
DSP	Digital Signal Processor or Digital Signal Processing
DP-DMEM	Dual-Port Data Memory
FIFO	First In First Out
FFT	Fast Fourier Transform
FFTTA	FFT-on-TTA
FU	Function Unit
GCC	GNU Compiler Collection
GNU	Gnu's Not Unix
GPP	General Purpose Processor
GPR	General Purpose Register

HDL	Hardware Description Language
HLL	High-Level Language
HW	Hardware
IC	Interconnection Network
IDU	Instruction Decode Unit
IFU	Instruction Fetch Unit
ILP	Instruction-Level Parallelism
IMEM	Instruction Memory
LSU	Load-Store Unit
OTA	Operation Triggered Architecture
PC	Program Counter
RF	Register File
RTL	Register-Transfer-Level
SFU	Special Function Unit
SP-DMEM	Single-Port Data Memory
SW	Software
TTA	Transport Triggered Architecture
VHDL	Very high speed integrated circuit Hardware Description Language
VLIW	Very Long Instruction Word

1. INTRODUCTION

The current trend in digital signal processing (DSP) application domain is to move towards high-level language (HLL) programming and customizable architectures. The reason behind this development is the increasing complexity of DSP applications. Designers are not able to produce the products on the market in time without effective design tools and design concepts. High performance requirements are also set for both the applications and the architectures. In addition, implementation costs of a DSP-system should be as minimal as possible to manufacture even millions of products economically viable. Usually, companies are forced to make compromises between high performance and low implementation costs to produce productive products. Low-level optimisation techniques are still used for obtaining the cost-performance requirements set to the products.

Application-specific instruction set processors (ASIP) can be designed to respond to cost-performance requirements of DSP applications. ASIP is programmable and it provides a customizable architecture for the requirements of the application. Unlike in general purpose processors (GPP), custom hardware can be easily tailored for the requirements of the application and added to the modular architecture for obtaining better performance. The design process of an ASIP is demanding and, therefore, also time consuming for designers since the ASIP has to be reprogrammed when the architecture is modified. Furthermore, the design space is often very large meaning that there can exist hundreds of good architecture configurations for a certain application. To ease the design process of ASIPs, a tool set, MOVE Framework, was developed [1]. The MOVE framework makes it possible to design ASIPs semi-automatically. Processors designed with the MOVE framework exploit the transport triggered architecture (TTA) concept proposed by Corporaal [2]. The MOVE framework is composed of three main components: design space explorer, hardware subsystem, and software subsystem. The design space explorer searches the design space and tries to find a good architecture configuration for the given application. The hardware subsystem is responsible for generating the layout of the processor. The purpose of the software subsystem is to compile HLL applications to binary executables by exploiting instruction-level paral-

lelism (ILP). Furthermore, the software subsystem contains an instruction set simulator and also an assembler which has been ported into it to be able to program assembly code for TTA processors.

Fast Fourier transform (FFT) is a fundamental operation in the DSP application domain. In this thesis, FFT is implemented on TTA to evaluate the performance of the TTA in computing FFT. First, FFT is implemented using HLL code. This is done to see how effectively the FFT application can be performed on TTA by using the HLL compiler. In the next implementing phase, FFT is implemented by using the assembler. The assembler implementation is done to see how effectively the FFT application can be performed on TTA in general. The performance of the HLL compiler can be also considered with the aid of the assembler implementation. Based on the assembler implementation, an effective TTA processor is proposed in the end of this thesis for computing the 1024-point radix-4 FFT. The proposed processor is also compared to several other commercial and academic FFT processors.

The structure of this thesis is as follows. The TTA concept is described in Chapter 2 by introducing three topics: development from VLIW architecture to TTA, hardware aspects, and software aspects of TTA. In addition, the MOVE framework is described with its subcomponents. Finally, the TTA assembler and the TTA assembly language are introduced. The needed FFT theory is represented in Chapter 3. First, the definition of the discrete Fourier transform (DFT), on which the FFT is based on, is given together with a couple of other usefull definitions. Next, FFT algorithms are discussed by getting of the ground from the principle of Cooley and Tukey. Based on the principle of Cooley and Tukey, radix-2 and radix-4 FFT algorithms are discussed. The definition of the algorithm realized on TTA is also given. Finally, the index generation principle of this algorithm is described. HLL implementations are discussed in Chapter 4. Three separate implementations, in which incremental improvements are done to both the code and the architecture, are described. Finally, the results of the explorations are shown. The optimized assembler implementation is described in Chapter 5 in detail. The used scheduling principle is discussed first. After that, the assembly programming phase is described together with the assembly code. Finally, the hardware implementation of the processor is discussed. The effective TTA processor for computing the 1024-point radix-4 FFT is proposed in Chapter 6. The characteristics of the proposed processor are listed first after which the processor is compared with several other FFT processors to evaluate TTA's performance against other FFT architectures. This thesis is summarized in Chapter 7 by representing the conclusions.

2. MOVE FRAMEWORK

The current trend in designing embedded systems, especially for DSP domain, is to move towards customizable processor architectures. This results from the fact the design process of an embedded system exploiting a customizable processor architecture is faster and easier for designers than that of the design process exploiting application-specific integrated circuits (ASICs). Due to scalability, flexibility and modularity of customizable processor architectures, they can be easily scaled for different applications, unlike ASICs, and the design process of them can be automated. In this chapter, a customizable processor architecture and an automated design environment for this architecture are descriped. TTA is such an architecture and the MOVE framework such a design environment that automates the design process of TTA processors.

TTA is described in Section 2.1. In Section 2.2, the MOVE framework is described together with its main components. The TTA assembler for the assembly level TTA coding is introduced in Section 2.3.

2.1 Transport Triggered Architectures

TTA is derived from the very long instruction word (VLIW) architecture, which is one of the processor architectures exploiting instruction-level parallelism (ILP). The development from VLIW to TTA is introduced in Section 2.1.1. The hardware (HW) and software (SW) aspects of TTAs are discussed briefly in Sections 2.1.2 and 2.1.3.

2.1.1 From VLIW to TTA

The VLIW architecture is flexible and scalable, which makes it an interesting choise for the designs of ASIPs. VLIWs are constructed from multiple, concurrently operating function units (FUs) where each FU supports RISC style operations. The ILP can also be seen from the instructions: each VLIW instruction specifies multiple RISC operations. An example VLIW processor is illustrated in Fig. 1. [3]

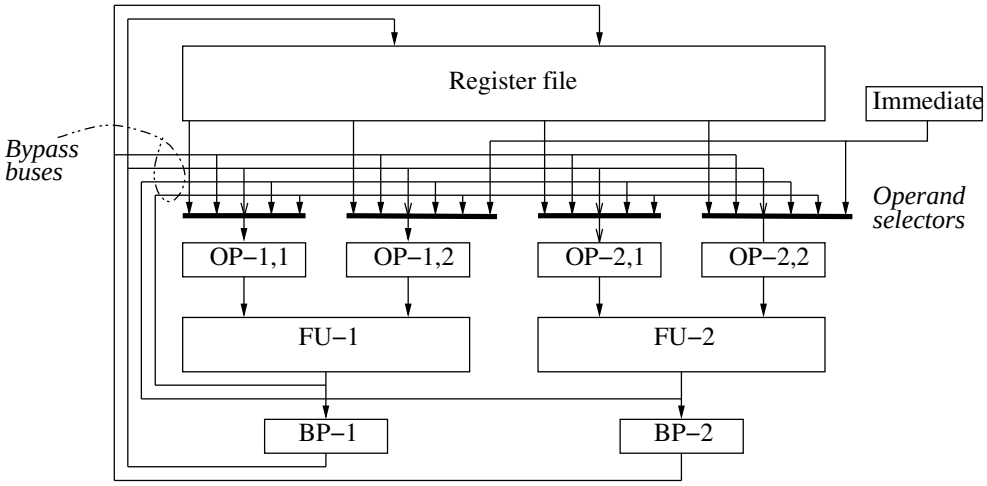


Figure 1. *The organization of a VLIW processor with two FUs.*

The VLIW processor contains a multi-ported register file (RF), that is shared by the FUs, and a bypassing circuit for forwarding the results from the outputs of FUs to the inputs of FUs. This bypassing circuit bypasses the register file and its purpose is to maximize the utilization of FUs. As ILP is increased in VLIWs by adding more FUs, the register file and the bypassing circuit must always be implemented for worst case situations even though these kind of situations rarely emerge. That is, the implementation of RF for worst case situation in two-operand instructions would mean that each FU must have three ports for accessing the RF to perform two reads and one write simultaneously [4]. This increases the complexity of the RF. Also the complexity of the bypassing circuit is increasing rapidly as the number of FUs is increased. This rapidly increasing connectivity of VLIWs datapath makes it complex and restricts the scalability of VLIW architecture [4].

Due to the limitations of VLIW architecture, TTA was proposed by Corporaal [3]. The fundamental idea was to reduce the RF and bypass complexity. In TTAs, the complexity of the RF is reduced by reducing the number of RF ports, and the number of registers in the RF itself [3]. The complexity of the bypass network is reduced by bringing the RFs into the same architectural level as FUs, which is illustrated in Fig. 2. Furthermore, the traditional programming model, in which a program specifies operations to be executed and then a single operation triggers data transports, is mirrored in TTAs. I.e., in TTAs, a program specifies data transports to be executed and then a single data transport triggers operations [3].

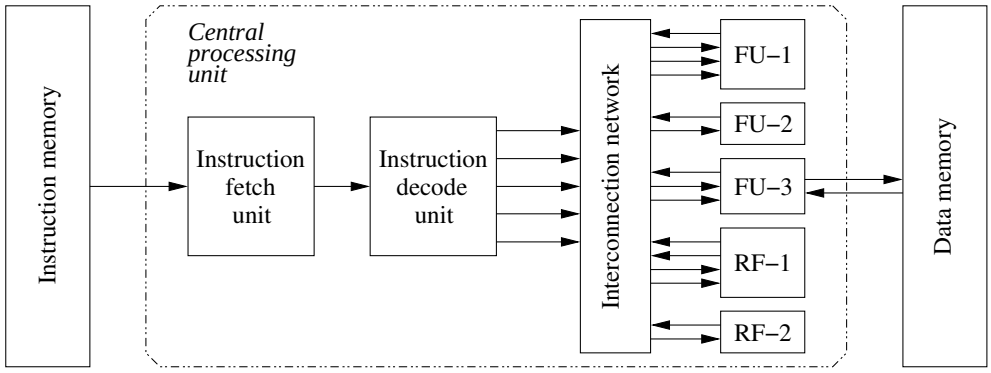


Figure 2. *Organization of the TTA.*

2.1.2 Hardware Aspects

A TTA processor consists of FUs, RFs, interconnection network (IC), control logic, and separate instruction and data memories. The structure of a TTA processor is depicted in Fig. 3.

The IC is responsible for transferring data between the FUs and the RFs. It is composed of data transport buses and sockets, which are connecting the FUs and the RFs to each others. There exists two types of sockets: input and output sockets. An input socket is responsible for feeding the data from a bus into an FU and an output socket places a result of an FU on a correct bus. Input sockets can be implemented using multiplexers, and output sockets, respectively, by using de-multiplexers. The connectivity of the IC, i.e., the connections between buses and sockets, can be optimized for the application. This makes the structure of the sockets simpler, i.e., they can be implemented by using simpler multiplexers and de-multiplexers which reduces the required chip area. Also the capacitive bus load is reduced indicating that faster data transport times can be obtained [3].

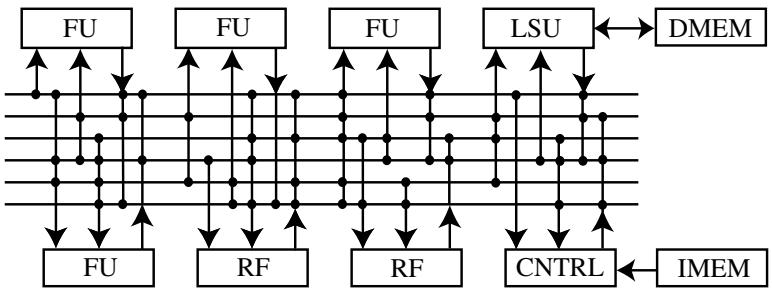


Figure 3. *TTA processor structure. FU: function unit. RF: register file. LSU: load-store unit. CNTRL: control unit. DMEM: data memory. IMEM: instruction memory. Dots represent connections between buses and sockets.*

FUs are performing the actual computation. They are completely independent of each other and of the interconnection network. FUs can, therefore, be designed separately, and pipelined independently. They contain three types of registers: operand, trigger, and result. Input data values of an FU are moved to its operand and trigger registers. An FU can have more than one operand register. As data is moved to the trigger register by a trigger move, the FU starts to perform an operation from its operation set. The operation to be performed is selected by an opcode, that is decoded in the trigger socket and transferred through it into the FU together with the trigger data. This is also known as *triggering* of the FU.

RFs provide general-purpose registers (GPRs) for storing temporary values. They can be partitioned and each partition can be accessed through its own input and output sockets. The number of input and output sockets of a RF partition can be adjusted according to the program's needs. Each RF partition can also have a different number of GPRs.

The control logic is implemented inside the control unit. The control unit is composed of four operational units: an instruction fetch unit (IFU), an instruction decode unit (IDU), an immediate unit, and a guard unit [5]. IFU fetches the instructions from the instruction memory and feeds them to IDU, which decodes them and activates the correct sockets to perform the data transports on the buses as requested by the instruction. Due to the fact that one bus can perform one data transport in one clock cycle, one TTA instruction specifies always as many data transports as there are data transport buses in a target TTA processor. The instruction format of the TTA is depicted in Fig. 4. A field in the instruction specifying a single data transport is called a *move slot*. A move slot contains three fields: guard ID, destination ID, and source ID. The destination ID specifies which input socket reads the data from a bus. The source ID specifies an output socket that writes the data on a bus. The guard ID specifies whether a data transport is performed on a bus or not. It can be used to implement conditional statements. The reserved immediate field is used for specifying long immediates.

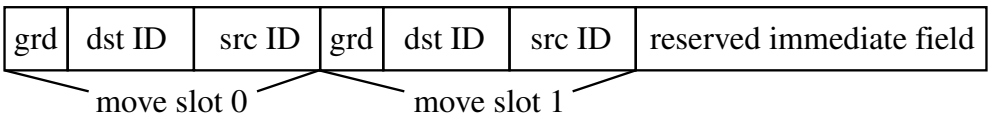


Figure 4. *TTA instruction format.*

2.1.3 Software Aspects

As already mentioned in Section 2.1.1, the traditional programming paradigm of operation triggered architectures (OTAs), in which a program specifies operations which trigger data transports, is mirrored in TTAs. That is, a TTA program specifies the data transports to be performed by the interconnection network. Therefore, only one type of operation is supported: the move operation, which performs a data transport from a source location to a destination location. The source location can be a register or an output of an FU and the destination location a register or an input of an FU.

Programming an operation on a TTA processor consists of moving operands to the input registers of an FU that is capable to perform the operation, and moving the result from the output of the FU to another FU or an RF after the FU has performed the operation [3]. The actual operation, e.g., addition, occurs as a *side effect* of the trigger move, which moves the data to the trigger register of the FU.

Typically, one OTA operation corresponds to three move operations: operand, trigger, and result moves. For example, a simple OTA addition:

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

which adds together values from register locations r1 and r2 and stores the result to the register location r3, can be converted to the following three move operations in the programming model of TTA:

```
r1 -> add_o
r2 -> add_t
add_r -> r3
```

The first move indicates the operand move to the operand register of an add-unit. 'add_o' indicates the input (operand) socket through which the operand move is performed. The second move indicates the trigger move and 'add_t' the input (trigger) socket, respectively. The third move indicates the result move to the register location r3. 'add_r' is the output (result) socket through which the result move is performed.

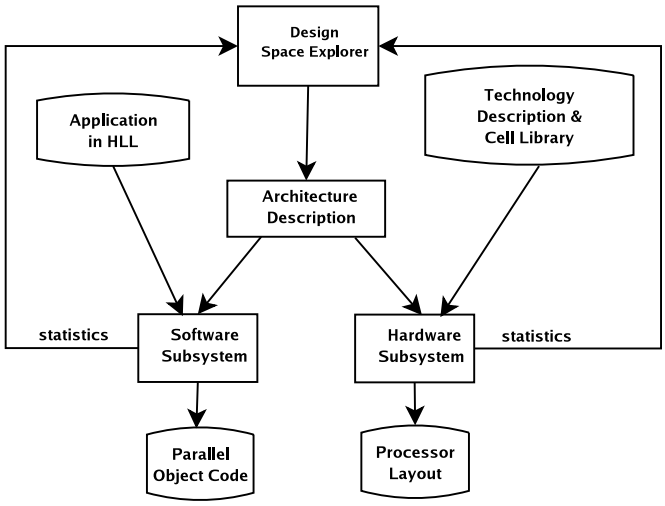


Figure 5. *MOVE framework.*

2.2 MOVE Tools

The MOVE Framework is a set of software tools for semi-automated design process of application-specific instruction set processors (ASIPs) [1]. The semi-automation, provided by these tools, is essential in ASIP designs as the design space is very large. The design time of ASIPs can thereby be shortened by exploiting these tools. Processors that are designed with the MOVE Framework, are based on TTA that was explained briefly in the previous Section 2.1. Fig. 5 depicts the general structure of the MOVE Framework that is composed of three main components: design space explorer, software subsystem and hardware subsystem.

The purpose of the design space explorer is to iteratively search for interesting processor configurations that satisfy design constraints set by a designer, for a given set of applications. The main design constraints of a processor design are in general the performance and the cost requirements the processor design has to meet. The main task of the software subsystem is to produce parallel object code for the target processor from the HLL code. The hardware subsystem is responsible for producing a hardware description language (HDL) description of the target processor. Furthermore, both the software and the hardware subsystem provide statistics for the design space explorer.

The architecture description file (ADF) is an essential file that fully characterizes the structure of a single target processor. Firstly, it specifies FUs, sockets, buses and RFs of the target processor. Secondly, it specifies connections between separate hardware blocks of the processor. That is, connections between FUs, sockets and buses are defined. Furthermore, the ADF contains information about immediates the target pro-

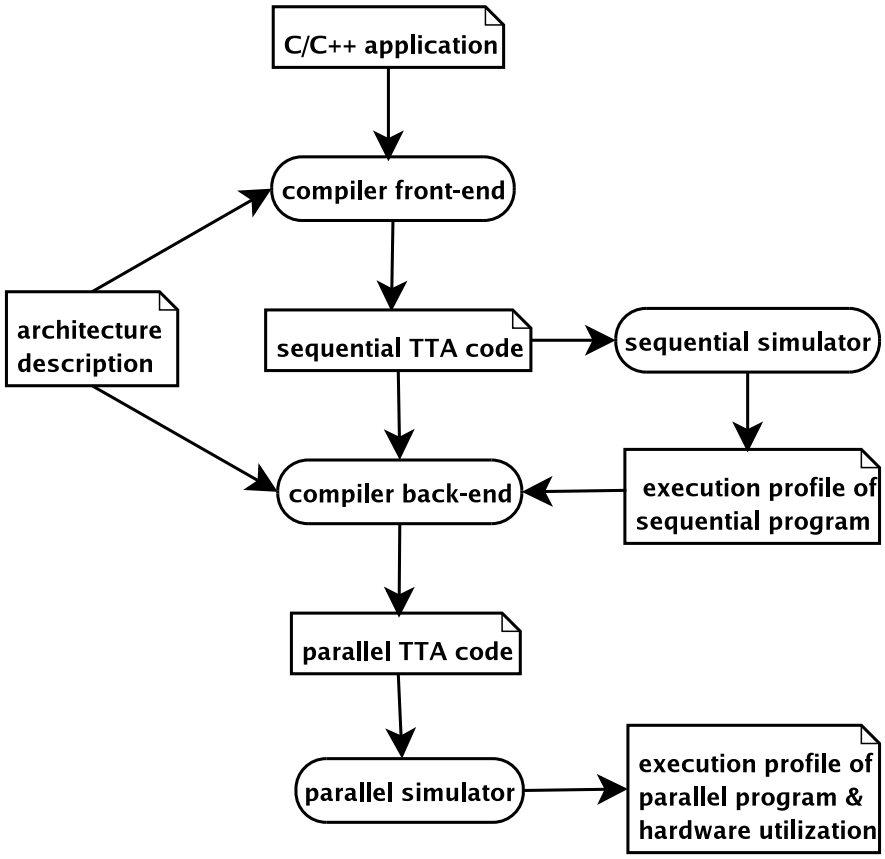


Figure 6. *Software subsystem.*

cessor is capable to handle.

The software subsystem is described in Section 2.2.1 and the hardware subsystem is discussed briefly in Section 2.2.2. The design space explorer is described in Section 2.2.3.

2.2.1 Software Subsystem

The software subsystem of the MOVE Framework is responsible for generating binary code for a given TTA processor design that comprehends both an architecture description of a target processor and an application the target processor is going to execute. The main component of the software subsystem is the compiler that is composed of two subcomponents: the front-end and the back-end. The software subsystem contains also two simulators: the sequential and the parallel simulator. Furthermore, tools for profiling, trace analysis and both code and control flow viewing of an application are provided by the software subsystem. The software subsystem is depicted in figure 6.

The purpose of the compiler front-end, i.e., the sequential code generator, is to trans-

form user applications written in a HLL into sequential TTA code. The compiler front-end is based on GNU gcc. It consists of the compiler, assembler, linker and binary utilities. All of these development tools have been ported to the sequential TTA target architecture which is capable to execute sequential TTA code. The sequential TTA target architecture contains only one bus and such a set of FUs that operations of an application can be executed. The sequential machine performs the data transports of the sequential code one at a time by utilizing the single bus of the sequential machine. [1]

The sequential TTA code can further be fed to the sequential simulator to generate execution profile of the sequential TTA program. This execution profile contains among others information about elapsed clock cycles, code size, immediates, most utilized operations and data transfers.

The compiler back-end, i.e., the scheduler, exploits the execution profile of the sequential program during the scheduling process in which the sequential TTA code is mapped onto the available hardware resources of the target TTA processor. As output, the scheduler produces the parallel TTA code in both binary and textual assembly formats. The textual assembly format, from which one can check the quality/effectiveness of the parallel TTA code, is described briefly in Section 2.3.1. The more detailed usage of the scheduler and brief descriptions of the algorithms, according to which the scheduling process is performed, are described in [1].

Finally, the parallel TTA code produced by the scheduler can be simulated in the parallel simulator, which produces the execution profile of the parallel TTA program and the utilisations of different hardware resources of the target TTA processor as its output. The correct functionality of the scheduled, parallel TTA code can be verified by comparing the result data of the parallel simulation to that of the sequential simulation.

2.2.2 Hardware Subsystem

The hardware subsystem of the MOVE Framework is responsible for generating and evaluating the hardware of a processor design. There are two components in the hardware subsystem: the hardware cost estimator and the processor generator. The purpose of the hardware cost estimator is to evaluate the target TTA processor in terms of chip area, power and timing. The current hardware cost estimator of the MOVE Framework and the cost estimation methods are described in [6].

The processor generator is responsible for generating a synthesizable very high speed

integrated circuit hardware description language (VHDL) [7] code from the architecture description. The current processor generator, explained more thoroughly in [8] and [9], exploits a separate library, that contains VHDL-descriptions of the basic building blocks of TTA processor, such as FUs, sockets, RFs and buses. If a processor design exploits user-defined FUs, SFUs, the VHDL-descriptions of the SFUs must be implemented and compiled to the library by the user.

2.2.3 Design Space Explorer

The design space explorer of the MOVE Framework is a software tool whose purpose is to automatically find target processor configurations with favourable cost/performance ratio for a given set of applications. It explores the design space by evaluating the cost and the performance of a large set of processor configurations. At first the explorer evaluates an initial target processor given as a parameter. Then, the explorer starts to evaluate different processor configurations that are obtained by removing components from the initial target processor configuration. That is, it is the responsibility of the designer to determine a suitable initial target processor configuration that contains enough resources for the needs of applications to be explored.

The design space exploration process is composed of two independent phases: *resource optimization* and *connectivity optimization*. The resource optimization phase adds and removes FUs, sockets, register files, register file ports and move buses according to a certain search algorithm. There are three search algorithms available in the current MOVE framework: local search, bactracking search and simulated annealing [1]. The connectivity optimization phase removes connections between the sockets and the move buses. The purpose of the connectivity optimization is to make the target processor both faster and cheaper because the reduction of connectivity brings down both the clock period and the chip area. In the both exploration phases, the explorer acts as a driver program by determining the performance and the cost of a configuration by invoking the scheduler and the estimator. In the following two subsections, both of the independent exploration phases are explained in more detail.

Resource Exploration

After the evaluation of the initial machine configuration, the explorer begins to remove components from the initial machine configuration until a minimum configuration that is needed to perform the application is reached. A component that will be removed

next is always determined by the following quality function

$$quality(config) = \left(\left(\frac{t_0}{t} \right)^{\alpha} \cdot \left(\frac{A_0}{A} \right)^{\beta} \right)^{\frac{1}{\alpha+\beta}} \quad (1)$$

where t_0 and A_0 are, respectively, the execution time and the area of the initial processor configuration, while t and A are, respectively, the execution time and the area of the processor configuration currently being evaluated. α and β are constants reflecting, respectively, the importance of cost and performance.

After the minimal configuration for performing the application is reached, the process is reversed and components are again put pack in a different order than they were removed until the initial configuration is again reached. Also the component to put back in a machine configuration is determined by the quality function. These reduce/extend phases of the resource exploration, executed 5 times, are made with different values of α and β parameters.

After these reduce/extend phases the explorer determines which of the evaluated configurations are Pareto configurations, i.e., such realizable configurations that there exists no other configurations that are both faster and cheaper. The Pareto configurations can then be plotted into the illustrative cost-execution time design space. From these Pareto configurations a designer then chooses the most interesting ones according to his own criteria for a more detailed evaluation. Fastness, cheapness or the best compromise between fastness and cheapness can be considered as a basic processor configuration selection criteria.

Connectivity Exploration

In the connectivity exploration, the explorer reduces connections between the buses and the sockets of the interconnection network of a TTA processor. Through this reduction of connectivity, the capasitive load of a bus is reduced. This, in turn, may shorten the critical path of the TTA processor and, therefore, shorten the duration of the minimum clock cycle at which the TTA processor can run. In addition, reducing connections results in smaller chip area due to simpler multiplexers and de-multiplexers in the sockets. Furthermore, the instruction size may decrease since the number of addressable locations per bus is lower.

2.3 TTA Assembler

Occasionally, there may occur situations in which the software subsystem is incapable to produce parallel code that is effective enough. In these situations, the TTA assembler can be utilized for code optimizations. However, one should not move to the quite tedious assembly level TTA coding if it can be avoided somehow. That is, one should first try to modify the HLL-code in such a way that the scheduler manages to generate the desired parallel code that is effective enough. Quite often, simple rearrangements in the original HLL-code can have tremendous effects in the parallel output code of the scheduler. If the modifications to the HLL-code take no effect and for some reason the scheduler fails to produce the desired parallel code, the usage of assembler for optimization purposes is justified.

Section 2.3.1 introduces the syntax and the structure of TTA assembly language the TTA assembler reads in. The usage of the tool and the simulation of binaries produced by this tool are discussed in Section 2.3.2.

2.3.1 TTA Assembly Language

In fact, there does not exist any formal specification of the TTA assembly language. However, it is quite easy to figure out how it works and matches with the architecture definition and binary encoding map (BEM) files by simply examining a piece of assembly code the scheduler has produced into a textual file.

Firstly, the front-end compiler of the software subsystem divides instructions of procedures of a HLL-application into separate blocks, known as basic blocks, that contain sequential TTA assembly code [1]. In the next compilation stage, the back-end compiler schedules the sequential TTA code onto the resources of the target processor and produces the parallel TTA assembly code of the application as an output. This parallel TTA assembly code matches with binary code the target processor reads from its instruction memory and it consists of the basic blocks that contain parallel TTA assembly instructions. These parallel TTA assembly instructions consist of data moves that are performed in the interconnection network of the target processor. One instruction contains always as much data moves as there are data transport buses in the target processor. The following example contains two TTA assembly code instructions of a basic block for a target processor that has a total of three data transport buses:

```
1 -> fu4.add_o [m1/-/-/fu4_o], r11 -> fu4.add_t [m2/-/ri_o4/fu4_t], ...;
fu4.add_r -> r11 [m1/-/fu4_r/ri_i4], fu4.add_r -> fu10.gtu_o [m2/-/fu4_r/fu10_o], 4 -> fu10.gtu_t [m3/-/-/fu10_t];
```

In the previous example, instructions are separated with semicolons (';') and the data moves inside one single instruction with colons (','). The right arrow ('→') implies a data move on a transport bus of the target processor. On the left side of the right arrow is a source location from which the data is transported to a destination location, which is always on the right side of the right arrow. Three dots ('...') imply an empty data transport in which no data is transported. Furthermore, after each data move, there is always some extra information enclosed by brackets ('[-/-/-/-]'). This additional information consists of four fields that are separated with slash characters ('/'). The name of the bus on which the data is transported is defined in the first field. The second field is reserved for long immediates. It indicates whether the register for long immediates is read or not during a data transport. If this field contains a hyphen ('-'), the register for long immediates is not read. The name of the output socket via which the data is transported from a source location is defined in the third field and, respectively, the name of the input socket via which the data is transported to a destination location is defined in the fourth field. If the third or fourth field contains a hyphen ('-'), it indicates, respectively, that the data transport does not utilize output or input sockets. This additional bus/immediate/socket information enclosed by brackets must always be provided when assembly code is being written for the TTA assembler.

2.3.2 Usage and Simulation

The TTA assembler is a software tool that reads in three input files: the parallel TTA assembly code of an application, the ADF of the target processor for the application, and the binary encoding map (BEM) file of the architecture description. The BEM file, that is produced by the software subsystem, describes how to generate instruction bit vectors for the given target TTA processor. As output, the TTA assembler produces the corresponding binaries of the application in the ascii-text format into a separate file. Furthermore, the TTA assembler provides a usefull script that reads the BEM file and produces a new textual file which shows how sockets are connected to data transport buses and through which sockets of a register file partition one can find a certain register rx where x is the index of the register. Especially, this register information is very usefull when TTA assembly code is being written.

Due to the fact that the current MOVE Framework can not read in and simulate binaries generated by the binary utilities of the software subsystem or the TTA assembler, the only way to verify the correct functionality of the hand-coded TTA assembly code is to simulate the binaries in a HW-simulator, such as `ModelSim SE PLUS 5.8a`

[10]. For the simulation of binaries in the HW-simulator, a HDL description of the target TTA processor for which the binaries are generated is needed. This is the main task of the Hardware Subsystem. A semi-automated design tool, that is the processor generator `MOVEgen`, translates the internal architecture description format of the MOVE Framework into generic HDL, VHDL. The detailed usage of this software tool is described in [9].

3. FAST FOURIER TRANSFORM

Transforms are mainly used for reducing the complexity of mathematical problems. By applying appropriate transforms, differential and integral equations may be converted into algebraic equations, which provide means to obtain the solutions more easily. A well-known example of such a transform is *Fourier transform* decomposing a signal into its frequency components. [11].

The Fourier transform is defined for countinuous-time signals but the DSP system manipulates discrete-time signals. Therefore, an alternative representation of the Fourier transform is used in DSP. The discretized approximation of the Fourier transform is the DFT, which is one of the most fundamental operations in DSP. The DFT is defined for discrete-time, finite-duration signals and is a uniform sampling of the Fourier transform of a signal, with number of samples equal to the length of the signal.

The direct computation of DFT is not reasonable due to the fact that the DFT contains redundant computations. Several algorithms have been developed by avoiding the redundancy and lowering the arithmetic complexity. These fast algorithms, which can speed up the DFT computation, are called *fast Fourier transform* (FFT).

In this chapter, a couple of FFT algorithms and a principle for the index generation of FFT are described. First, the exact definitions of Fourier transform and DFT are given in Section 3.1. The algorithms are described in Section 3.2. Finally, the index generation principle of the radix-4 FFT algorithm realized on TTA is explained in detail in Section 3.3.

3.1 Definitions

The DFT is a discretized approximation of the continuous Fourier transform defined as [11]

$$X(\omega) = F[x(t)] = \sqrt{\frac{1}{2\pi}} \int_{-\infty}^{\infty} x(t) e^{(-j\omega t)} dt \tag{2}$$

where $j = \sqrt{-1}$. The standard definition of the DFT over an N-point complex data sequence x_n is [11]

$$X_m = \sum_{n=0}^{N-1} x_n e^{-j2nm\pi/N} \tag{3}$$

In literature, it is common to use the notation

$$W_N = e^{-j2\pi/N} \tag{4}$$

for the N-th root of unity. Its powers are often referred to as *twiddle factors*. With the aid of this notation, the DFT transform matrix F_N of order N can be written as [11]

$$[F_N]_{nm} = W_N^{nm}, \quad n,m = 0,1,\ldots,N-1 \tag{5}$$

where n and m refer to the indices of the elements of the matrix.

3.2 FFT Algorithms

Most of the FFT algorithms are based on the fundamental principle of decomposing the computation of FFT of a sequence into successively smaller FFTs where the coefficients turn out to be trivial complex numbers, such as, 1, -1, j, or $-j$. The manner, with which this principle is implemented, leads to a variety of different algorithms.

The structure of this section is the following. First, the *algorithms of Cooley and Tukey* are discussed on abstract level in Subsection 3.2.1. Radix-2 algorithms are described next in Subsection 3.2.2. Radix-4 algorithms are discussed in general and the exact definition of the radix-4 algorithm realized on TTA is given in Subsection 3.2.3.

3.2.1 Cooley and Tukey

The first and most common FFT algorithms were presented by Cooley and Tukey (1965), after whom they are often named. Their FFT algorithms exploit the inherent computational redundancy in the DFT. In fact, they invented a method of which several FFT algorithms can be derived. This method, known as the *Cooley-Tukey decomposition*, re-expresses the DFT of an arbitrary composite size $N = PQ$ in terms of smaller DFTs of sizes P and Q, recursively. The initial computational complexity of the DFT is of order N^2. It can be reduced down to order of $N \log N$ by exploiting the Cooley-Tukey decomposition.

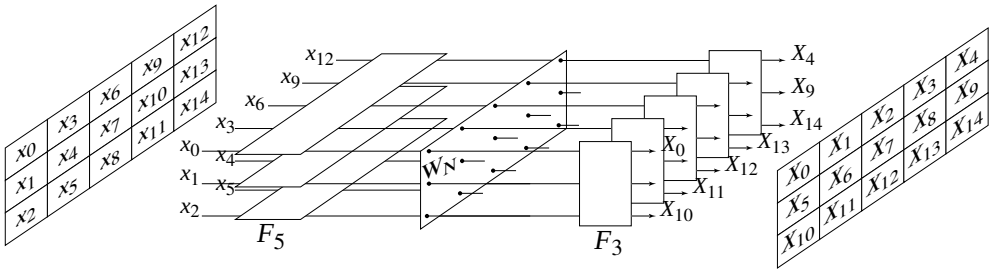


Figure 7. *Principle of Cooley-Tukey DFT decomposition,* $N = 15, P = 3, Q = 5$

The principle of Cooley-Tukey decomposition, illustrated in Fig. 7, for the composite sequence size of $N = PQ$ is as follows:

1. Decimate the sequence into P sequences of size Q.
2. Compute P DFTs of size Q.
3. Multiply the resulting sequences by twiddle factors.
4. Re-order the sequences.
5. Compute Q DFTs of size P.

The previous decomposition process can again be applied to the resulting shorter DFTs of size P or Q, if either P or Q is not a prime number. Such a recursion can be continued until the sizes of all the resulting DFTs are prime. In such a case, a *mixed radix* algorithm is obtained. If the original length N is a power of an integer number R, $N = R^r$, the resulting algorithm is called *radix-R* FFT. In this case, the sizes of the DFTs in the final computational structure are the same. Furthermore, when R is prime, the algorithm is called *prime radix* FFT. When P is chosen as the smallest prime factor of the size of the transform N, the time domain sequence is decimated, and the resulting algorithm is called *decimation-in-time* (DIT) FFT. By reversing the role of P and Q, *decimation-in-frequency* (DIF) algorithm is obtained. [11]

3.2.2 Radix-2

The simplest and the most common Cooley-Tukey algorithm is the radix-2 DIT FFT, in which the size of the transform has to be a power of two. This is not, however, a big limitation in practise since the size of the transform can usually be chosen freely by the application. By choosing $P = 2$, $Q = N/2$, the DFT of size N is decomposed into two DFTs of size $N/2$ followed by multiplications with twiddle factors and then $(N/2)$

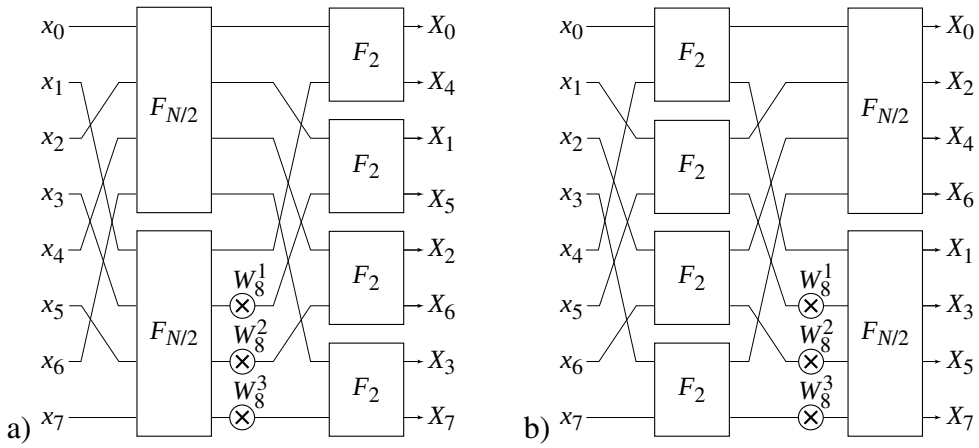


Figure 8. *Principle of radix-2 FFT algorithms: a) decimation-in-time and b) decimation-in-frequency,* $N = 8$ [11].

DFTs of size two as illustrated in Fig. 8(a) for $N = 8$. A regular decimation-in-time radix-2 FFT algorithm can be obtained by applying similar decomposition recursively until the entire DFT is constructed only of 2-point DFTs. That is, both of the 4-point DFTs in Fig. 8(a) would have to be decomposed into two 2-point DFTs to obtain the regular DIT radix-2 FFT. Due to the duality between the DIT and DIF algorithms, the DIF radix-2 FFT algorithm is obtained simply by interchanging the role of the time domain and the frequency domain sequences [11] as depicted in Fig. 8(b). The basic building block of the regular radix-2 FFT is a multiplication followed by the 2-point DFT. This operation sequence is often called a *radix-2 butterfly*. The duality between the DIT and DIF algorithms is also reflected to the butterflies. The 2-point DFT F_2, radix-2 DIT butterfly B_2^{DIT} and radix-2 DIF butterfly B_2^{DIF} are defined as [11]

$$F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}; \quad B_2^{\mathrm{DIT}} = F_2 \begin{pmatrix} 1 \\ W_N^i \end{pmatrix}; \quad B_2^{\mathrm{DIF}} = \begin{pmatrix} 1 \\ W_N^i \end{pmatrix} F_2 \tag{6}$$

The total number of butterflies in an N-point radix-2 FFT, $N = 2^k$, is $(N/2)\log_2 N$. Thus, the computational complexity is reduced down to the order of $N \log N$ instead of the complexity of N^2 of the direct DFT computation.

Let us assume an 8-point input sequence $\{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$ is stored, correspondingly, to memory locations $\{0,1,2,3,4,5,6,7\}$. Then, in the first *stage* of computation, two 4-point DFTs of the even-indexed sequence from memory locations $\{0,2,4,6\}$, and of the odd-indexed sequence from memory locations $\{1,3,4,7\}$, are computed and the intermediate results of even- and odd-indexed sequences stored to memory locations $\{0,1,2,3\}$ and $\{4,5,6,7\}$, respectively. In the second stage of computation, four 2-point DFTs from memory locations $\{0,4\}$, $\{1,5\}$, $\{2,6\}$ and $\{3,7\}$ are

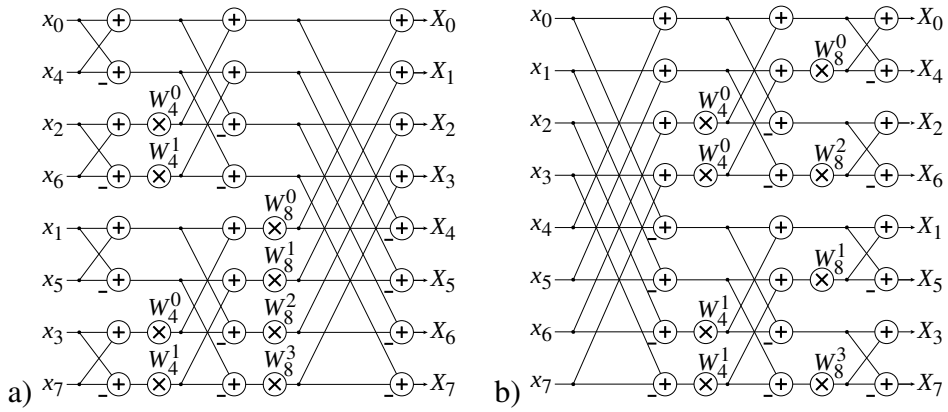


Figure 9. *Signal flow graphs of 8-point decimation-in-time FFT algorithms: a) bit-reversed input, in-order output and b) in-order input, bit-reversed output* [11].

computed and the final results of 8-point FFT stored, respectively, to memory locations $\{0,1\}$, $\{2,3\}$, $\{4,5\}$ and $\{6,7\}$. This kind of procedure, in which the input sequence, intermediate results between stages of computation, and the output sequence of the FFT computation are stored into same memory locations, is called *in-place computation* and the corresponding algorithm an *in-place* FFT algorithm. In-place FFT algorithms are often used since they are compact in terms of memory consumption.

The in-place computation of DIT radix-2 FFT is possible, if its signal flow graph is reordered in an appropriate way from many different choices of reordering. Two most common reorderings allowing the in-place computation are presented in Fig. 9. In Fig. 9(a), the input is *bit-reversed* and the output is *in-order*. Meanwhile, in Fig. 9(b), the input is in-order and the output is bit-reversed.

The traditional bit-reversed input, in-order output radix-2 DIT algorithm is defined as [12]

$$\begin{aligned} F_{2^n} &= \left[\prod_{s=n-1}^{0} \left(I_{2^{(n-s-1)}} \otimes F_2 \otimes I_{2^s}\right)\left(I_{2^{(n-s)}} \otimes T_{2^s}\right)\right] P_{2^k}^r \\ T_J &= \left(I_{J/2} \oplus D_{J/2}\right) \\ D_{J/2} &= \operatorname{diag}\left(W_J^k\right),\ 0 \le k < J/2 \\ F_2 &= \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \end{aligned} \tag{7}$$

where $\otimes$ denotes tensor product and $\oplus$ matrix direct sum, I_k is the identity matrix of order k, F_2 is the 2-point DFT matrix, and P_N^r is a bit-reversed permutation matrix of order N. The bit-reversed permutation matrix can be defined with the aid of a stride

permutation as

$$P_{2^n}^r = \prod_{i=0}^{n-2} (I_{2^{(n-i-2)}} \otimes P_{2^{(i+2)},2}). \tag{8}$$

The matrix $P_{N,K}$ is stride-by-K permutation matrix of order N defined as [12]

$$P_{N,K} = \begin{cases} 1, & \text{if } n = \text{mod}(mK,N) + \lfloor mK/N \rfloor \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

where $\lfloor \cdot \rfloor$ denotes floor function and $\text{mod}(\cdot,\cdot)$ is modulus function.

3.2.3 Radix-4

Computationally efficient forms of the Cooley-Tukey algorithms can be derived by using the number four as the radix. Radix-4 FFT algorithms, in which the size of the transform has to be a power of four, are efficient since the 4-point DFT F_4 can be computed without multiplications [11]. The 4-point DFT F_4, *radix-4 butterfly* B_4^{DIT} and B_4^{DIF} are defined as

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix}; \; B_4^{\text{DIT}} = F_4 \begin{pmatrix} 1 \\ W_N^i \\ W_N^{2i} \\ W_N^{3i} \end{pmatrix}; \; B_4^{\text{DIF}} = \begin{pmatrix} 1 \\ W_N^i \\ W_N^{2i} \\ W_N^{3i} \end{pmatrix} F_4 \tag{10}$$

The traditional in-order input, permuted output, decimation-in-time radix-4 FFT is defined as [13]

$$F_{4^n} = \left[\prod_{s=n-1}^{0} (I_{4^{n-s-1}} \otimes F_4 \otimes I_{4^s}) S_{2^n}^s \right] P_{4^n}^{in4} \tag{11}$$

$$S_N^s = I_{N/4^{(s+1)}} \otimes \text{diag}\,(M_{s,0}, M_{s,1}, M_{s,2}, M_{s,3}) \tag{12}$$

$$M_{s,k} = (W_{4^{(s+1)}}^{km}),\ 0 \le m < 4^s - 1 \tag{13}$$

$$P_{4^n}^{in4} = \prod_{k=1}^{n} I_{4^{(n-k)}} \otimes P_{4^k,4}. \tag{14}$$

The length N of the complex input sequence was chosen to be $N = 1024 = 4^5$ in the realizations of Chapter 4; i.e., the radix-4 approach could be utilized in all of the realizations. Therefore, a variation of the *1024-point complex in-place radix-4 DIT FFT algorithm*, where the input is *permuted* and the output is *in-order*, was realized on TTA in this work. To be exact, this FFT algorithm is a different formulation of the traditional radix-4 DIT FFT algorithm defined with the aid of Equations 11, 12, 13 and 14. These equations interleave the operations from different butterflies due to the term

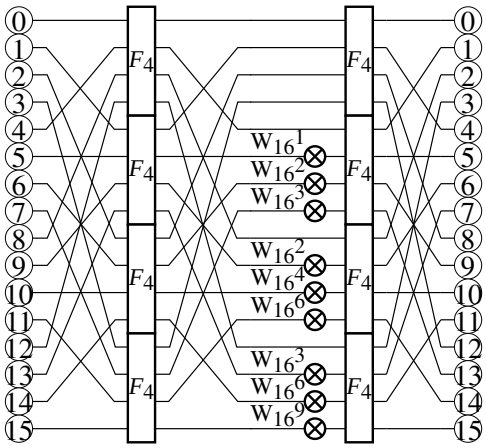


Figure 10. *Signal flow graph of the FFT algorithm for TTA in the 16-point radix-4 DIT case.*

$(I_{4^{n-s-1}} \otimes F_4 \otimes I_{4^s})$. The algorithm realized on TTA has been formulated from these equations and can be defined exactly as

$$F_{4^n} = \left[\prod_{s=n-1}^{0} [P_{4^n}^s]^T (I_{4^{n-1}} \otimes F_4) D_{4^n}^s P_{4^n}^s \right] P_{4^n}^{in4} \tag{15}$$

where P_N^{in4} is the input permutation matrix defined in (14), P_N^s is a permutation matrix of order N, and D_N^s is a diagonal matrix of order N, which contains N twiddle factors as follows

$$D_N^s = \bigoplus_{k=0}^{N/4-1} \operatorname{diag}\left(W_{4^{s+1}}^0, W_{4^{s+1}}^{\operatorname{mod}(k,4^s)}, W_{4^{s+1}}^{2\operatorname{mod}(k,4^s)}, W_{4^{s+1}}^{3\operatorname{mod}(k,4^s)} \right). \tag{16}$$

The permutation matrix P_N^s is given as

$$P_{4^n}^s = I_{4^{(n-s-1)}} \otimes P_{4^{(s+1)},4^s}. \tag{17}$$

This permutation matrix indicates directly the access order of operands for butterfly operations in processing column s. This can be seen in Fig. 10 where the signal flow graph of the algorithm in (15) is illustrated.

The principle in realizing this radix-4 DIT FFT algorithm by utilizing a computer is the following for an input sequence I_{16} of size $16 = 4^2$:

1. Allocate array T of size $2 \cdot 16 = 32$ for twiddle factors W_{16}^R and array I of size 16 for input sequence I_{16}. The twiddle factors are stored separately, in the order they appear in the signal flow graph, for each stage of the FFT computation. Therefore, the needed size s for the array of twiddle factors is $s = Nlog_4 N$.

2. Initialize T by storing correct twiddle factors, which are computed beforehand, in correct order into it.

3. Initialize I by storing the input sequence I_{16} into it.

4. Perform the input permutation to the input sequence I_{16}; i.e., reorder input array I according to the signal flow graph of Fig. 10.

5. Evaluate the four butterflies of the first stage S_0.

6. For each of the four butterflies of the second stage S_1:

   - Compute four indexes of the input array I and evaluate the butterfly by accessing the input array I with the computed indexes and by performing the complex butterfly multiplications and summations to the accessed values.
   - Store the results of the butterfly evaluation back to the input array I into the same index locations that were computed at the beginning of this step for the butterfly to be evaluated.

In the steps four, five and six, the input array I has to be always accessed according to a certain, correct method to perform the FFT computation correctly. This method involves both the input permutation and the computation of the indexes of the input operands of the butterflies. By applying this method, the correct *indexes can be generated* for accessing the input array I. Therefore, this method is called *index generation*. The index generation of the radix-4 DIT FFT realized on TTA is discussed in the following section.

3.3 Index Generation

In this section, the fundamental index generation principle of the radix-4 DIT FFT algorithm realized on TTA for permuting and accessing the input array I is explained in detail. Firstly, for both the input permutation and the butterfly operand access, the discussion gets of the ground from the exact matrix equations describing them. Then these equations are applied for a couple of input sequences of different lengths N. Next, the results of applying these equations are shown in tabulated forms, in which the associations between the inputs and outputs of the index generation process are illustrated both in decimal and binary representations.

Table 1. *The flow of linear* ***i*** *and* ***index*** *in the* ***input permutation*** *of radix-4 DIT FFT realized on TTA for a) $N = 16$ and b) $N = 64$.*

a)

i		**index**	
decimal	binary	decimal	binary
0	**0000**	0	**0000**
1	**0001**	4	**0100**
2	**0010**	8	**1000**
3	**0011**	12	**1100**
4	**0100**	1	**0001**
.	.	.	.
.	.	.	.
9	**1001**	6	**0110**
10	**1010**	10	**1010**
11	**1011**	14	**1110**
12	**1100**	3	**0011**
13	**1101**	7	**0111**
14	**1110**	11	**1011**
15	**1111**	15	**1111**

b)

i		**index**	
decimal	binary	decimal	binary
0	**00**0000	0	**00**0000
1	**00**0001	16	**01**0000
.	.	.	.
.	.	.	.
6	**00**0110	36	**10**0100
.	.	.	.
.	.	.	.
31	**01**1111	61	**11**1101
32	**10**0000	2	**00**0010
33	**10**0001	18	**01**0010
.	.	.	.
.	.	.	.
62	**11**1110	47	**10**1111
63	**11**1111	63	**11**1111

Finally, based on the observations, which will be seen from the tabulated examples, the fundamental index generation principle for both the input permutation and butterfly operand access is clarified and illustrated. The input permutation is described in Subsection 3.3.1 and the butterfly operand access in Subsection 3.3.2.

3.3.1 Input Permutation

The purpose of the input permutation is to reorder the input sequence $I_N = \{x_0, x_1, x_2, ..., x_{N-1}\}$ in an appropriate way before the actual butterfly computations on different stages of FFT. The input permutation matrix of the radix-4 DIT FFT realized on TTA was defined in Equation 14. Table 1 has been generated by applying this permutation matrix for the input sequences I_{16} and I_{64}. This table illustrates from which **index** location of the original input sequence, the item x_i of the permuted sequence should be read from; i.e., the item x_i of the permuted sequence equals the item x_{index} of the original input sequence.

By investigating the binary representations of *i* and *index* in the table 1, the possible associations and regularities between *i* and index can be examined. In Table 1(a), the

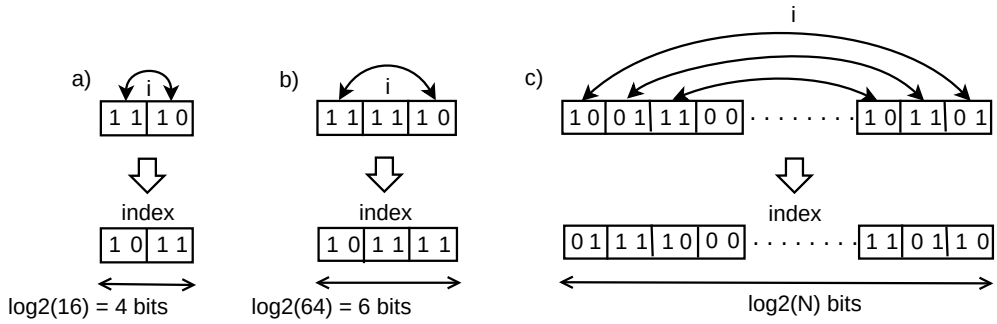


Figure 11. *The formation of* ***index*** *from the linear* ***i*** *in the* ***input permutation*** *of radix-4 DIT FFT realized on TTA for a)* $N = 16$*, b)* $N = 64$*, and c)* $N = 4^k$.

index can be generated from **i** by swapping the two bits' LSB- and MSB-parts of **i** as illustrated in Fig. 11(a). Meanwhile, in Table 1(b), the **index** can be generated from **i** by swapping the two lowermost and two uppermost bits with each others and by leaving the middlemost two bits untouched as illustrated in Fig. 11(b). That is, it seems the **index** *can be generated from* ***i*** *by simply swapping two bits' fields from the LSB- and MSB-parts of* ***i*** *in pairs*. The previous observation can be generally proved for the input permutation and it is, therefore, also generally valid for the FFT transform of size $N = 4^k$ as illustrated in Fig. 11(c).

3.3.2 Operand Access

As was already mentioned in Section 3.2.3, the permutation matrix $P_{4^n}^s$ defined in Equation 17 indicates directly the access order of operands for butterfly operations in processing column s. This access order was illustrated in Fig. 10 for the transform of size $N = 16$.

Let us apply the algorithm in (15) for an input sequence of size $N = 256$. In this resulting 256-point FFT, the number of stages of computation, S, is $S = \log_4 N = 4$ and the number of radix-4 butterflies in a single stage, B, is $B = N/4 = 64$. Each single radix-4 butterfly takes four complex input operands and produces four complex outputs. The number of butterfly input operands, K, in each stage of the 256-point FFT is thus $K = 4B = 4N/4 = N = 256$. Table 2 shows the results of applying this algorithm from the index generation point of view.

In this table, from the different stages s of the 256-point FFT, a couple of examples of the associations between inputs and outputs of the index generation process for the butterfly operand access are shown. The input of the index generation process for the butterfly operand access is a linear counter **i**, that always counts the input operands of

Table 2. *The flow of linear* ***i*** *and* ***index*** *on different stages* ***s*** *of the radix-4 DIT FFT algorithm realized on TTA for* $N = 4^4 = 256$.

s	**i**		**index**	
	decimal	binary [8 bits]	decimal	binary [8 bits]
0	0	00000**00**	0	00000**00**
	:	:	:	:
	64	01000**00**	64	01000**00**
	:	:	:	:
	255	11111**11**	255	11111**11**
1	:	:	:	:
	1	0000**0001**	**4**	0000**0100**
	:	:	:	:
	7	0000**0111**	**13**	0000**1101**
	:	:	:	:
2	:	:	:	:
	1	00**000001**	**16**	00**010000**
	:	:	:	:
	23	00**010111**	**53**	00**110101**
	:	:	:	:
3	:	:	:	:
	1	**00000001**	**64**	**01000000**
	:	:	:	:
	75	**01001011**	**210**	**11010010**
	:	:	:	:

the butterflies in each single stage; i.e., **i** counts always from zero to 255. The output of this process is then the corresponding **index** where an input operand of a butterfly is located at, and which should be somehow generated on the basis of the linear **i** and the current stage **s**.

By investigating the binary representations of *i* and *index* on the different stages *s*, few interesting observations can be made. Firstly, in the stage zero, **i** and the **index** are the same. Secondly, in the stages one, two and three, the index can always be generated from *i* by manipulating bit fields of different lengths in the LSB-part of it; i.e., in the stage one the length equals four bits and in the stages two and three, respectively, six and eight bits. As a conclusion from the previous observations, it can be stated that the length of the manipulated bit field seemed to be dependent on the stage of the

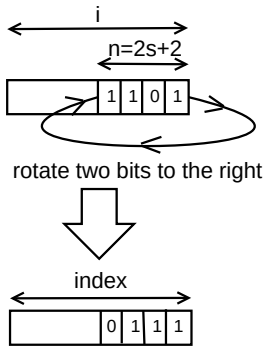


Figure 12. *The formation of* ***index*** *from the linear* ***i*** *in the butterfly* ***operand access*** *of the radix-4 DIT FFT realized on TTA.*

FFT computation. The manner, with the aid of which the bit fields of varying lengths are manipulated, is *right rotation*; i.e., *the index can always be generated from i by simply rotating the lowermost n bits of i two bits to the right*. For the index generation process for the butterfly operand access, it can be generally proved that the number of the lowermost bits of i, n, to be rotated two bits to the right depends on the stage of the FFT, s, as

$$n = 2s+2, \quad 0 < s \leq log_4 N - 1 \tag{18}$$

As a summary, the principle for accessing the operands of the butterflies in the radix-4 DIT FFT realized on TTA can be summarized into the following two steps:

1. Compute the number of the lowermost bits of i, n, to be rotated two bits to the right according to Equation 18.
2. Form the **index** by rotating the n lowermost bits of i two bits to the right.

These steps are illustrated in Fig. 12.

Implementing the index generation principle on TTA for both the input permutation and the butterfly operand access invokes interesting questions and ideas. Especially interesting would be to clarify how the special characteristics of TTAs, i.e., the user-defined SFUs, could be exploited and how they affect on the performance of the FFT computation. Among other things these ideas are experimented and the answers to these questions given in Chapter 4.

4. HLL IMPLEMENTATIONS

Software process becomes often a bottleneck when companies are trying to launch DSP-products on the market in time. Therefore, HLLs and HLL-compilers are used in the co-design process of a DSP-system for both easing and fastening the work of SW designers. It is also quite common that high demands are set for the performance of the HLL-compiler meaning that compiled programs should be fast and fit to a compact memory.

In this chapter, the radix-4 DIT FFT is implemented using HLL code to see how effectively the HLL-compiler of the MOVE Framework can implement the FFT application. In the HLL implementations of this chapter, two important tools from the MOVE Framework have been used. These are the *scheduler* and the *design space explorer*.

First, the radix-4 DIT FFT was written in ANSI-C by using only integers and C-language's basic operations. Thereafter, the code and the architecture of the processor were incrementally improved to achieve better performance in terms of elapsed clock cycles. The following improvements were made:

1. Two SFUs were added to the architecture for performing complex arithmetic.
2. An SFU was added to the architecture for performing index generation, extra multiplications were removed from the code, and the structure of the inner loop was modified slightly.

In the last phase of this semi-automated implementation process with the C-compiler, the design space explorer was used to obtain the pareto curves which illustrate the improvements that were done.

The features that are common to all the implementations are discussed briefly in Section 4.1. Next, the C-language implementations, which are henceforth to be called the *Cases*, are descriped in Sections 4.2, 4.3, and 4.4 in detail. The exploitation of the design space explorer is discussed briefly in Section 4.5.

4.1 Common Features

The starting point for this experiment is a reference implementation that has been made with MATLAB so that the outputs of the C-language implementations can be verified by comparing to the output of the reference implementation.

Common to all the cases are the following issues:

- Each of the cases implement the 1024-point radix-4 DIT FFT.
- No memory optimizations for the coefficients have been considered, i.e., input data and twiddle factors are stored into one-dimensional tables.
- Complex numbers are represented as 32-bit integers in such a way that the uppermost 16 bits are reserved for the real part and the lowermost 16-bits for the imaginary part.
- Computations are performed in the 16-bit fractional number presentation.
- The result of the real multiplication is always shifted 15 bits to the right in complex multiplications since the result can be 32-bit at the maximum as two 16-bit real numbers are multiplied.
- The result of the real addition is always divided by two, i.e., shifted one bit to the right in complex additions for avoiding possible overflow. This implicates that the final results of the transform have been divided by N since a result is always divided by the radix, r, inside a single computation stage and there exists $\log_r N$ computation stages in total in the N-point transform.

The last issue does not cause a big problem since the scaling of the results can easily be compensated by shifting the results the same number of bits to the left as they were shifted to the right in complex additions.

4.2 Case 1: ANSI-C Code

The first case was intentionally made as simple as possible and no optimisations were considered. The main purpose of the first case is only to have it working properly, i.e., equally as the MATLAB-reference.

The source code can be found from Appendix A. As can be seen from the source code, it is written in ANSI-C by using only integer data-types and C-language's basic operations. The code is written by using three loops. The input permutation is performed in the first loop with the aid of the `swapBitPairs`-function that performs the needed bit-level manipulation to the linear index as was already illustrated in the Fig. 11(c) in Chapter 3.

The other two loops are nested and they are performing the butterfly computations through the stages of the N-point FFT application. The outer of the loops iterates through the stages, i.e., from zero to $log_4 N - 1$. The inner loop iterates always from zero to $N - 1$ in the incremental steps of four. The $N/4$ radix-4 DIT butterflies of a single computation stage are evaluated in the inner loop.

One radix-4 DIT butterfly is evaluated in the kernel of the inner loop. This is done by first computing the correct indices from where the input operands of the butterfly are read. The operands can be accessed by using the function `r4bfin_idx` that performs the needed bit-level manipulation to the linear index as was already illustrated in the Fig. 12 in Chapter 3. Next, the real and imaginary parts of the operands are read from the output-buffer into variables. After that the complex multiplications and additions of a butterfly can be performed as shown in Appendix A. Finally, by exploiting the *in-place* computation, the results of the evaluation of a butterfly can be stored back to the same index locations from where the operands were read, as can be seen from the code.

It can be seen by examining the code that quite many variables are required for storing the real and imaginary parts of the complex numbers. In addition, and-, shift-, and or-operations are needed when complex numbers are processed in the code. These operations make the code quite tedious to follow. For the previous reasons, the following data-type and macros were added to the code:

```
typedef union {
    int word;
    struct {
       short int imag;
       short int real;
    }cplx;
}Complex;

#define Real(a) (a.cplx.real)
#define Imag(a) (a.cplx.imag)
#define Word(a) (a.word)
```

Now a complex number can be represented conveniently with the data-type. The accessing of the number becomes also easier by using the macros. By writing the code of

Appendix A by using the union-type and the macros, the following of the code became easier in pursuance of the functionality and the performance remained constant.

4.3 Case 2: SFUs for Complex Arithmetic

As can be seen from the Equation 10 in Chapter 3, the evaluation of the radix-4 DIT butterfly requires both complex multiplication and addition. These are operations for which custom hardware can be easily tailored to obtain more computational power. Thus, in this case, two separate SFUs were added to the architecture of the processor for performing complex arithmetic.

The basic structure of the code remained constant meaning that loop structure remained the same as in the previous case. Meanwhile, the structure of the kernel of the inner loop became much simpler. Now, one can just call the functions that are simulating the functionality of the SFUs from the scheduler's backend as can be seen from Appendix B.

The HW implementations of these SFUs are discussed briefly at the block diagram level in the following subsections. The complex multiplier is described in Subsection 4.3.1 and the complex adder in Subsection 4.3.2.

4.3.1 Complex Multiplier

The block diagram of the complex multiplier is presented in Fig. 13. The complex multiplier calculates the product of two 32-bit complex numbers in which the uppermost 16 bits are reserved for the real part and the lowermost 16-bits for the imaginary part. As can be seen from Fig. 13, the product can be calculated by using four real multipliers and two real adders.

The 16-bit real multipliers calculate the needed partial products of the real and imaginary parts as illustrated in the block diagram of Fig. 13. The partial products are shifted 15 bits to the right before the additions to obtain the real and imaginary parts of the product. The scaling can be implemented easily using only correct wiring, as depicted in Fig. 13. This scaling has to be done since the result of the multiplication can be 32 bits in length at the maximum as two 16-bit numbers are multiplied with each other. The loss of accuracy is, therefore, minimized only to the loss of the 15 least significant fractions.

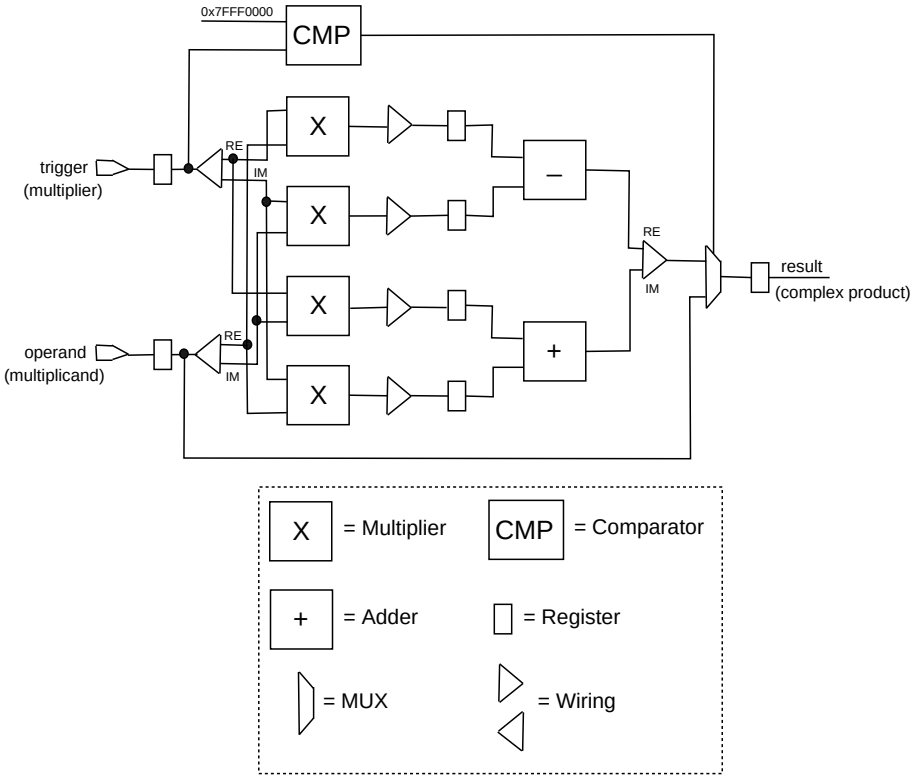


Figure 13. *Block diagram of the complex multiplier.*

The comparator is used for detecting such kind of a situation in which the value of the multiplier is one (0x7FFF0000). In that case, the value of the multiplicand can directly be passed to the result register by the mux. The mux is controlled by the comparator as illustrated in Fig. 13.

Furthermore, the complex multiplier has been pipelined by using pipeline registers between the real multipliers and the real adders. Higher clock frequencies can be obtained for the complex multiplier in the synthesis by using the pipelining. Since there are registers also in the inputs and in the output of the complex multiplier, the *latency* of the unit is three clock cycles.

4.3.2 Complex Adder

The C-language function simulating the functionality of the complex adder can be found from Appendix B. The `cadd`-function could be written by simplifying the definition of the radix-4 DIT butterfly, which can be found from the Equation 10 in Chapter 3. The definition was simplified as much that the real and the imaginary parts of the complex sums could be written by using the real and the imaginary parts of the four input complex numbers to be added together. There are four slightly differ-

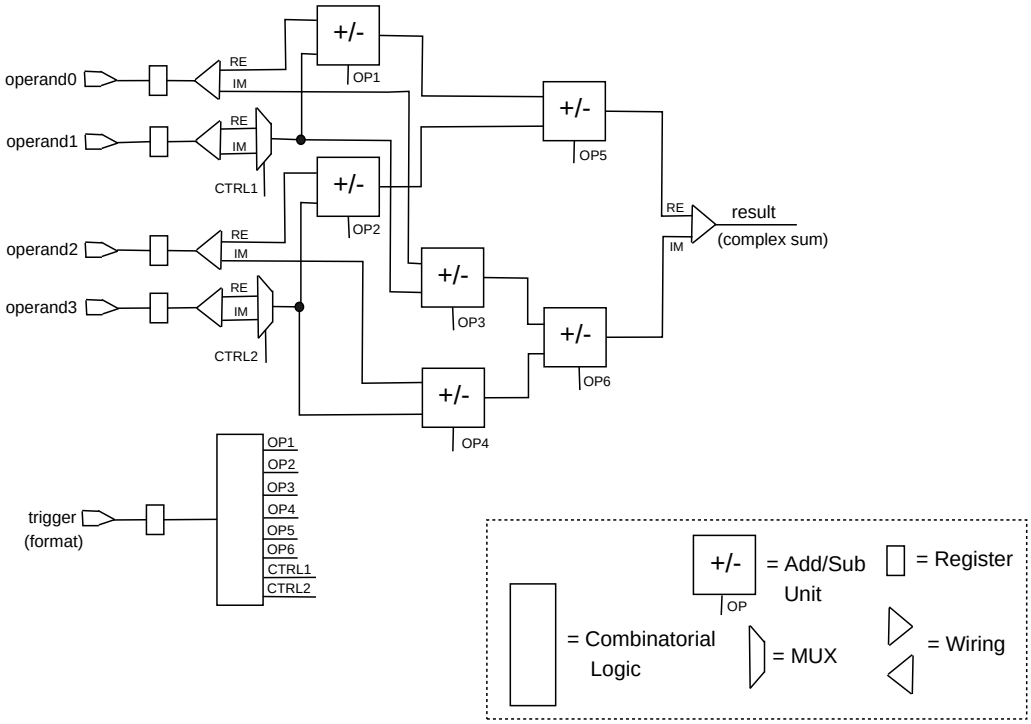


Figure 14. *Block diagram of the complex adder.*

ent summation formats in the evaluation of the radix-4 DIT butterfly. These formats can be simulated easily by using a simple switch-case structure as can be seen from Appendix B. In this switch-case structure, the summation format is used to select the complex summation to be performed.

This kind of a structure can also be implemented quite easily on HW. Thus, it was decided to implement the complex adder as depicted in Fig. 14. The four complex numbers to be added together are written to the operand registers of the unit. The summation format (0, 1, 2, or 3) is written to the trigger register of the unit, i.e., the format triggers the unit. The unit has one output: the result of the single summation format. Thus, to be able to evaluate the complex summations of the single radix-4 DIT butterfly, all the four summation formats have to be calculated meaning that the unit has to be triggered four times with all the values 0, 1, 2, and 3.

The different summation formats can be computed by using six 16-bit add/sub-units as can be seen from Fig. 14. The add/sub unit can perform either addition or subtraction. The operation to be performed by this unit is selected by a one-bit opcode. In addition, the add/sub unit shifts its result one bit to the right to avoid the possible overflow. This can be seen also from the SW simulation function of Appendix B. The opcodes can be specified easily on the basis of the format. A combinatorial logic network can produce the needed opcodes for the add/sub-units in different summation formats as

can be seen from Fig. 14. Also the multiplexors are controlled by using this network. Since there exists registers only in the inputs of the complex adder, the *latency* of the complex adder is *one* clock cycle.

4.4 Case 3: SFU for Index Generation

The index generation is a very important operation in the FFT computations as was already discussed in Chapter 3. The index generation of the radix-4 DIT FFT algorithm realized in this work can be made in the bit-level, i.e., by manipulating bit-level representations of the data items present in the index generation process. These bit-level operations can be performed fastly on HW, by using custom HW. Thus, in this case, an SFU was added to the architecture to perform index generation.

This SFU is described in Subsection 4.4.1. The structure and the functionality of the source code are explained briefly in Subsection 4.4.2.

4.4.1 Index Generator

The index generator is capable of performing the memory address arithmetics that is needed in both permuting the input and accessing the operands of the butterflies. The index generator has four inputs:

1. **Base address of the input buffer** (operand 0).
2. **Base address of the output buffer** (operand 1).
3. **Stage** of the FFT computation (operand 2).
4. **Index, i**, which is linearly traversing through the inputs of the radix-4 DIT butterflies (trigger).

Based on the four inputs, the index generator produces, concurrently, two memory addresses as its outputs:

1. The first one is an address of the input buffer and it is generated for the input permutation.
2. The other one is an address of the output buffer.

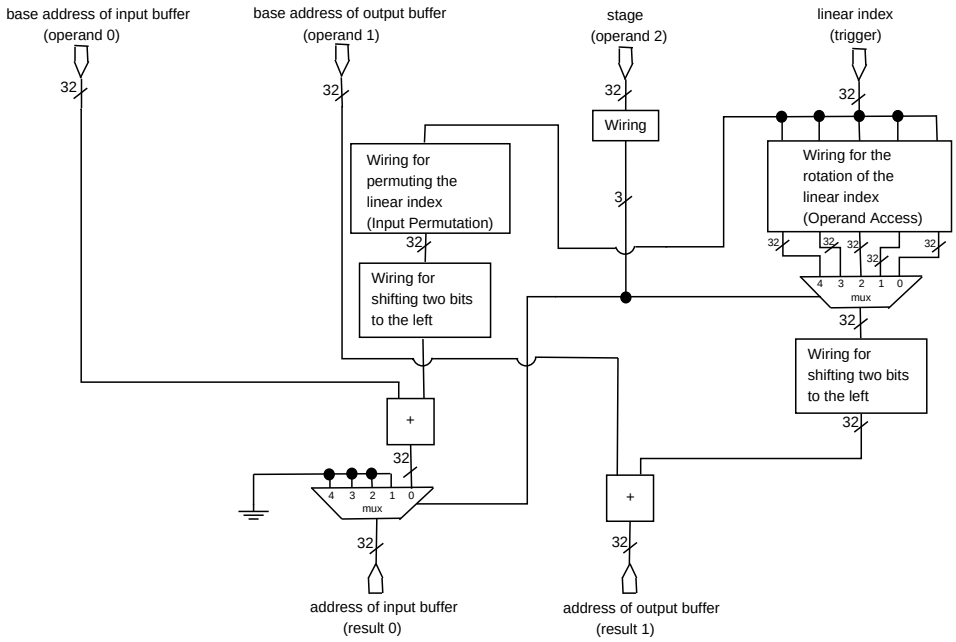


Figure 15. *Block diagram of the index generator.*

The block diagram of the index generator is depicted in Fig. 15. The two output addresses are generated according to a method, which depends on the current stage of the FFT computation. In the first stage, zero, the address of the input buffer is generated according to the input permutation principle, i.e., by manipulating the lowermost ten bits of **i** as illustrated in Fig. 11(c) in Chapter 3. This bit-level manipulation can be implemented on HW by using only correct wiring as can be seen from Fig. 15. Due to the implementation technique of the load-store unit (LSU), the manipulated index has to be shifted two bits to the left before the address of the input buffer is computed by adding the base address of the input buffer together with the manipulated index as illustrated in Fig. 15. In the stage zero, the address of the output buffer is always the linear address that corresponds to the value of the linear index, **i**. Thus, the results of the butterfly evaluations are stored linearly, in the increasing order, to the output buffer.

In the other stages (1, 2, 3, and 4), the address of the input buffer is negligible from the FFT computation point of view, since the *in-place computation* is performed inside the output buffer. Thus, the address of the input buffer is driven to zero in the other stages as can be seen from Fig. 15. Meanwhile, the address of the output buffer is now generated by applying the operand access principle of the butterflies which was discussed in Section 3.3.2. I.e., the linear index, **i**, is first rotated two bits to the right as illustrated in Fig. 12 in Chapter 3, and then added together with the base address of the output buffer as shown in the block diagram.

4.4.2 Source Code

The source code of the case 3 can be found from Appendix C. The source code has been further implemented by using three loops. The input permutation and the butterfly evaluations of the stage zero are performed in the first loop by reading the operands of the butterflies from the input buffer by using the index generator. The butterflies are evaluated next by computing the needed complex multiplications and additions. In the end of the first loop, the results of the butterfly evaluations are written linearly to the output buffer.

The other two loops are nested and they are performing the butterfly evaluations of the other stages (1, 2, 3, and 4). The structure of the inner loop is equal to that of the first loop but now the input operands are read from the output buffer and the results of the butterfly evaluations are also written to the same buffer by exploiting the *in-place computation* characteristic of the realized radix-4 DIT FFT algorithm.

Instead of one single radix-4 DIT butterfly, two butterflies are evaluated at a time inside both the input permutation and the inner loop to obtain more computation speed. The scheduler of the MOVE Framework can namely exploit the ILP better when the C-code is written in this way.

Extra multiplications have been also removed from the code. Every fourth complex multiplication can be removed from the code since every fourth twiddle factor, W_{4k}, is one. Thus, there is no need to multiply the input operands x_{4k} into the complex multiplier but they can be temporarily stored for waiting the complex addition to take place.

4.5 Explorations

After the C-codes of the cases were implemented, the cases were scheduled and simulated with the tools of the software subsystem. This was done to be able to verify the correct functionality of the cases. After the cases were verified, the design space explorer was used for obtaining the pareto curves, and for optimizing the connectivity.

The resource optimization phase is explained in Section 4.5.1 and the connectivity optimization phase is discussed in Section 4.5.2.

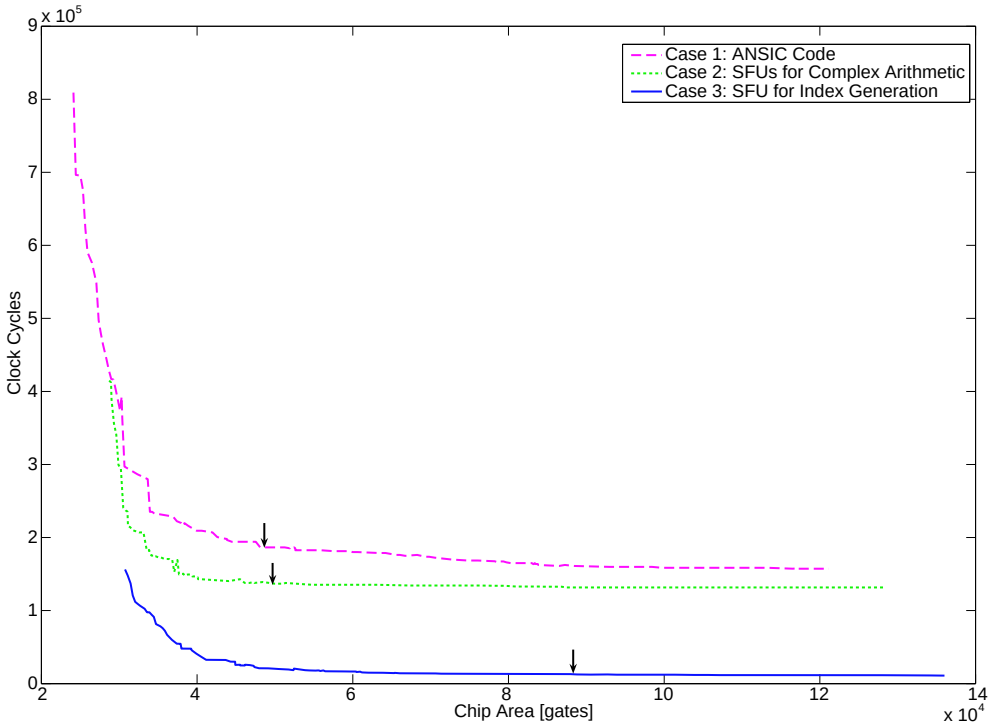


Figure 16. *Results of resource explorations. The three configurations selected for the connectivity optimizations are marked with arrows in the figure.*

4.5.1 Resource Optimization

In the resource exploration, the amount of resources was chosen to be oversized in the initial processor configurations so that the design space would be large enough. This meant that there were 16 buses and four FUs supporting each needed operation set. RFs were chosen to contain enough registers. The pareto curves of the cases, obtained from the resource explorations, are shown in Fig. 16. The chip area is shown in the x-axis and the number of elapsed clock cycles in the y-axis of the pareto curves.

It can be seen from the pareto curves that the improvements made to the code and the architecture have been succesfull. The complex multiplier and the complex adder together improved the cycle count about 19%. The index generator further improved the cycle count about 85% which is a tremendous improvement.

4.5.2 Connectivity Optimization

After the resource optimization phase, one processor configuration from each of the cases was chosen to the connectivity optimization phase. A medium configuration having a chip area of about 50 kgates was chosen for the cases one and two. The medium configuration can be considered as a compromise between processor cost and

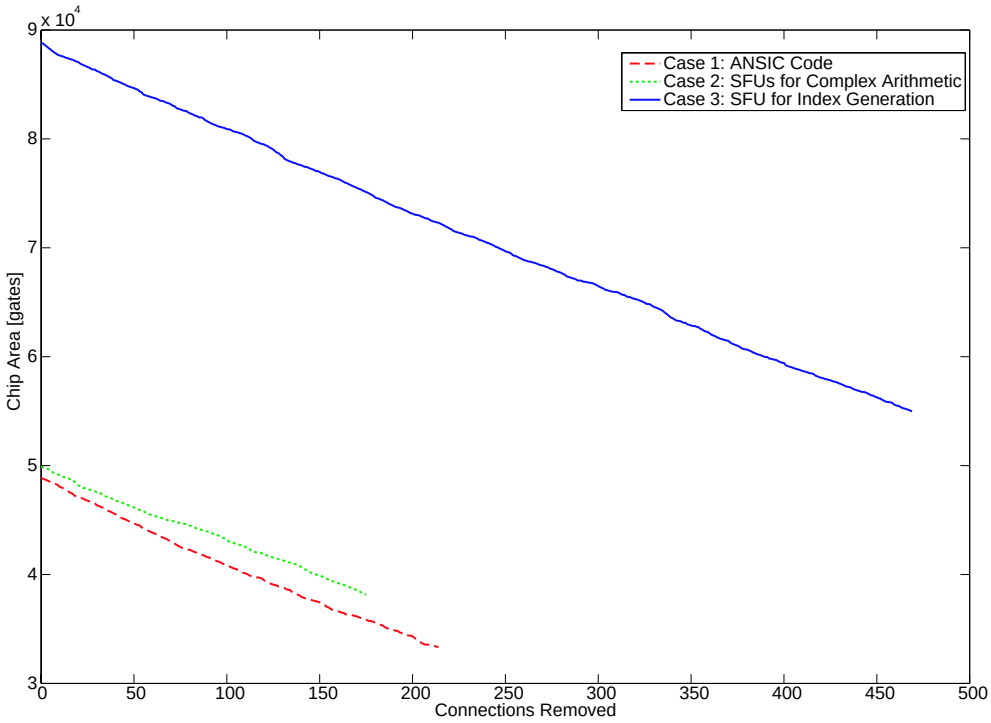


Figure 17. *Results of connectivity optimizations on processor configurations shown in Fig. 16.*

performance meaning that the costs are relatively low and the performance is, however, relatively high. A high-performance configuration having a chip area of about 88 kgates was chosen for the case three to be able to see what kind of a performance can be obtained for the 1024-point radix-4 DIT FFT by using the scheduler.

The connectivity exploration did not improve the performance. However, the needed chip area decreased notably as can be seen from Fig. 17. The essential characteristics of the three cases are shown in Table 3. It can be stated by examining Table 3 that the maximum performance for the 1024-point radix-4 DIT FFT was 12 kcycles when the application was written in C-code and the code was compiled using the software subsystem of the MOVE framework. The following two chapters show how fast and effectively the 1024-point radix-4 DIT FFT can be performed by using the TTA assembler.

Table 3. *The characteristics of the cases.*

Case	Configuration	Clock Cycles	Connections Removed	Area [kgates]
1	Medium	**186487**	100	41
2	Medium	**136810**	100	43
3	High-Performance	**12366**	400	59

5. OPTIMIZED ASSEMBLER IMPLEMENTATION

In the DSP application domain, many optimization techniques are exploited in the advanced HLL compilers to be able to produce effective binary code. One of these techniques is *Software Pipelining* which allows utilization of the available HW level parallelism by the program code.

Due to the fact that SW pipelining was not effectively supported by the current HLL compiler of the MOVE Framework, the performances of the HLL implementations were not below the desired level in terms of clock cycles. For this reason, it was decided to optimize the 1024-point radix-4 DIT FFT manually. First, an effective method was discovered to get the performance of the FFT computation below the desired level and then a TTA assembly language description of this method was written. Finally, a TTA processor, on which the designed assembly code can run on, was constructed and analyzed.

This manual implementation process with the aid of the TTA assembler is described in detail in this chapter. The method for improving the performance of FFT computation is discussed in Section 5.1. The assembly programming process is described in detail in Section 5.2 and the HW implementation of the processor, respectively, in Section 5.3.

5.1 Operation Scheduling

It was found out by analysing different choices for improving the performance of FFT computation that pipelining, in which the utilizations of performance-critical operations would be as high as possible, could be implemented fairly easily also manually. In the same analysis, it was also found that with the aid of this kind of pipelining, the performance of FFT could be improved remarkably. For these reasons, it was decided to schedule the operations of the 1024-point complex radix-4 DIT FFT manually in a pipelined fashion. The needed operations for performing the FFT are described in Subsection 5.1.1. In Subsection 5.1.2, the scheduling principle is described with the aid of

an example. An effective manner for the evaluation of several radix-4 DIT butterflies is described in Subsection 5.1.3.

5.1.1 Operations

The manual schedule was done with the aid of the same user-defined operations whose HW-implementations were already described in Chapter 4. Memory addresses of operands can be generated with the aid of operation AG, which has always two results. The first is an address of the input buffer and the second an address of the output buffer. Complex multiplications can be performed with the aid of operation CMUL and the complex additions, respectively, with the aid of operation CADD.

Real addition is needed for two purposes; for generating both linear indices of the address generation and linear addresses of the twiddle factors. The real addition is denoted with the operation ADD. Data memory accesses can be performed with the aid of operations LD and ST. The LD operation loads a word from the memory and the ST operation stores a word to the memory. Input operands of the butterflies and twiddle factors need to be loaded from the data memory and the results of butterfly evaluations must be stored back to the data memory. The operations ADD, LD and ST are overwhelmingly the most often used operations in the FFT computation. Thus, an effective utilisation of these operations is very essential from the performance point of view, i.e., there should not exist such clock cycles, in which those operations are not simultaneously utilized.

Furthermore, comparison operations EQ and GTU are needed for evaluating conditional moves and operation JUMP for jumping in the loops. The essential characteristics of operations to be manually scheduled is shown in Table 4.

5.1.2 Scheduling Principle

The radix-4 DIT butterfly defined in (10) can be computed as

$$\begin{cases} y_0 = x_0 + W_1 x_1 + W_2 x_2 + W_3 x_3 \\ y_1 = x_0 - jW_1 x_1 - W_2 x_2 + jW_3 x_3 \\ y_2 = x_0 - W_1 x_1 + W_2 x_2 - W_3 x_3 \\ y_3 = x_0 + jW_1 x_1 - W_2 x_2 - jW_3 x_3 \end{cases} \tag{19}$$

where x_i denotes an input operand of a butterfly, W_i is a twiddle factor, and y_i an output operand of a butterfly.

Table 4. *The characteristic of the operations needed for the FFT application.*

Operation	Latency	Number of inputs	Number of outputs
AG	2	4	2
CMUL	3	2	1
CADD	1	5	1
ADD	1	2	1
LD	3	1	1
ST	3	2	0
EQ	1	2	1
GTU	1	2	1
JUMP	4	1	0

Firstly, let us figure out how a single radix-4 DIT butterfly could be effectively scheduled with the aid of the operations tabulated in Table 4. In Appendix D, there is a chart, in which this effective schedule is depicted. In this chart, clock cycles are increasing in the horizontal direction, and they are marked with yellow color. The needed data moves between different operations for performing a single butterfly are highlighted with red and blue colors in the vertical column on the left margin of the chart. On the left side of the column for data moves, also the buses, on which the single data moves are performed, are highlighted with gray color. The data items that are transported on the buses in different clock cycles are depicted with green boxes.

Let us go this chart through from clock cycle zero to clock cycle 16 for clarifying the schedule of a single butterfly. In cycle zero, the moves highlighted with blue color have to be performed for initialisation purposes, i.e., the input operands of operation AG are transported to the operand registers and addition operation ADD1, for the generation of linear indices, is triggered by transporting zeros to both the operand and trigger registers. The next operation that should be triggered is the address generation, AG, so that the address of the first input operand of the butterfly, ax_0, would be obtained. As the latency of operation ADD1 is one, the result of the addition is ready in cycle one. Therefore, operation AG is triggered in cycle one by transporting the first linear index, 0, from the output of the addition operation to the trigger register of the AG-operation on bus zero. Furthermore, to be able to compute the following linear index that should be one, the new addition operation is triggered by transporting the previous result of addition, 0, to the operand register and value one from RF to the trigger register.

In cycle two, the next linear index, 1, is ready as the result of addition and the generation of the address of the second input operand, ax_1, can be triggered on bus zero.

As in the previous cycle, the addition operation must be again triggered for computing the next linear index, 2. In cycle three, the generation of the address of the third input operand, ax_2, is triggered equally as that of the next linear index, 3. In addition, the result of the first AG operation, ax_0, is ready in cycle three as the latency of AG-operation is two clock cycles. Therefore, the loading of the first input operand of the butterfly, x_0, can be triggered from the data memory with the aid of operation LD. The address of the first input operand, ax_0, must also be stored to the RF so that the results of the evaluation of the butterfly could later be stored back to the data memory. Also the generation of the address of the first twiddle factor, aW_1, has to be triggered in cycle three by transporting number four from RF to the trigger register of operation ADD2 that is linearly computing the addresses of the twiddle factors. The generation of addresses of twiddle factors has to be started precisely in cycle three so that the first factor, W_1, would be loaded from the data memory in cycle seven when the first complex multiplication can be performed.

In cycle four, the generation of ax_3 is triggered equally as the load of the second input operand, x_1. Address aW_1 is also ready in this cycle, i.e., the load of W_1 must be triggered. In cycles five and six, the loads of operands x_2 and x_3 as well as that of factors W_2 and W_3 are triggered. Furthermore, the memory load of operand x_0 is ready in cycle six and it has to be stored into RF, thus it is the first operand of the complex addition to be triggered in cycle 12.

In cycle seven, the loads of x_1 and W_1 are ready. Thus, the first complex multiplication can be triggered on buses eight and nine. Equally, in cycles eight and nine, the loads of operands x_2 and x_3 as well as the loads of factors W_2 and W_3 are ready and the rest of the multiplications of the butterfly can be triggered. The results of the first, second and third multiplication are ready, respectively, in cycles 10, 11 and 12. Therefore, the first complex addition for computing result y_0 can not be triggered until cycle 12, since all the three results of multiplications have to be ready before the first complex addition can be triggered. Therefore, the first and the second result of complex multiplication have to be temporary stored into RF in cycles 10 and 11.

In cycle 12, the complex addition is triggered for the first time by transporting $prod1 - prod3$ and x_0 to the operand registers, and zero from the RF to the trigger register. In cycles 13, 14, and 15, the complex addition is further triggered by the values one, two, and three for computing, respectively, y_1, y_2, and y_3. Furthermore, the results y_0, y_1, and y_2 will be ready in these cycles and they can be stored back to the data memory into the same locations ax_0, ax_1, and ax_2 from where they were loaded in cycles three,

four and five. In the last cycle of the evaluation of the butterfly, the last result, y_3, becomes ready and it can be stored back to the data memory.

5.1.3 Identification of Iteration Kernel

The radix-4 DIT FFT can be computed with the aid of two nested loops as was already explained in Chapter 4. The computation stages are iterated in the outer of these loops, and the butterflies of a single computation stage are evaluated inside the inner loop. To be able to compute the FFT effectively, the structure of the inner loop should be such that the utilisations of operations accessing data memory, i.e., LD and ST, would be as high as possible. In practice, if loads of both an input operand of a butterfly and a twiddle factor, and a store of a single result would be triggered simultaneously in as many clock cycles as possible, the accessing of data memory would be optimal. In other words, the inner loop should be scheduled in such a way that two loads and one store take place simultaneously in as many instructions as possible.

To obtain this situation, the operations of the radix-4 DIT FFT were scheduled manually a little bit further with the aid of same kind of a chart as in the case of a single radix-4 DIT butterfly. This new chart was quite easy to implement with the aid of the chart of Appendix D, thus the schedule of Appendix D should only be copied a couple of times successively. It can be found from Appendix E. In this chart, the schedules of different butterflies are highlighted with different colors, i.e., with green, red, yellow etc., and the data items moving on the buses are also marked with text inside the boxes.

From the chart of Appendix E, it can be seen that starting from clock cycle 13 two load operations and one store operation can be triggered simultaneously. In this clock cycle, the first result, y_0, is stored back to the data memory. Thus, the utilisations of LD and ST operations are optimal after cycle 13. Some regularities can be searched from the schedule after this cycle. If the same operations can be repeated from one clock cycle to another, the inner loop can be implemented with the aid of quite a clever manner. In this manner, the operations that take place before this regularity, which is henceforth to be called the *Kernel*, are executed first and then this kernel can be looped until the evaluation of the last butterfly of the stage is approaching. After an appropriate number of executions of the kernel, the looping of the kernel can be stopped and the operations that still have to be executed after the kernel can be performed.

It can be seen by examining Appendix E that the kernel really exists. The chosen kernel to be implemented is framed twice by the black rectangle in Appendix E. The

INSTR.	bus0	bus1	bus2	bus3	bus4	bus5	bus6	bus7	bus8
1	ADD1.r ->AG.t	ADD1.r --> ADD1.o	RF.one --> ADD1.t	AG.r1 --> LD1.t	AG.r1 --> RF	RF.four --> ADD2.t	ADD2.r --> ADD2.o	ADD2.r --> LD2.t	LD1.r --> CMUL.o
2	ADD1.r ->AG.t	ADD1.r --> ADD1.o	RF.one --> ADD1.t	AG.r1 --> LD1.t	AG.r1 --> RF	RF.four --> ADD2.t	ADD2.r --> ADD2.o	ADD2.r --> LD2.t	LD1.r --> CMUL.o
3	ADD1.r ->AG.t	ADD1.r --> ADD1.o	RF.one --> ADD1.t	AG.r1 --> LD1.t	AG.r1 --> RF	RF.four --> ADD2.t	ADD2.r --> ADD2.o	ADD2.r --> LD2.t	LD1.r --> RF
4	ADD1.r ->AG.t	ADD1.r --> ADD1.o	RF.one --> ADD1.t	AG.r1 --> LD1.t	AG.r1 --> RF	RF.four --> ADD2.t	ADD2.r --> ADD2.o	NOP	LD1.r --> CMUL.o

INSTR.	bus9	bus10	bus11	bus12	bus13	bus14	bus15	bus16
1	LD2.r --> CMUL.t	CMUL.r --> CADD.o3	RF --> CADD.o0	RF.tmp1 --> CADD.o1	RF.tmp2 --> CADD.o2	RF.zero --> CADD.t	CADD.r --> ST.t	RF --> ST.o
2	LD2.r --> CMUL.t	NOP	NOP	NOP	NOP	RF.one --> CADD.t	CADD.r --> ST.t	RF --> ST.o
3	NOP	CMUL.r --> RF.tmp1	NOP	NOP	NOP	RF.two --> CADD.t	CADD.r --> ST.t	RF --> ST.o
4	LD2.r --> CMUL.t	CMUL.r --> RF.tmp2	NOP	NOP	NOP	RF.three --> CADD.t	CADD.r --> ST.t	RF --> ST.o

Figure 18. *The instructions of the kernel.*

four instructions of the kernel are illustrated in Fig. 18.

5.2 Assembly Coding Process

In this section, the coding process for producing the parallel TTA assembly code, that is implementing the scheduling method of the previous section, is explained in detail. First, the essential resources, on which the implemented assembly code is based on, are specified in Subsection 5.2.1. Then the structure and the functionality of the implemented assembly code is described in Subsection 5.2.2. The manual optimization of the connectivity of the processor can be done with the aid of the implemented assembly code. This is descriped in Subsection 5.2.3.

5.2.1 Resources

It is fairly easy to figure out the essential resources of the processor to be programmed, such as FUs, RFs, buses, and sockets, by getting of the ground from the schedule chart presented in Appendix E.

Firstly, two addition operations are simultaneously triggered in cycle 16 on the buses two and five so that two FUs supporting the ADD operation are needed. Secondly, two load and one store operations are simultaneously triggered on the buses 3, 7, and 16 indicating that three LSUs are needed for accessing the data memory. Furthermore, a dedicated FU for each of the three user-defined operations (AG, CMUL, CADD) and two FUs supporting comparison operations EQ and GTU are needed. One of these comparator FUs is used in evaluating conditional moves that contain jumps, and the other in comparisons that chase up the current stage of the FFT computation so that

Table 5. *Characteristics of the FUs.*

FU	Latency	Supported Operations	Purpose of Use
FU1	2	AG	Generate **address of operand**
FU2	3	CMUL	**Complex multiplication**
FU3	1	CADD	**Complex addition**
FU4	1	ADD, SUB	Generate **linear index**
FU5	1	ADD, SUB	Generate **address of factor**
FU6	3	LD, ST	Load **operand**
FU7	3	LD, ST	Load **factor**
FU8	3	LD, ST	Store **result**
FU9	1	EQ, GTU	Evaluation of **stage**
FU10	1	EQ, GTU	**Conditional jumps**

the data memory area, from which the input operands are loaded, can be decided. The essential characteristics of the FUs of the processor are shown in Table 5.

Thirdly, by examining Appendix E, it can be seen that a total of 10 memory addresses, i.e., the addresses $ax0 - ax9$, have to be stored temporarily into the RF on the bus four in cycles 3-12 to be able to store the results $y_0 - y_9$ back to the data memory later in cycles 13-22 on the buses 15 and 16. Therefore,

1. 10 registers are needed for the temporary storage of the memory addresses, ax_k, that are generated in cycle t and used for the last time in cycle $t + 10$.

2. After cycle 12, the situation is always such that one address, ax_k, is generated by the address generator and one address, ax_{k-10}, is used by the LSU for the last time and can, therefore, be removed.

Due to these two reasons, an address FIFO containing 10 registers had to be implemented for the temporary storage of the adresses $\{ax_{k-9}, ax_{k-8}, ax_{k-7}, ax_{k-6}, ax_{k-5}, ax_{k-4}, ax_{k-3}, ax_{k-2}, ax_{k-1}, ax_k\}$ using 10 separate RFs as illustrated in Fig. 19(a). Respectively, a same kind of FIFO is needed for storing the input operands, x_{4k}, that must not be multiplied in the complex multiplier as every fourth twiddle factor, W_{4k}, equals one. The size of this other FIFO has to be two registers. This can be seen by examining the contents of buses 8 and 11 in cycles 6, 10, 12, and 14 in Appendix E. Registers **r11** and **r12** are reserved for this other FIFO. In addition, a total of 11 integer and 2 boolean registers are needed for different storage purposes, i.e., there is a total of 23 registers in the integer RFs and,

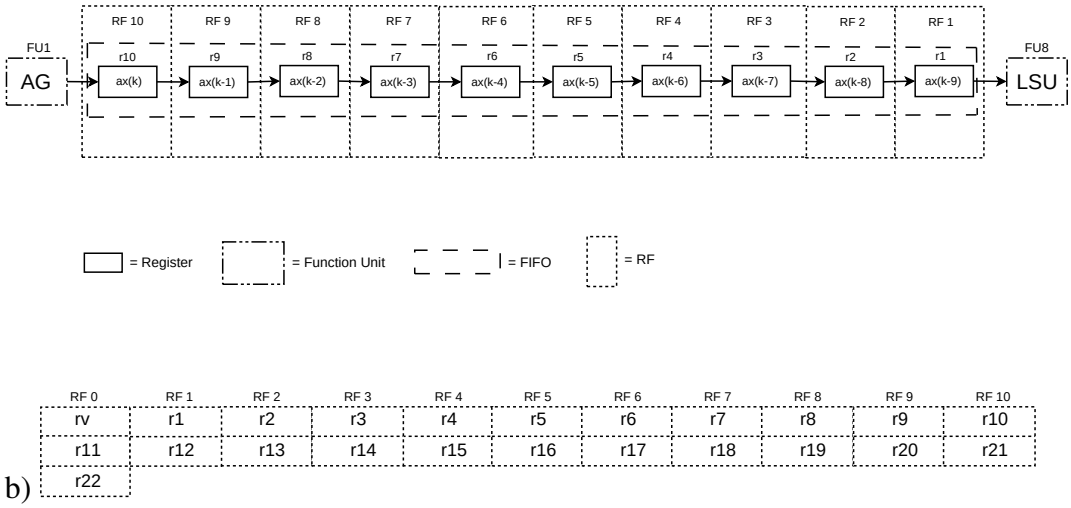


Figure 19. *Aspects of the integer RFs: a) register FIFO for addresses and b) locations of the registers in the integer RFs.*

respectively, two for the boolean values. There are 11 integer RFs in the processor in total as illustrated in Fig. 19(b). In addition, there is one boolean RF in the processor containing those two boolean registers. The usage of integer and boolean registers is shown in Table 8.

The number of buses can easily be decided by examining the discovered kernel and the address FIFO which are illustrated, respectively, in Fig. 18 and 19(a). Firstly, it can be seen from Fig. 18 that there exists a total of 17 buses in the kernel. However, one additional bus is needed for the move operations of the kernel since the load of the input operand of a butterfly is performed conditionally either from the input or the output buffer. In the first stage of the FFT computation, the input permutation is always performed by loading the operands of a butterfly from the input buffer according to the permutation principle explained in Chapter 3. Then the butterfly is evaluated and the results of the evaluation are stored to the output buffer. In the other stages, the *in-place* computation can be exploited by loading the operands and storing the results of the evaluation of a butterfly into same locations of the output buffer. That is, a total of 18 buses must be reserved for the kernel.

Secondly, the address transports from one register to another in the address FIFO can only be performed via sockets and buses since this FIFO is implemented with the aid of RFs as illustrated in Fig. 19(a). Therefore, a total of nine buses is needed for moving the address FIFO one step forward in every clock cycle that takes place after the FIFO has been once filled up with addresses. Respectively, one bus is required for moving the FIFO for input operands one step forward.

Table 6. *Characteristic of the buses.*

Number of Buses	Width	Purpose of Use
18	32	**Kernel operations**
9	32	Move **Address FIFO** forward
1	32	Move **Operand FIFO** forward
2	1	Transport **results of comparators**
1	32	Transport **Long immediates**
TOTAL 31		

Thirdly, a 1-bit bus is needed for both of the comparators to be able to move the 1-bit result of the comparator into the boolean RF. Furthermore, one bus is dedicated for moving long immediates. Moving of short immediates is not supported on any of the buses as the bits reserved for short immediates would increase the size of the instruction. The essential characteristics of the needed buses is summarized in Table 6.

Now the needed FUs, RFs, and buses of the processor have been specified. In addition to the buses, the sockets are also needed by the interconnection network of the processor to be able to transport data between FUs and RFs. Data transports are made possible by connecting the sockets to the buses. The operand and trigger registers of the FUs must be connected to corresponding input sockets and the result registers of the FUs to corresponding output sockets. Also the RFs have to be connected to input and output sockets so that, respectively, a write and a read of a register would be possible. The types of the sockets are tabulated and the total number of them shown in Table 7.

Table 7. *Types of the sockets.*

Type of Socket	Number of Sockets	Purpose of Use
FU input	**26**	Transport data from bus into FU
FU output	**12**	Transport data from FU onto bus
RF input	**15**	Transport data from bus into register
RF output	**20**	Transport data from register onto bus
PC input	**1**	Transport data from bus into PC
IMM output	**1**	Transport data from immediate register onto bus
	TOTAL 75	

Table 8. *Usage of registers.*

	INTEGER RFs
Registers	**Purpose of Use**
rv	**Four** register
r1-r10	**Address FIFO**
r11, r12	**FIFO for input operands**
r13	Store the **2nd complex product** of a butterfly.
r14	Store the **3rd complex product** of a butterfly.
r15	**Stage** register
r16	**Zero** register
r17	**One** register
r18	**Two** register
r19	**Three** register
r20	Store **Base address of input buffer**
r21	Store **Base address of factor buffer**
r22	Store **Base address of output buffer**
	BOOLEAN RF
b0	Select **input/output buffer** for the operand load
b1	Evaluate **conditional jumps**

5.2.2 Code

The parallel TTA assembly code is composed of six basic blocks whose purpose is discussed in the following paragraphs. The code can be found from Appendix F and the structure of the code is depicted in Fig. 20. The registers used by the code can be found from Table 8.

There are 10 instructions in basic block 0 which performs initialisations. In instructions 0-8, the needed long immediates are transported to the registers from the immediate unit on bus 31 which is dedicated for transporting long immediates. In instruction 9, the address generators are initialized with the base addresses of the memory buffers and the last index of the kernel (1013) is transported to a temporary register from which it can be read when the end-condition of the kernel is being evaluated.

Basic block 1 is the first block of the outer loop containing two instructions. The evaluation of a computation stage is always started in basic block 1 by updating the stage into the address generator. In addition, the stage is compared with zero so that the correct data memory area, from which the input operands are loaded, can be selected with the aid of the boolean register b0. If the stage equals zero, this register is true,

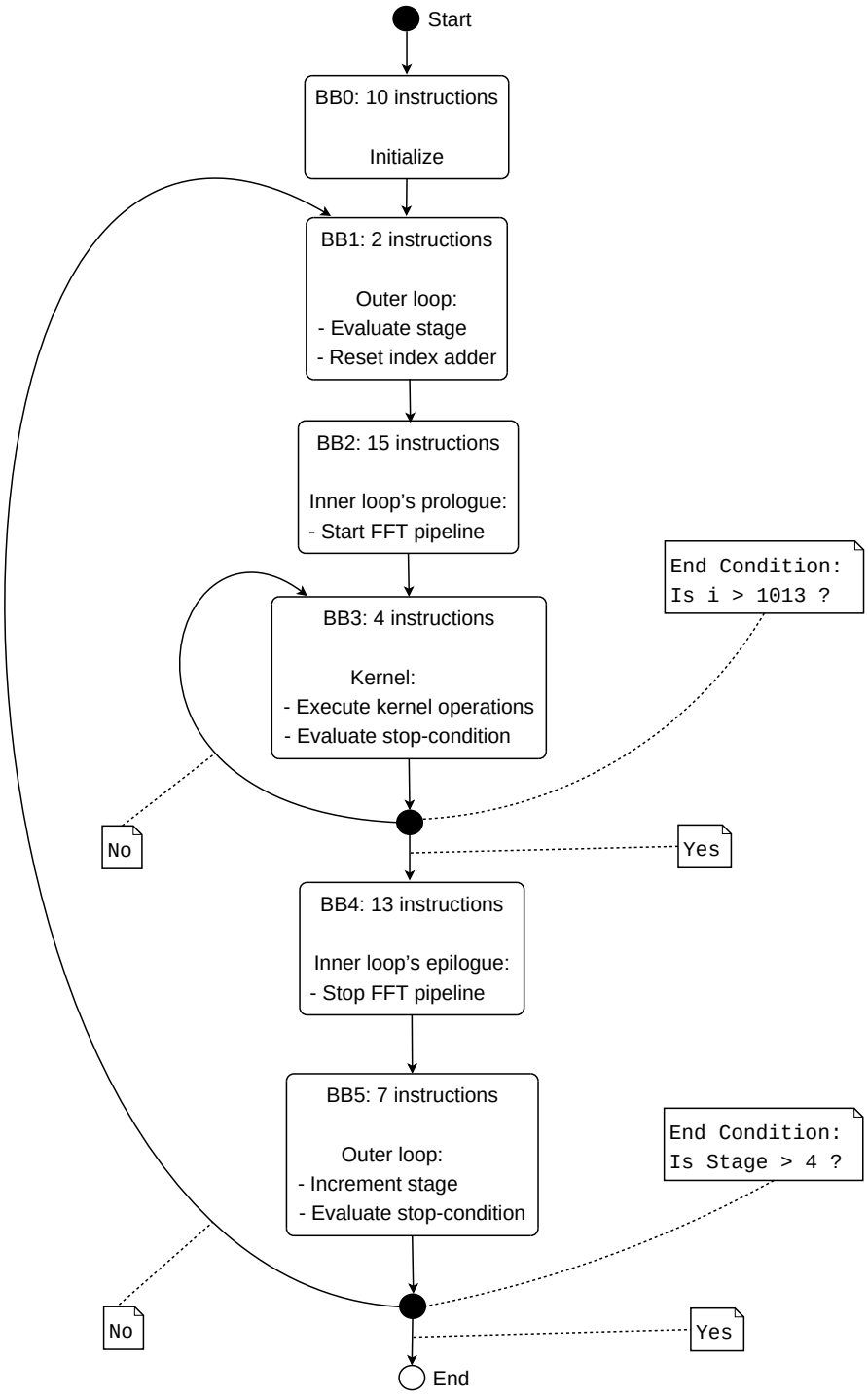


Figure 20. *Flowchart illustrating the structure and the control flow of the assembly code.*

i.e., its value is one, and the input operands must be loaded from the input buffer in the instructions 14-34. I.e., the load move of bus six (m6) is always executed in the stage zero in these instructions. In the other stages, the boolean register b0 is false and the input operands are loaded normally from the output buffer meaning that the load move of bus seven (m7) is executed. The index adder (fu4) must also be cleared in

Table 9. *Essential operation sequences of the inner loop (basic blocks 2, 3, and 4).*

Operation Sequence	**Occurs on Move Buses**
Conditional **jump**	**m1, m2, m31**
Generation of **linear index**	**m3, m5**
Generation of **operand address**	**m4**
Load of **operand**	**m6, m7**
Adding **address to FIFO**	**m8**
Complex Multiplication	**m9, m10**
Generation of **factor address**	**m11, m13**
Load of **factor**	**m12**
Complex Addition	**m14, m15, m27, m29, m30**
Store of **result**	**m16, m17**
Moving **Address FIFO** one step forward	**m18-m26**
Moving **Operand FIFO** one step forward	**m28**

instruction 11 so that the generation of linear indices can be restarted.

The inner loop is composed of basic blocks 2, 3, and 4. The functionality of the inner loop is discussed in the following four paragraphs. The buses, on which the essential operation sequences of the inner loop are performed, are tabulated in Table 9 to ease the following of the code.

Basic block 2 operates as the prologue of the inner loop. Its purpose is to execute the operations that have to be performed to get the performance-critical resources, such as LSUs, fully utilized. The operations that occur before the kernel, i.e., the operations from cycles 1-15 in Appendix E, are, therefore, executed in the 15 instructions of the basic block 2. In addition to operations presented in Appendix E, also the FIFOs for operands and addresses have to be forwarded one step, respectively, in instruction 23 on bus 28, and in instructions 24-26 on buses 18-26. In instruction 23, the first complex addition is triggered on bus 15 meaning that the the first operand, which has been stored temporarily to the operand FIFO, must be transported from the operand FIFO to the complex adder on bus 27. Correspondingly, the second temporarily stored operand must be transported to the register r11 in the operand FIFO. This must be done since the temporarily stored operands are always moved from the register r11 to the complex adder. In instructions 24-26, respectively, the first, the second, and the third result are stored back to the data memory on buses 16 and 17 and, therefore, the addresses have to be moved forward in the address FIFO as illustrated in Fig. 19(a).

The kernel is implemented in the four instructions of the basic block 3. These instruc-

tions implement the kernel operations presented in Fig. 18. Similarly as in the case of the basic block 2, the address FIFO has to be further stepped forward in each of the instructions since one result is stored back to the data memory and one new address is generated in every clock cycle. Meanwhile, the operand FIFO has to step forward only in every 4th clock cycle, i.e., this takes place in instruction 27 on bus 28. The end-condition of the kernel, which is illustrated in Fig. 20, is evaluated in instructions 27, 29, and 30. When the execution of the program comes to the kernel for the first time inside a computation stage, the conditional jump to instruction 27 on bus 31 is always executed since the boolean register b1 has been set to zero in instruction 25 on bus 2. Since the jump latency is four cycles, this means that the execution of the program comes again to instruction 27 after the instructions 28, 29, and 30 have been executed. Then the end-condition is evaluated in instructions 29 and 30 by comparing the current linear index with value 1013. If the current linear index is greater than 1013, b1 is set to one in instruction 30 on bus 2 and the conditional jump in instruction 27 on bus 31 is no longer executed when the execution of the program moves on to basic block 4.

The last index of the kernel equals 1013 thus the kernel must be iterated 252 times in a computation stage to get all the butterflies evaluated. Then it can be concluded by examining Appendix E that the linear index equals $15 + 251 \cdot 4 = 1019$ in instruction 27 and 1021 in instruction 29 as the kernel is being executed for the last time in a computation stage. Because the comparison takes place in instruction 29, index $1021 - 4 = 1017$ is the first one, which gives true as a result of GTU comparison meaning that the jump condition is false and just the indices 1019, 1020, 1021, and 1022 are the last ones in the kernel to be transported to the address generator.

Basic block 4 operates as the epilogue of the inner loop. There are 13 instructions in the basic block 4 which execute the operations that remain after the kernel has been iterated 252 times. In this phase of the evaluation of the computation stage, one address of an operand has to be still generated by triggering the address generator with the last linear index(1023) in the instruction 31 on bus 4. In addition, the following **operations** have to be executed inside the basic block 4:

- **Load 3 operands** from the data memory in instructions 31, 32, and 33.
- **Perform 5 complex multiplications** in instructions 31, 32, 34, 35, and 36.
- **Generate 3 addresses of twiddle factor** in instructions 31, 32, and 33.
- **Load 3 factors** from the data memory in instructions 31, 32, and 33.

- **Perform 12 complex additions** in instructions 31-42.
- **Store 13 results** back to the data memory in instructions 31-43.
- Move the **address FIFO 12 steps forward** in instructions 31-42.
- Move the **operand FIFO 3 steps forward** in instructions 31, 35, and 39.

The last block of the code, basic block five, is the second code block that is keeping the outer loop up. In the seven instructions of this block, the stage is incremented by one and the end-condition of the computation of the 1024-point radix-4 DIT FFT is evaluated as illustrated in Fig. 20.

5.2.3 Connectivity Optimization

There are 31 buses and 75 sockets in total in the IC of the processor as can be seen from Tables 6 and 7. The numbers of buses and sockets are quite high, which can easily increase the costs and the complexity of the IC. In addition, the length of the instruction is increased which further increases the size of the program code. The larger code, the larger instruction memory is required which increases both chip area and power consumption. The key issue to avoid these problems is to optimize the connectivity of the IC to a minimum. This ensures the fact that sockets can be implemented with the aid of simple multiplexors and demultiplexors, which is decreasing the implementation costs of the entire processor remarkably. On the other hand, the programmability of the processor is always reduced by minimizing the connectivity. This is, however, not a problem since the purpose of this assembler implementation is to be as optimal as possible also in terms of processor cost. It is enough from the programmability's point of view that the processor is just capable of performing the assembly code presented in Appendix F.

If the IC was fully connected, there would exist a total of $31 \cdot 75 = 2325$ connections in the IC. Let us figure out what is the minimum number of connections for performing the FFT application presented in Appendix F. The minimal connectivity can easily be obtained by examining the assembly code. The sockets that must be connected to a bus can be seen by going through all the data moves occurring on the bus. Then the sockets can be seen by examining the socket fields of the data moves. A couple of examples of specifying the connections of a couple of buses are shown in Table 10.

The connections, which are surrounded by curly brackets in the 'Sockets'-section of Appendix G, are obtained by traversing through all the buses in the manner illustrated

Table 10. *Examples of specifying the connections.*

Bus	Moves	Attached Sockets
m1	fu9.gtu_r → b0 [m1/-/fu9_r/b_i1]	**fu9_r, b_i1**
m5	r15 → fu9.eq_t [m5/-/ri_o5/fu9_t]	**ri_o5, fu9_t,**
	r16 → fu4.add_t [m5/-/ri_o6/fu4_t]	**ri_o6, fu4_t,**
	r17 → fu4.add_t [m5/-/ri_o7/fu4_t]	**ri_o7**

in Table 10. The connectivity of the processor is, therefore, as minimal as possible. The number of connections equals 119 which is only about 5% of the full connectivity. This means that major savings in needed chip area will be obtained. It can be also stated, that now the architecture of the core of the processor is fully characterized. The core of the processor is illustrated in Appendices H and I. The FUs with their IC are presented in Appendix H and RFs, respectively, in Appendix I. The structure of the core can be examined by concatenating these two appendices. Next, it can be moved on to the HW aspects of the processor.

5.3 Hardware Implementation

The HW-implementation of the processor is discussed briefly in this section. First, the memories interacting with the core are discussed in Subsection 5.3.1. After that the functionality of the core is described in Subsection 5.3.2. Lastly, the HDL-simulations and the synthesis of the processor are described briefly in Subsection 5.3.3.

5.3.1 Memories

The block diagram of the processor is presented in Fig. 21. In this block diagram, the arrows illustrate only the data flow between different blocks. In reality, the buses and the sockets always exist between the FUs and the RFs when data transports are being performed in the processor as illustrated in Appendices H and I. As can be seen from the block diagram, there are two separate data memories and an instruction memory (IMEM) in the processor.

The size of the dual-port data memory (DP-DMEM) is 2048 words and it is used for lodging the input and the output data of the 1024-point FFT computation. The 1024 complex inputs are lodged in the beginning of this memory buffer from where they are always read by FU6 during the first computation stage. The results of the

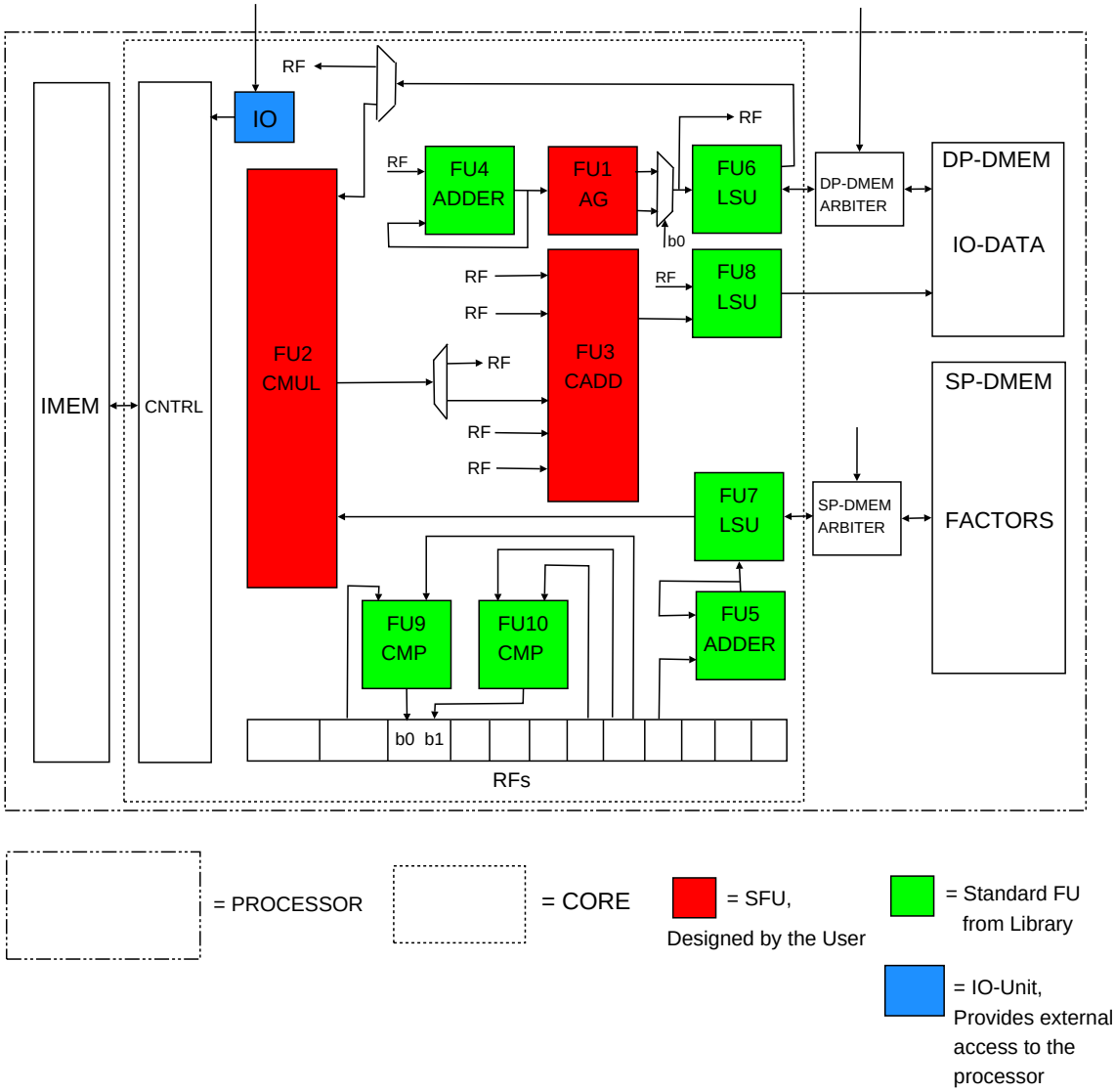


Figure 21. *Block diagram of the processor.*

butterfly evaluations are always stored by FU8 into the output-section which is locating in the addresses 1024-2048 of the DP-DMEM. The other data memory is a single-ported memory (SP-DMEM) which is used for lodging the 5120 twiddle factors. This memory is connected to the FU7 which is used to load the factors into the complex multiplier (FU2).

The IMEM contains the binary representations of the 51 instructions of the program code presented in Appendix F. These 51 instruction bit vectors can be generated by using the TTA assembler discussed in Chapter 2 on page 23. Each instruction consists of 31 slots, i.e., there is a dedicated slot for each of the 31 buses. The size of the instruction is 256 bits which can be seen by examining the statistic provided by the TTA assembler. The instructions are fetched from the IMEM by the IFU of the control

Table 11. *Characteristics of the instruction and data memories of the processor.*

Memory	Width [bits]	Address Space	Size [bits]	Usage
IMEM [51 · 256 bits]	256	0-50	13056	**instruction bit vectors**
DP-DMEM [2048 · 32 bits]	32	0-2047	65536	**input/output data** (addresses **0-1023/1024-2047**)
SP-DMEM [5120 · 32 bits]	32	0-5119	163840	**twiddle factors**

unit and decoded by the IDU. Thus, the IDU decodes the slots of the instruction and activates the correct sockets via which the data transports are going to take place. The essential characteristics of the memories of the processor are shown in Table 11.

5.3.2 Core

The core of the processor is surrounded by the dashed rectangle in the block diagram of Fig. 21. As can be seen from the block diagram, there are three SFUs and seven standard FUs in the core. The HW-implementations of the SFUs were already discussed in Chapter 4. There is no need to know the internal HW-implementation of a standard FU or a standard RF by the designer since the HDL-models of those can be found from the HW block library. The HW-subsystem of the MOVE framework is also capable of generating the HDL-models of the control unit and the IC as was described in Chapter 2. Thus, there is no need to know the internal implementations of those either, i.e., the designer must be familiar only with the external interface of the core of the processor. These issues quicken the design process and ease the work of the designer remarkably.

As can be seen from Fig. 21, FU4 has been connected to FU1. FU4 is the index adder that is generating linear indices for FU1, the address generator. FU4 operates as an incrementer, i.e., the previous result is always moved from the result register to the operand register. FU4 is triggered by moving the value to be incremented, i.e., one, from RF to the trigger register. FU1 is generating addresses of operands for FU6 that is loading the operands from the DP-DMEM. As explained already before, FU1 has always two outputs: one is an address of the input section of the DP-DMEM and the other is, respectively, an address of the output section. The section (input/output) from where the operands are loaded in different computation stages is selected by using the

boolean register b0. Thus, b0 is controlling the mux that selects the correct address for the FU6 in different computation stages. The boolean register b0 is always updated in the beginning of a computation stage by the comparator-unit FU9. In the first stage, zero, an address of the input-section need to be selected since the input permutation is performed in this stage. In other stages(1, 2, 3, and 4), an address of the output-section is selected since the *in-place* computation can be exploited. In addition, the selected address must be always stored into the address FIFO which is implemented by using the RFs.

The operands are moved from the FU6 either to the RF or to the complex multiplier (FU2). Every fourth twiddle factor is one and there is no need to multiply the corresponding operand with it. Thus, every fourth operand is moved to the operand FIFO, which is also implemented by using the RFs, for waiting the complex addition to take place. The twiddle factors are moved to the complex multiplier via FU7 which is loading the factors from the SP-DMEM. The linear addresses of the factors are generated by FU5 which is operating equally as the FU4, except that the FU5 increments always its value by four. The results of complex multiplications are transported either via the RF or straightly to the complex adder (FU3) as can be seen from Fig. 21. The second and the third result of complex multiplications of a butterfly have to be always stored temporarily into the RF before these results can be transported to the complex adder as was already explained in Subsection 5.1.2.

The results of complex additions are stored back to the output-section of the DP-DMEM by using the FU8. These results are stored just into same locations as from where the input operands were read from by the FU6. Thus, the store addresses can be read from the address FIFO to the LSU. The comparator-unit FU10 together with the boolean register b1 are used for evaluating conditional jumps that occur in the processor.

The external communication with the processor can be done by using the IO-unit which has been marked with blue color in Fig. 21. E.g., external interrupts can be generated and the processor halted with it. The essential registers of the control unit, such as the PC, can also be initialized by the IO-unit. The changing of the contents of the data memories can also be performed by using the IO-unit, i.e., the processor is halted and the new contents are written to the data memories via the memory arbiters. Especially, the writing of new data to the DP-DMEM would be a quite often executed operation in a real DSP-system where the TTA processor might be, e.g., a slave processor. In this case, new data would be written to the DP-DMEM always when the host-processor

would like to perform a new 1024-point FFT transform.

5.3.3 HDL Simulation and Synthesis

To be able to verify the correct functionality of the 1024-point radix-4 DIT FFT presented in Appendix F, a VHDL-model of the processor was build for the simulation and the synthesis by using the processor generator `MOVEgen` [9].

The VHDL-description of the core could be generated fairly easily as described in [9]. Only the VHDL-descriptions of the SFUs had to be manually implemented. Next, the data and instruction memories, and the data memory arbiters were connected to the core by exploiting the pre-designed components of MOVEgen. Then a simple, pre-designed testbench was built surround the processor.

First, the register-transfer-level(RTL) simulation was run in the ModelSim's HDL-simulator [10]. The correct functionality of the processor could be verified as follows:

1. The contents of the buses was examined in different clock cycles and compared with the structure and the known, correct operationality of the assembly code presented in Appendix F.

2. The contents of the DP-DMEM's output-section was compared with the known, correct output contents of a HLL-implementation whose correct functionality was verified by comparing to the reference implementation done with MATLAB.

After the RTL simulation, the processor was synthesized to the gate-level. The gate-level simulation was run next for obtaining the utilizations of the gates. These were needed in the power synthesis which was also done for getting information about its power consumption. The results of the synthesis are discussed in the following chapter where the performance of the FFT-on-TTA (FFTTA) processor is compared with a couple of other FFT processors.

6. PERFORMANCE ANALYSIS

There exists a considerable number of commercial and academic FFT processors in the market. Designers have developed processors from different bases meaning that the characteristics of the processors vary widely. User can choose, e.g., a low-power or a high-speed processor according to his own requirements. The implementation techniques vary also; one can choose an ASIC or a programmable DSP. The best possible cost-performance characteristics can be obtained by using ASICs but on the other hand, DSPs offer programmability and almost the same cost-performance characteristics can be obtained by using them.

In this chapter, the essential characteristics of the FFTTA processor are summarized. In addition, the FFTTA processor is compared against other commercial and academic FFT processors. The characteristics are summarized in Section 6.1 and the comparison is presented in Section 6.2.

6.1 Proposed FFT Processor

The essential characteristics of the proposed FFTTA processor are described in this section. The computation speed of the processor is discussed briefly in Subsection 6.1.1. Chip area and power consumption are listed and discussed in Subsection 6.1.2.

6.1.1 Computation Speed

It can be seen from the synthesis results that the FFTTA processor can easily operate with the clock frequency of 250 MHz. The critical path lies in the control unit and it is 3.2 ns. Thus, 300 MHz is the absolute maximum clock frequency on which the processor can run.

The performance of the 1024-point radix-4 DIT FFT computation in terms of elapsed clock cycles can easily be calculated by examining the execution frequencies of the

Table 12. *Performance of the 1024-point radix-4 DIT FFT in terms of elapsed clock cycles.*

Basic Block	Execution Frequency	Number of Instructions	Elapsed Cycles
BB0	1	10	**10**
BB1	5	2	**10**
BB2	5	15	**75**
BB3	$252 \cdot 5 = 1260$	4	**5040**
BB4	5	13	**65**
BB5	5	7	**35**
			TOTAL 5235

basic blocks from Appendix F. The clock cycles elapsed by a single basic block can be obtained by multiplying the execution frequency of the basic block with the number of instructions as shown in Table 12. The execution frequency of the kernel (basic block 3) is naturally the highest as it is being executed 252 times inside each computation stage.

Now the total number of elapsed clock cycles, 5235, is also close to the theoretical minimum, which can be obtained with this implementation technique. The theoretical minimum would be 5120 cycles since this implementation technique requires 5120 twiddle factors for computing the 1024-point transform. I.e., if one factor would be loaded from the data memory in every cycle, the theoretical minimum would be obtained. Now there is a small overhead of 115 cycles caused by the loops and initialisations.

Execution time elapsed in computing one 1024-point transform, t, can be calculated as $t = c/f$ where c is the number of elapsed clock cycles and f the used clock frequency. The execution time t equals 20.94 μs by placing $c = 5235$ and $f = 250$ MHz into this formula.

6.1.2 Chip Area and Power Consumption

Chip areas and power consumptions of the different components of the FFTTA processor can be seen from Table 13. Instruction memory is optimized by using the dictionary-based code compression approach [14]. In this approach, all the unique bit strings that occur in the code to be compressed are collected into a dictionary. The bit strings of the original program are then replaced with indices pointing to the dictionary. The size of the compressed instruction is $\lceil \log_2 D \rceil$ bits where D is the number of

Table 13. *Chip area and power consumption of the proposed FFTTA processor.*

Component	**Area [kgates]**	**Power [mW]**
Core	39.8	48.5
Compressed IMEM [51 · 6 bits]	0.9	1.4
DP-DMEM [2048 · 32 bits]	102.4	27.0
SP-DMEM [5120 · 32 bits]	78.0	8.4
	TOTAL 221.3	**TOTAL 85.3**

entries in the dictionary. [14]

The programmability of the FFTTA processor is already extremely low since its connectivity has been optimized to a minimum. Therefore, the code compression can also be implemented as optimally as possible. Since there are 51 instructions in the program code, the size of the compressed instruction becomes only 6 bits by placing the original 256-bit instruction into the dictionary. The dictionary can be implemented using a separate memory or standard cells. Using memory to implement the dictionary maintains the programmability but results in poor compression ratio, r. Thus, as the programmability is already low, it is more favorable to implement the dictionary using standard cells as it provides a significantly better compression ratio. This results from the fact that the synthesis tool can optimize the dictionary by removing the redundancy occurring in the bit strings stored to the dictionary. The compression ratio can be calculated as $r = (IMEM_c + CNTR_c)/(IMEM_o + CNTR_o)$ where $IMEM_c$ is the area of the compressed IMEM, $CNTR_c$ the area of the control unit using code compression, and $IMEM_o$ and $CNTR_o$ the areas of the uncompressed IMEM and control unit. By applying the previous formula, r equals only 10% for the IMEM of the FFTTA processor which can be considered as an extremely good result. The previous formula can also be applied for the power's compression ratio, p. Now the areas of the previous formula are just replaced with powers consumed by the IMEM and the control unit. Power's compression ratio p equals also 10% indicating that the compressed IMEM and the control unit together consume 90% less power than the uncompressed ones.

The data memory for the twiddle factors (SP-DMEM) could be optimized by removing the redundancy of the factors. Namely, instead of $5N$ factors, only $N/8+1$ factors are necessarily needed to compute the N-point radix-4 DIT FFT. I.e., in this case, only 129 factors could be used instead of 5120 factors. This optimisation could be implemented by using an address generator also for the factor-memory. Namely, a correct factor for each of the 5120 clock cycles can always be generated by performing simple modifications, such as multiplying with minus one or swapping the real and the

Table 14. *Estimated chip area and power consumption of the proposed processor if also the data memory for the factors (SP-DMEM) is optimized.*

Component	**Area [kgates]**	**Power [mW]**
Core	39.8	48.5
Compressed IMEM [51 · 6 bits]	0.9	1.4
DP-DMEM [2048 · 32 bits]	102.4	27.0
Optimized SP-DMEM [129 · 32 bits]	25.7	2.8
	TOTAL 168.8	**TOTAL 79.7**

imaginary parts, to the 129 factors, which are stored into the SP-DMEM. The address of the factor, from which the needed factor can be generated, as well as the modification to be performed to the factor can be computed in an SFU with the aid of the current stage and the current linear index.

Estimated area and power consumption of the proposed FFTTA processor can be seen from Table 14 in the case the factor memory would also be optimized in addition to the IMEM. The estimates for SP-DMEM's area and power have been obtained from the synthesis results by proporting the area and power of the unoptimized SP-DMEM to the size of the optimized SP-DMEM. The total area consumption should improve about 24% due to optimizing the SP-DMEM. The total power consumption is not reduced too much since the relative share of power due to SP-DMEM is small. The essential characteristics of the proposed FFTTA processor are summarized in Table 15.

Table 15. *Summary of the proposed FFTTA processor.*

FFT size (N)	1024
Elapsed clock cycles	**5235**
Word length	32
Algorithm	radix-4
Process [μm]	0.11
Voltage [V]	1.5
Clock frequency [MHz]	**250**
1024-pt. exec. time [μs]	**21**
Area [kgates]	**221.3**
Power [mW]	**85.3**
Code compression ratio [%]	**10**

6.2 Performance Comparison

It is not easy to make a fair comparison between different FFT processors since these processors are fabricated in different complementary metal oxide semiconductor (CMOS) technologies and the sizes of the FFT also vary. Thus, some kind of normalizations are needed to be able to make comparisons between FFT processors that are fabricated in different technologies.

Three normalizations can be used for comparisons as described in [15]. The *normalised area*, A_n, can be used to evaluate the cost of silicon. It is given by

$$A_n = \frac{A_{1024}}{(T/0.35\mu m)^2} \tag{20}$$

where A_{1024} is the area of the 1024-point FFT (mm^2) and T the used CMOS technology (μ). Since the layout of the proposed FFTTA processor with correct routings and placement of cells has not been implemented, it is difficult to accurately estimate the metric area of the FFTTA processor. Therefore, a more reliable index, the transistor count C, is used in the comparison. One equivalent gate, 2-NAND, can be implemented using four transistors. Thus, g 2-NAND gates require

$$C = 4g \tag{21}$$

transistors in total.

The transistor count of the FFTTA processor was compared to the best found ASIC implementation and a couple of other programmable DSPs. The results are shown in Table 16. The transistor count of the FFTTA is about 48% higher than that of the best found ASIC implementation. There is still quite a lot of overhead in the transistor count of the FFTTA Processor since the SP-DMEM has not been optimized. The transistor count should improve about 24% by optimizing the SP-DMEM. Meanwhile, the transistor count of the FFTTA processor is evidently better than the transistor counts of the DSPs.

Energy efficiency, *FFT/Energy*, gives an adjusted number of 1024-point complex FFTs that can be calculated for a fixed amount of energy. It is given by

$$FFT/Energy = \frac{T}{P_{1024} \cdot t} \tag{22}$$

where T is the technology (μ), P_{1024} the power consumed by the 1024-point FFT (mW), and t the execution time of the 1024-point FFT (μs). The energy efficiency of the FFTTA processor was compared with two good ASIC implementations and a

Table 16. *Comparison of transistor counts.*

Processor	**Transistor Count [ktransistors]**
ASIC	**598**
Lin, Tsai & Chiueh [15], Taiwan University	
Programmable DSP	**21 000**
Imagine [16], Stanford University	
HiPAR-DSP4 [16], Hannover University	**1 200**
FFTTA	**885**

couple of DSPs. Table 17 shows the results. The FFTTA processor can not compete against the best found ASIC implementation, i.e., the processor of [17], in energy efficiency since Spiffee's power consumption is so low and the implementation technology sparse. However, the FFTTA processor is better than the processor of [15] in energy efficiency since the power consumption of [15] is so much higher than that of the FFTTA processor. Furthermore, the FFTTA processor is much better than other

Table 17. *Comparison of energy efficiencies.*

Processor	**Tech. [μ]**	**Voltage [V]**	**Power [mW]**	**Exec. time [μs]**	**Clock rate [MHz]**	**FFT/energy [$\frac{10^{15}}{J}$]**
ASIC						
Spiffee I [17], Bevan Baas	0.6	1.1	9.5	330	16	**191.39**
Lin, Tsai & Chiueh [15]	0.35	3.3	480	22.5	45.45	**32.41**
Programmable DSP						
Imagine [16]	0.15	1.5	9000	20.6	180	**0.81**
C40 [16], Texas Instruments	0.7	5.0	4500	1298	60	**0.12**
HiPAR-DSP4 [16]	0.5		5000	222	66	**0.45**
FFTTA	**0.11**	**1.5**	**85.3**	**20.9**	**250**	**61.58**

Table 18. *Comparison of energy-time products.*

Processor	**Exec. time [μs]**	**FFT/energy**	**Energy x Time [$\frac{\mathbf{J \cdot s}}{\mathbf{10^{21}}}$]**
ASIC			
Lin, Tsai & Chiueh [15]	22.5	32.41	**0.69**
Spiffee I [17]	330	191.39	**1.72**
Programmable DSP			
Imagine [16]	20.6	0.81	**25.43**
C40 [16]	1298	0.12	**10816.67**
HiPAR-DSP4 [16]	222	0.45	**493.33**
FFTTA	**20.9**	**61.58**	**0.34**

programmable DSPs in energy efficiency as can be seen from Table 17.

A metric considering both energy efficiency and speed performance is the *energy-time product*, which is given by

$$Energy \times Time = \frac{t}{FFT/Energy} \tag{23}$$

The energy-time product of the FFTTA processor was compared equally as the energy efficiency. The results of the comparison can be seen from Table 18. It can be seen from Table 18 that the energy-time product of the FFTTA processor is evidently better than the energy-time products of the ASIC implementations. The FFTTA processor is better than the processor of [15] since the energy efficiency of the FFTTA processor is much better than that of the processor in [15]. Naturally, one has to remember that there exists inaccurary in the power consumption of the FFTTA processor since it has been obtained as a result of the power synthesis. The power synthesis does not take into account, e.g., the leakage currents of the memories which yields to a too good estimate of the power consumption. Thus, in reality, the energy efficiency of the FFTTA processor would be somewhat lower yielding to a higher energy-time product. Meanwhile, the FFTTA processor beats the processor of [17] in energy-time product due to its slow execution time.

7. CONCLUSIONS

In this thesis, FFT was implemented on TTA to evaluate TTA's performance for performing FFT. Firstly, background information of the TTA concept and the MOVE framework, the semi-automated design environment for designing TTA processors, was given. Secondly, the needed FFT theory was explained. In the first implementation phase, FFT was implemented using HLL code and the HLL compiler of the MOVE framework. Since the performances of the HLL implementations were unfavourable, FFT was implemented manually by hand using assembly code in the second implementation phase. Finally, based on the assembler implementation, the effective FFTTA processor was proposed and compared with other commercial and academic FFT processors to see TTA's performance against other FFT processors.

The performances of the HLL implementations remained at the low level in terms of elapsed clock cycles. This results from the fact that there are defects in the HLL compiler of the MOVE framework. Above all, the support for software pipelining would be needed to exploit the available ILP better. Naturally, the performance of the assembler implementation can never be reached by using the HLL compiler but considerably better performance, which would be sufficient for the needs of most of the customers, could be obtained with the aid of software pipelining.

Meanwhile, the performance of the assembler implementation was found out to be extremely good and the utilisations of the essential resources, such as SFUs and LSUs, were extremely high. The chip area was much better than that of the other DSPs and it was even quite close to the chip areas of the ASIC implementations that are totally unprogrammable. There still exists quite a lot of overhead in the chip area. Namely, if the amount of the data memory lodging the twiddle factors would be optimized from the current $Nlog_4N$ to the lowest possible $N/8$, the area would decrease considerably, i.e., about 24%. Furthermore, the power consumption would decrease slightly but not so much as the area since the SP-DMEM consumes much less power than the DP-DMEM. The optimisation of the SP-DMEM can be implemented without a considerable loss of computation speed meaning that the number of elapsed clock cycles

should not increase more than a couple of hundreds of clock cycles at the maximum. The optimisation of the SP-DMEM was left as future work. Promising numbers were also obtained for the energy efficiency and the energy-time product but one has to remember that there exists some inaccuracy in the power consumption of the FFTTA processor which may yield to too optimistical results.

The original 256-bit instruction word could be reduced down to only six bits by using the dictionary-based code compression. This proves that TTA's long instruction word does not cause a big problem if the code is compressed properly. Code compression decreases also power consumption. In this case, the power consumption was reduced as much as 34%.

The assembler implementation supports only the 1024-point transform at the moment. However, certain other transform sizes could be supported quite easily. The support for the sizes $N = 4^k$ could be implemented by using the same prologue and the kernel. Only the number of iterations of the kernel and the code of the epilogue should be adjusted according to the supported size N. Also the support for the transform sizes N consisting of mixed radix-4 and radix-2 processing columns would be an easy extension. Only the complex adder should be extended to evaluate the radix-2 butterfly in addition to the radix-4 butterfly. Furthermore, an additional loop evaluating the radix-2 butterflies should be added to the end of the current assembly code. The expandability for several transform sizes is of great importance since several FFT sizes are usually used in DSP applications.

Performance of TTA in performing FFT was found out to be extremely good when the code was programmed using the assembler. The HLL implementations could not compete against the assembler implementation in performance which was an expected result. As a conclusion, based on the results obtained in this thesis work, it can be stated that TTA is a promising programmable architecture candidate for implementing DSP applications.

BIBLIOGRAPHY

[1] *The MOVE Framework User's Manual*, Tampere University of Technology, 2004.

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

[3] H. Corporaal, "Transport Triggered Architectures: Design and Evaluation," Ph.D. dissertation, Delft Univ. Tech., Sept. 1995.

[4] R. P. Colwell, R. P. Nix, J. J. O'Connel, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE T. Comput.*, vol. 37, no. 8, pp. 679–967, Aug. 1988.

[5] M. Niiranen, "Transport Triggered Architectures of Field Programmable Gate Array," Master's thesis, Tampere Univ. Tech., Tampere, Finland, Oct. 2004.

[6] T. Rantanen, "Cost Estimation for Transport Triggered Architectures," Master's thesis, Tampere Univ. Tech., Tampere, Finland, May 2004.

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

[8] J. Sertamo, "Processor Generator for Transport Triggered Architectures," Master's thesis, Tampere Univ. Tech., Tampere, Finland, Sept. 2003.

[9] *MOVEgen User's Manual*, Tampere University of Technology, 2004.

[10] *ModelSim SE User's Manual*, Model Technology, 2003.

[11] J. Takala, "Real-Time Signal Processing Systems: Parallel Algorithms and Architectures," Ph.D. dissertation, Tampere Univ. Tech., 1999.

[12] J. Granata, M. Conner, and R. Tolimieri, "Recursive fast algorithms and the role of the tensor product," *IEEE T. Signal Process.*, vol. 40, no. 12, pp. 2921–2930, Dec. 1992.

[13] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ, U.S.A.: Prentice Hall, 1975.

[14] J. Heikkinen, J. Takala, and H. Corporaal, "Dictionary-based program compression on transport triggered architectures," in *Proc. IEEE Int. Symp. on Circuits and Systems*, May 2005, pp. 1122–1125.

[15] Y.-T. Lin, P.-Y. Tsai, and T.-D. Chiueh, "Low-power variable-length fast Fourier transform processor," *IEE Proc.-Comput. Digit. Tech.*, vol. 152, no. 4, pp. 505–506, Jul. 2005.

[16] B. M. Baas, "Fast Fourier Transform Processor Information Page," http://nova.stanford.edu/ bbaas/fftinfo.html.

[17] B. M. Baas, "A low-power, high-performance, 1024-point FFT processor," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, pp. 380–384, Mar. 1999.

Appendix A

SOURCE CODE OF CASE1

```
#include "r4fftditinplace.h"

extern const int Factors[];

/* Introductions of static functions. */
static
unsigned int swapBitPairs(unsigned int i);
static
unsigned int r4bfin_idx(unsigned int stage, unsigned int linidx);

/*Implementation of the algorithm begins.*/
void
r4fftditinplace (const int Input[], int Output[]) {
    unsigned int idx0;
    unsigned int idx1;
    unsigned int idx2;
    unsigned int idx3;
    unsigned int stage;
    unsigned int linidx;
    short int re_num0, re_num1, re_num2, re_num3;
    short int re_prod0, re_prod1, re_prod2, re_prod3;
    short int re_sum0, re_sum1, re_sum2, re_sum3;
    short int re_factor0, re_factor1, re_factor2, re_factor3;
    short int im_num0, im_num1, im_num2, im_num3;
    short int im_prod0,  im_prod1, im_prod2, im_prod3;
    short int im_sum0, im_sum1, im_sum2, im_sum3;
    short int im_factor0, im_factor1, im_factor2, im_factor3;
    unsigned int facto = 0;

    /* Input permutation */
    for (linidx = 0; linidx < FFT_POINTS; ++linidx) {
      Output[linidx] = Input[swapBitPairs(linidx)];
    }
```

```
    /* Butterfly computations through stages of FFT begin.
     * Loop Counters:
     * stage -> counter for stages of fft
     * linidx -> counter for butterfly input operands in a stage of fft
     */
    for (stage = 0; stage < FFT_POINTS_LOG4; stage++) {
        for (linidx = 0; linidx < (FFT_POINTS - 1); linidx += 4) {
            idx0 = r4bfin_idx(stage, linidx);
            idx1 = r4bfin_idx(stage, linidx + 1);
            idx2 = r4bfin_idx(stage, linidx + 2);
            idx3 = r4bfin_idx(stage, linidx + 3);

            re_num0 = ((Output[idx0] >> 16) & 0x0000FFFF);
            im_num0 = (Output[idx0] & 0x0000FFFF);
            re_num1 = ((Output[idx1] >> 16) & 0x0000FFFF);
            im_num1 = (Output[idx1] & 0x0000FFFF);
            re_num2 = ((Output[idx2] >> 16) & 0x0000FFFF);
            im_num2 = (Output[idx2] & 0x0000FFFF);
            re_num3 = ((Output[idx3] >> 16) & 0x0000FFFF);
            im_num3 = (Output[idx3] & 0x0000FFFF);

            re_factor0 = ((Factors[facto] >> 16) & 0x0000FFFF);
            im_factor0 = (Factors[facto] & 0x0000FFFF);
            re_factor1 = ((Factors[facto+1] >> 16) & 0x0000FFFF);
            im_factor1 = (Factors[facto+1] & 0x0000FFFF);
            re_factor2 = ((Factors[facto+2] >> 16) & 0x0000FFFF);
            im_factor2 = (Factors[facto+2] & 0x0000FFFF);
            re_factor3 = ((Factors[facto+3] >> 16) & 0x0000FFFF);
            im_factor3 = (Factors[facto+3] & 0x0000FFFF);
            facto += 4;

            /* Complex multiplications */
            re_prod0 = ((re_num0 * re_factor0) >> 15) - ((im_num0 * im_factor0) >> 15);
            im_prod0 = ((re_num0 * im_factor0) >> 15) + ((im_num0 * re_factor0) >> 15);
            re_prod1 = ((re_num1 * re_factor1) >> 15) - ((im_num1 * im_factor1) >> 15);
            im_prod1 = ((re_num1 * im_factor1) >> 15) + ((im_num1 * re_factor1) >> 15);
            re_prod2 = ((re_num2 * re_factor2) >> 15) - ((im_num2 * im_factor2) >> 15);
            im_prod2 = ((re_num2 * im_factor2) >> 15) + ((im_num2 * re_factor2) >> 15);
            re_prod3 = ((re_num3 * re_factor3) >> 15) - ((im_num3 * im_factor3) >> 15);
            im_prod3 = ((re_num3 * im_factor3) >> 15) + ((im_num3 * re_factor3) >> 15);

            /* Complex additions */
            re_sum0 = ((((re_prod0 + re_prod1) >> 1) + ((re_prod2 + re_prod3) >> 1)) >> 1);
            im_sum0 = ((((im_prod0 + im_prod1) >> 1) + ((im_prod2 + im_prod3) >> 1)) >> 1);

            re_sum1 = ((((re_prod0 + im_prod1) >> 1) - ((re_prod2 + im_prod3) >> 1)) >> 1);
            im_sum1 = ((((im_prod0 - re_prod1) >> 1) - ((im_prod2 - re_prod3) >> 1)) >> 1);

            re_sum2 = ((((re_prod0 - re_prod1) >> 1) + ((re_prod2 - re_prod3) >> 1)) >> 1);
            im_sum2 = ((((im_prod0 - im_prod1) >> 1) + ((im_prod2 - im_prod3) >> 1)) >> 1);

            re_sum3 = ((((re_prod0 - im_prod1) >> 1) - ((re_prod2 - im_prod3) >> 1)) >> 1);
            im_sum3 = ((((im_prod0 + re_prod1) >> 1) - ((im_prod2 + re_prod3) >> 1)) >> 1);

            /* Store the results of butterfly evaluation back to Output
             * -array into the same index locations.
             */
            Output[idx0] = (((int) re_sum0) << 16) | (((int) im_sum0) & 0x0000FFFF);
            Output[idx1] = (((int) re_sum1) << 16) | (((int) im_sum1) & 0x0000FFFF);
            Output[idx2] = (((int) re_sum2) << 16) | (((int) im_sum2) & 0x0000FFFF);
            Output[idx3] = (((int) re_sum3) << 16) | (((int) im_sum3) & 0x0000FFFF);
        }
    }
}
```

```
/* Declarations of static functions */

/* This function permutes the lowermost 10 bits of linear i by swapping two bits'
 * bit fields from the LSB- and MSB-parts of i in pairs. I.e., the lowermost two
 * bits are swapped with the 9th and 10th bits etc.
 */
static
unsigned int swapBitPairs(unsigned int i) {
  unsigned int bit_pair0, bit_pair1, bit_pair3, bit_pair4;
  bit_pair0 = (i >> 8);
  bit_pair1 = ((i >> 4) & 0x0000000C);
  bit_pair3 = ((i << 4) & 0x000000C0);
  bit_pair4 = ((i << 8) & 0x00000300);
  return (bit_pair4 | bit_pair3 | (i & 0x00000030) | bit_pair1 | bit_pair0);
}

/* This function forms an input index of a butterfly by rotating the
 * (2*stage+2) lowermost bits of the linidx two
 * bits to the right; i.e., the output-array is accessed with the aid of this
 * function in the butterfly computations.
 */
static
unsigned int r4bfin_idx(unsigned int stage, unsigned int linidx) {
    unsigned int upper_part = 0; unsigned int rotated_part = 0;
    unsigned int rotated_MSB = 0; unsigned int rotated_LSB_part = 0;
    unsigned int idx = 0;
    unsigned int rotated_part_width = (stage << 1) + 2;

    if (stage > 0) {
        upper_part = ((linidx >> rotated_part_width) << rotated_part_width);
        rotated_part = ((linidx << (32 - rotated_part_width)) >>
                        (32 - rotated_part_width));
        rotated_LSB_part = rotated_part >> 2;
        rotated_MSB = ((rotated_part << 30) >> 30);
        rotated_MSB = (rotated_MSB << (rotated_part_width - 2));

/* Concatenate different parts of the final index by orring bit by bit
   and return the correct index.
*/
        idx = (upper_part | rotated_MSB | rotated_LSB_part);
/* Assertions that can be used to verify that results are legal. */
#ifndef __move__
        assert(idx <= 1023);
#endif
        return idx;

    } else {
#ifndef __move__
        assert(linidx <= 1023);
#endif
        return linidx;
    }
}
```

Appendix B

SOURCE CODE OF CASE2

```
/*Implementation of the algorithm begins.*/
void
r4fftditinplace (const int Input[], int Output[]) {
    unsigned int idx0, idx1, idx2, idx3;
    unsigned int stage, linidx;
    int prod0, prod1, prod2, prod3;
    unsigned int facto = 0;

    /* Input permutation */
    for (linidx = 0; linidx < FFT_POINTS; ++linidx) {
      Output[linidx] = Input[swapBitPairs(linidx)];
    }

    /* Butterfly computations through stages of FFT begin.
     * Loop Counters:
     * stage -> counter for stages of fft
     * linidx -> counter for butterfly input operands in a stage of fft
     */
    for (stage = 0; stage < FFT_POINTS_LOG4; stage++) {
        for (linidx = 0; linidx < (FFT_POINTS - 1); linidx += 4) {
            idx0 = r4bfin_idx(stage, linidx);
            idx1 = r4bfin_idx(stage, linidx + 1);
            idx2 = r4bfin_idx(stage, linidx + 2);
            idx3 = r4bfin_idx(stage, linidx + 3);

            /* Complex multiplications are performed with the aid of the cmul-SFU. */
            prod0 = cmul(Output[idx0], Factors[facto]);
            prod1 = cmul(Output[idx1], Factors[facto+1]);
            prod2 = cmul(Output[idx2], Factors[facto+2]);
            prod3 = cmul(Output[idx3], Factors[facto+3]);
            facto += 4;

            /* Complex summations are performed with the aid of the cadd-SFU. */
            Output[idx0] = cadd(prod0,prod1,prod2,prod3,0);
            Output[idx1] = cadd(prod0,prod1,prod2,prod3,1);
            Output[idx2] = cadd(prod0,prod1,prod2,prod3,2);
            Output[idx3] = cadd(prod0,prod1,prod2,prod3,3);
        }
    }
}
```

```
/*This function simulates the multiplication of two complex numbers.*/
inline int
r4mul(int in1, int in2) {
    Complex num1, num2, prod;
    Word(num1) = in1;
    Word(num2) = in2;

/* If the value of multiplier, i.e. twiddle-factor, equals 1,
 * multiplicand, i.e. a value of input table of FFT, can be returned.
 */
    if (in2 == 0x7FFF0000) {
        return in1;

    } else {
        Real(prod) = ((Real(num1) * Real(num2)) >> 15) - ((Imag(num1) * Imag(num2)) >> 15);
        Imag(prod) = ((Real(num1) * Imag(num2)) >> 15) + ((Imag(num1) * Real(num2)) >> 15);

        return Word(prod);
    }
}


 /* This function simulates the addition of four complex numbers in four
 * sligthly different forms needed in the butterfly calculations of radix-4
 * DIT FFT.
 */
int cadd(int in1, int in2, int in3, int in4, unsigned char format) {

    Complex num1, num2, num3, num4, sum;
    Word(num1) = in1;
    Word(num2) = in2;
    Word(num3) = in3;
    Word(num4) = in4;

    switch(format) {
    case 0:
        Real(sum) = ((((Real(num1) + Real(num2)) >> 1) + ((Real(num3) + Real(num4)) >> 1)) >> 1);
        Imag(sum) = ((((Imag(num1) + Imag(num2)) >> 1) + ((Imag(num3) + Imag(num4)) >> 1)) >> 1);
        break;
    case 1:
        Real(sum) = ((((Real(num1) + Imag(num2)) >> 1) - ((Real(num3) + Imag(num4)) >> 1)) >> 1);
        Imag(sum) = ((((Imag(num1) - Real(num2)) >> 1) - ((Imag(num3) - Real(num4)) >> 1)) >> 1);
        break;
    case 2:
        Real(sum) = ((((Real(num1) - Real(num2)) >> 1) + ((Real(num3) - Real(num4)) >> 1)) >> 1);
        Imag(sum) = ((((Imag(num1) - Imag(num2)) >> 1) + ((Imag(num3) - Imag(num4)) >> 1)) >> 1);
        break;
    case 3:
        Real(sum) = ((((Real(num1) - Imag(num2)) >> 1) - ((Real(num3) - Imag(num4)) >> 1)) >> 1);
        Imag(sum) = ((((Imag(num1) + Real(num2)) >> 1) - ((Imag(num3) + Real(num4)) >> 1)) >> 1);
        break;
    }
    return Word(sum);
}
```

Appendix C

SOURCE CODE OF CASE3

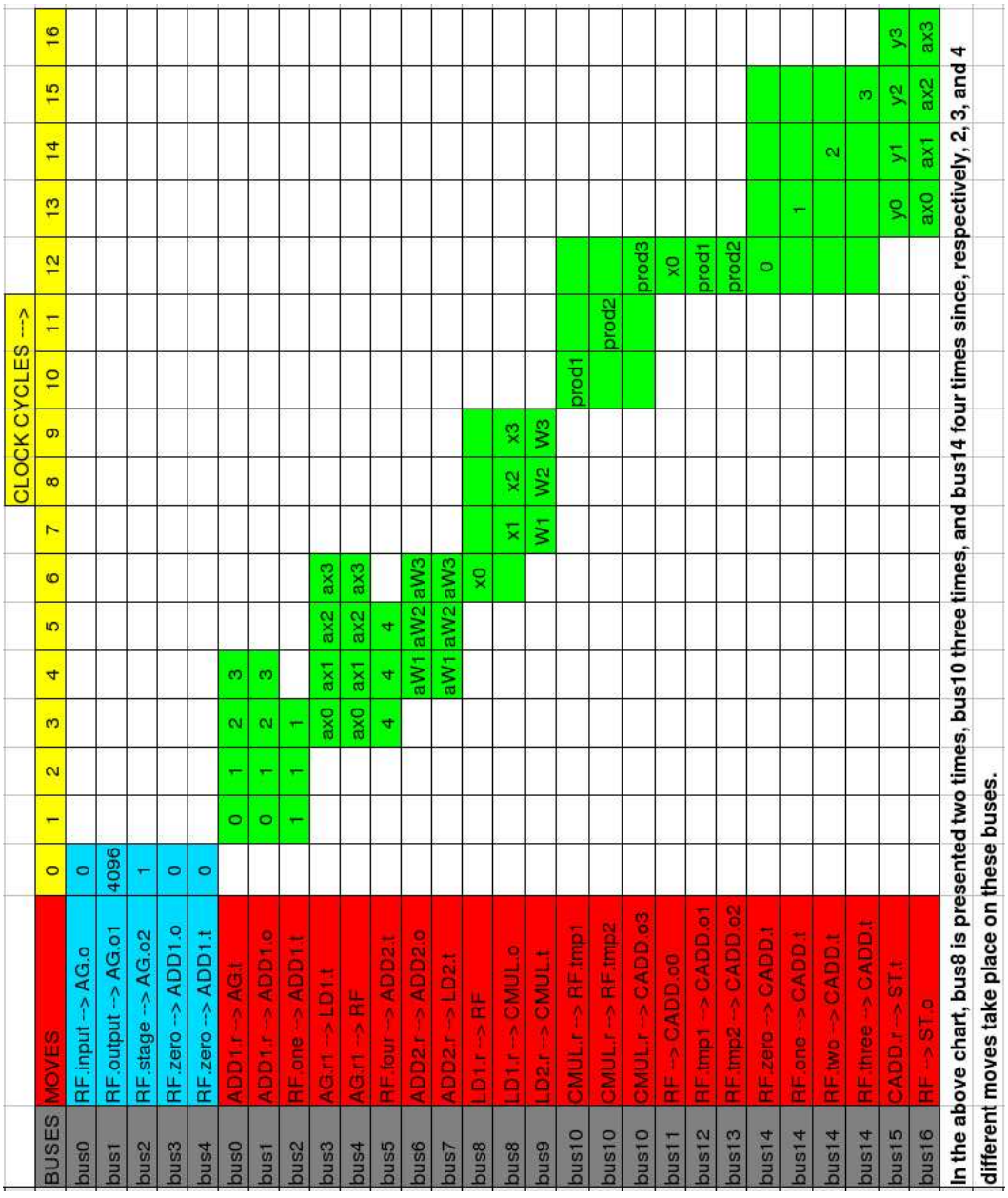
```
#include "r4fftditinplace.h"
#include "sfus.h"

extern const int Factors[];

/*Implementation of the algorithm begins.*/
void
r4fftditinplace (const int Input[], int Output[]) {
    int oper0, oper1, oper2, oper3, oper4, oper5, oper6, oper7;
    int *oper0in_addr = 0;
    int *oper1in_addr = 0;
    int *oper2in_addr = 0;
    int *oper3in_addr = 0;
    int *oper4in_addr = 0;
    int *oper5in_addr = 0;
    int *oper6in_addr = 0;
    int *oper7in_addr = 0;
    int *oper0out_addr = 0;
    int *oper1out_addr = 0;
    int *oper2out_addr = 0;
    int *oper3out_addr = 0;
    int *oper4out_addr = 0;
    int *oper5out_addr = 0;
    int *oper6out_addr = 0;
    int *oper7out_addr = 0;
    unsigned int stage = 0;
    unsigned int linidx = 0;
    int prod1, prod2, prod3, prod5, prod6, prod7;
    unsigned int facto = 0;
```

```
/* Loop Counters:
 * stage -> counter for stages of FFT
 * linidx -> counter for butterfly input operands in a stage of FFT
 */
for (linidx = 0; linidx < (FFT_POINTS - 1); linidx += 8) {
  ag(oper0in_addr,oper0out_addr,Input,Output,stage,linidx);
  ag(oper1in_addr,oper1out_addr,Input,Output,stage,linidx+1);
  ag(oper2in_addr,oper2out_addr,Input,Output,stage,linidx+2);
  ag(oper3in_addr,oper3out_addr,Input,Output,stage,linidx+3);

  ag(oper4in_addr,oper4out_addr,Input,Output,stage,linidx+4);
  ag(oper5in_addr,oper5out_addr,Input,Output,stage,linidx+5);
  ag(oper6in_addr,oper6out_addr,Input,Output,stage,linidx+6);
  ag(oper7in_addr,oper7out_addr,Input,Output,stage,linidx+7);

  oper0 = *oper0in_addr;
  oper1 = *oper1in_addr;
  oper2 = *oper2in_addr;
  oper3 = *oper3in_addr;

  oper4 = *oper4in_addr;
  oper5 = *oper5in_addr;
  oper6 = *oper6in_addr;
  oper7 = *oper7in_addr;

  /* Every 4:th twiddle-factor equals always one so that there
   * is no need to utilize the cmul-SFU as computing the result
   * of every 4:th complex multiplication.
   */

  /* The rest of the complex multiplications are performed with
   * the aid of the cmul-SFU.
   */
  prod1 = cmul(oper1, Factors[facto+1]);
  prod2 = cmul(oper2, Factors[facto+2]);
  prod3 = cmul(oper3, Factors[facto+3]);

  prod5 = cmul(oper5, Factors[facto+5]);
  prod6 = cmul(oper6, Factors[facto+6]);
  prod7 = cmul(oper7, Factors[facto+7]);
  facto += 8;

  /* Complex summations are performed with the aid of
   * the cadd-SFU.
   */
  *oper0out_addr = cadd(oper0,prod1,prod2,prod3,0);
  *oper1out_addr = cadd(oper0,prod1,prod2,prod3,1);
  *oper2out_addr = cadd(oper0,prod1,prod2,prod3,2);
  *oper3out_addr = cadd(oper0,prod1,prod2,prod3,3);

  *oper4out_addr = cadd(oper4,prod5,prod6,prod7,0);
  *oper5out_addr = cadd(oper4,prod5,prod6,prod7,1);
  *oper6out_addr = cadd(oper4,prod5,prod6,prod7,2);
  *oper7out_addr = cadd(oper4,prod5,prod6,prod7,3);
}
```

```
    for (stage = 1; stage < FFT_POINTS_LOG4; stage++) {
      for (linidx = 0; linidx < (FFT_POINTS - 1); linidx += 8) {
        ag(oper0in_addr,oper0out_addr,Input,Output,stage,linidx);
        ag(oper1in_addr,oper1out_addr,Input,Output,stage,linidx+1);
        ag(oper2in_addr,oper2out_addr,Input,Output,stage,linidx+2);
        ag(oper3in_addr,oper3out_addr,Input,Output,stage,linidx+3);

        ag(oper4in_addr,oper4out_addr,Input,Output,stage,linidx+4);
        ag(oper5in_addr,oper5out_addr,Input,Output,stage,linidx+5);
        ag(oper6in_addr,oper6out_addr,Input,Output,stage,linidx+6);
        ag(oper7in_addr,oper7out_addr,Input,Output,stage,linidx+7);

        oper0 = *oper0out_addr;
        oper1 = *oper1out_addr;
        oper2 = *oper2out_addr;
        oper3 = *oper3out_addr;

        oper4 = *oper4out_addr;
        oper5 = *oper5out_addr;
        oper6 = *oper6out_addr;
        oper7 = *oper7out_addr;

        prod1 = cmul(oper1, Factors[facto+1]);
        prod2 = cmul(oper2, Factors[facto+2]);
        prod3 = cmul(oper3, Factors[facto+3]);

        prod5 = cmul(oper5, Factors[facto+5]);
        prod6 = cmul(oper6, Factors[facto+6]);
        prod7 = cmul(oper7, Factors[facto+7]);
        facto += 8;

        /* Complex summations are performed with the aid of
         * the cadd-SFU.
         */
        *oper0out_addr = cadd(oper0,prod1,prod2,prod3,0);
        *oper1out_addr = cadd(oper0,prod1,prod2,prod3,1);
        *oper2out_addr = cadd(oper0,prod1,prod2,prod3,2);
        *oper3out_addr = cadd(oper0,prod1,prod2,prod3,3);

        *oper4out_addr = cadd(oper4,prod5,prod6,prod7,0);
        *oper5out_addr = cadd(oper4,prod5,prod6,prod7,1);
        *oper6out_addr = cadd(oper4,prod5,prod6,prod7,2);
        *oper7out_addr = cadd(oper4,prod5,prod6,prod7,3);
      }
    }
}
```

Appendix D

OPERATION SCHEDULING OF RADIX-4 DIT BUTTERFLY



		CLOCK CYCLES -->															
BUSES	MOVES	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
bus0	RF.input --> AG.o	0																
bus1	RF.output --> AG.o1	4096																
bus2	RF.stage --> AG.o2	1																
bus3	RF.zero --> ADD1.o	0																
bus4	RF.zero --> ADD1.t	0																
bus0	ADD1.r --> AG.t		0	1	2	3												
bus1	ADD1.r --> ADD1.o		0	1	2	3												
bus2	RF.one --> ADD1.t		1	1	1													
bus3	AG.r1 --> LD1.t				ax0	ax1	ax2	ax3										
bus4	AG.r1 --> RF				ax0	ax1	ax2	ax3										
bus5	RF.four --> ADD2.t				4	4	4											
bus6	ADD2.r --> ADD2.o					aW1	aW2	aW3										
bus7	ADD2.r --> LD2.t					aW1	aW2	aW3										
bus8	LD1.r --> RF							x0										
bus8	LD1.r --> CMUL.o								x1	x2	x3							
bus9	LD2.r --> CMUL.t								W1	W2	W3							
bus10	CMUL.r --> RF.tmp1											prod1						
bus10	CMUL.r --> RF.tmp2												prod2					
bus10	CMUL.r --> CADD.o3													prod3				
bus11	RF --> CADD.o0													x0				
bus12	RF.tmp1 --> CADD.o1													prod1				
bus13	RF.tmp2 --> CADD.o2													prod2				
bus14	RF.zero --> CADD.t													0				
bus14	RF.one --> CADD.t														1			
bus14	RF.two --> CADD.t															2		
bus14	RF.three --> CADD.t																3	
bus15	CADD.r --> ST.t														y0	y1	y2	y3
bus16	RF --> ST.o														ax0	ax1	ax2	ax3

In the above chart, bus8 is presented two times, bus10 three times, and bus14 four times since, respectively, 2, 3, and 4 different moves take place on these buses.

Appendix E

OPERATION SCHEDULING FOR DISCOVERING THE KERNEL

										CLOCK CYCLES --->																
BUSES	MOVES	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
bus3	RF.zero --> ADD1.o	0																								
bus4	RF.zero --> ADD1.t	0																								
bus0	ADD1.r --> AG.t		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
bus1	ADD1.r --> ADD1.o		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
bus2	RF.one --> ADD1.t		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
bus3	AG.r1 --> LD1.t				ax0	ax1	ax2	ax3	ax4	ax5	ax6	ax7	ax8	ax9	ax10	ax11	ax12	ax13	ax14	ax15	ax16	ax17	ax18	ax19	ax20	ax21
bus4	AG.r1 --> RF				ax0	ax1	ax2	ax3	ax4	ax5	ax6	ax7	ax8	ax9	ax10	ax11	ax12	ax13	ax14	ax15	ax16	ax17	ax18	ax19	ax20	ax21
bus5	RF.four --> ADD2.t				4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
bus6	ADD2.r --> ADD2.o					aW1	aW2	aW3	aW4	aW5	aW6	aW7	aW8	aW9	aW10	aW11	aW12	aW13	aW14	aW15	aW16	aW17	aW18	aW19	aW20	aW21
bus7	ADD2.r --> LD2.t					aW1	aW2	aW3		aW5	aW6	aW7		aW9	aW10	aW11		aW13	aW14	aW15		aW17	aW18	aW19		aW21
bus8	LD1.r --> RF							x0				x4				x8				x12				x16		
bus8	LD1.r --> CMUL.o								x1	x2	x3		x5	x6	x7		x9	x10	x11		x13	x14	x15		x17	x18
bus9	LD2.r --> CMUL.t								W1	W2	W3		W5	W6	W7		W9	W10	W11		W13	W14	W15		W17	W18
bus10	CMUL.r --> RF.tmp1											prod1				prod5				prod9				prod13		
bus10	CMUL.r --> RF.tmp2												prod2				prod6				prod10				prod14	
bus10	CMUL.r --> CADD.o3													prod3				prod7				prod11				prod15
bus11	RF --> CADD.o0													x0				x4				x8				x12
bus12	RF.tmp1 --> CADD.o1													prod1				prod5				prod9				prod13
bus13	RF.tmp2 --> CADD.o2													prod2				prod6				prod10				prod14
bus14	RF.zero --> CADD.t													0				0				0				0
bus14	RF.one --> CADD.t														1				1				1			
bus14	RF.two --> CADD.t															2				2				2		
bus14	RF.three --> CADD.t																3				3				3	
bus15	CADD.r --> ST.t														y0	y1	y2	y3	y4	y5	y6	y7	y8	y9	y10	y11
bus16	RF --> ST.o														ax0	ax1	ax2	ax3	ax4	ax5	ax6	ax7	ax8	ax9	ax10	ax11

In the above chart, bus8 is presented two times, bus10 three times, and bus14 four times since, respectively, 2, 3, and 4 different moves take place on these buses.

Appendix F

TTA ASSEMBLY CODE

```
// 1K R4 DIT FFT
// ********************************************************************************************************
// TTA ASSEMBLY with correct registers when JUMP latency = 4 and LSU latency = 3
// ********************************************************************************************************
// Latencies of other FUs:
// addrgen 2
// cmul    3
// cadd    1
// add     1
// cmp     1
// ********************************************************************************************************
// Number of move buses = 31
// Number of instructions = 51
// ********************************************************************************************************
// Author: Risto Makinen <rmmakine@cs.tut.fi>
// Organization: Institute of Digital and Computer Systems, Tampere University of Technology (TUT), Finland
// Project: FlexDSP
// Date: 2005-18-08
// ********************************************************************************************************
// This assembly code has to be compiled with ttasm and proper machine configuration file.
// Then the generated binary code can be simulated with HW-simulator vsim. The processor model for vsim
// can be generated with the processor generator MOVEgen.
// ********************************************************************************************************


// BB0
// purpose      : Initializations
// frequency    : 1 (* 10 = 10)

// Fill registers with initial parameters:
// r15 stage
// r16 zero register
// r17 one register
// r18 two register
// r19 three register
// rv four register
// r20 base address of input buffer
// r21 base address of buffer for twiddle factors
// r22 base address of output buffer
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 0 -> r15 [m31/i1/ir_1/ri_i5]; // INSTRUCTION 0 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 0 -> r16 [m31/i1/ir_1/ri_i6]; // INSTRUCTION 1 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 1 -> r17 [m31/i1/ir_1/ri_i7]; // INSTRUCTION 2 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 2 -> r18 [m31/i1/ir_1/ri_i8]; // INSTRUCTION 3 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 3 -> r19 [m31/i1/ir_1/ri_i9]; // INSTRUCTION 4 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 4 -> rv  [m31/i1/ir_1/ri_i1]; // INSTRUCTION 5 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 0 -> r20 [m31/i1/ir_1/ri_i10]; // INSTRUCTION 6 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., 0 -> r21 [m31/i1/ir_1/ri_i11]; // INSTRUCTION 7 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
```

```
..., ..., ..., ..., ..., ..., ..., ..., 4096 -> r22 [m31/i1/ir_1/ri_i1]; // INSTRUCTION 8 ENDS
..., ..., r20 -> fu1.ag_o [m3/-/ri_o10/fu1_o], r22 -> fu1.ag_o1 [m4/-/ri_o1/fu1_o1], ..., ..., ..., ..., ...,
..., r21 -> fu5.add_o [m11/-/ri_o11/fu5_o], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., 1013 -> r21 [m31/i1/ir_1/ri_i11]; // INSTRUCTION 9 ENDS

// BB1
// purpose      : Upkeeping of the outer loop
// frequency    : 5 (* 2 = 10)
..., ..., r15 -> fu1.ag_o2 [m3/-/ri_o5/fu1_o2], r16 -> fu9.eq_o [m4/-/ri_o6/fu9_o],
r15 -> fu9.eq_t [m5/-/ri_o5/fu9_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 10 ENDS
fu9.eq_r -> b0 [m1/-/fu9_r/b_i1], ..., ..., r16 -> fu4.add_o [m4/-/ri_o6/fu4_o],
r16 -> fu4.add_t [m5/-/ri_o6/fu4_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 11 ENDS

// BB2
// purpose      : Start the pipeline for FFT
// frequency    : 5 (* 15 = 75)
// comment      : Inner loop’s prologue
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 12 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 13 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r1  [m8/-/fu1_r1/ri_i2], ..., ..., ..., ...,
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ...; // INSTRUCTION 14 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r2  [m8/-/fu1_r1/ri_i3], ..., ...,
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ...; // INSTRUCTION 15 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r3  [m8/-/fu1_r1/ri_i4], ..., ...,
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ...; // INSTRUCTION 16 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r4  [m8/-/fu1_r1/ri_i5],
fu6.ld_r -> r11 [m9/-/fu6_r/ri_i1], ..., fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o],
fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t], rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 17 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r5  [m8/-/fu1_r1/ri_i6],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], ..., rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 18 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r6  [m8/-/fu1_r1/ri_i7],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ...; // INSTRUCTION 19 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r7  [m8/-/fu1_r1/ri_i8],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ...; // INSTRUCTION 20 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r8  [m8/-/fu1_r1/ri_i9],
fu6.ld_r -> r12 [m9/-/fu6_r/ri_i2], ..., fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o],
fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t], rv -> fu5.add_t [m13/-/ri_o1/fu5_t],
fu2.cmul_r -> fu3.cadd_o1 [m14/-/fu2_r/fu3_o1], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
```

```
..., ..., ..., ..., ..., ...; // INSTRUCTION 21 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r9  [m8/-/fu1_r1/ri_i10],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], ..., rv -> fu5.add_t [m13/-/ri_o1/fu5_t],
fu2.cmul_r -> fu3.cadd_o2 [m14/-/fu2_r/fu3_o2], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ...; // INSTRUCTION 22 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], fu2.cmul_r -> fu3.cadd_o3 [m14/-/fu2_r/fu3_o3],
r16 -> fu3.cadd_t [m15/-/ri_o6/fu3_t], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
r11 -> fu3.cadd_o [m27/-/ri_o12/fu3_o], r12 -> r11 [m28/-/ri_o22/ri_i1], ..., ..., ...; // INSTRUCTION 23 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., r17 -> fu3.cadd_t [m15/-/ri_o7/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
r16 -> fu10.gtu_o [m27/-/ri_o6/fu10_o], r16 -> fu10.gtu_t [m28/-/ri_o6/fu10_t],
..., ..., ...; // INSTRUCTION 24 ENDS
..., fu10.gtu_r -> b1 [m2/-/fu10_r/b_i1], fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o],
fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t], r17 -> fu4.add_t [m5/-/ri_o7/fu4_t],
b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t], !b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t],
fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11], fu6.ld_r -> r12 [m9/-/fu6_r/ri_i2], ...,
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], fu2.cmul_r -> r13 [m14/-/fu2_r/ri_i3], r18 -> fu3.cadd_t [m15/-/ri_o8/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10],
r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7],
r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42],
r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22], ..., ..., ..., ..., ...; // INSTRUCTION 25 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], ..., rv -> fu5.add_t [m13/-/ri_o1/fu5_t],
fu2.cmul_r -> r14 [m14/-/fu2_r/ri_i4], r19 -> fu3.cadd_t [m15/-/ri_o9/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9],
r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6],
r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32],
r2 -> r1 [m26/-/ri_o3/ri_i22], ..., ..., ..., ..., ...; // INSTRUCTION 26 ENDS
```

```
// BB3
// purpose      :
// frequency    : 252 * 5 = 1260 (* 4 = 5040)
// comment      : Inner loop’s kernel: jump latency = 4 and LSU latency = 3
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], fu2.cmul_r -> fu3.cadd_o3 [m14/-/fu2_r/fu3_o3],
r16 -> fu3.cadd_t [m15/-/ri_o6/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10],
r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7],
r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42],
r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22], r11 -> fu3.cadd_o [m27/-/ri_o12/fu3_o],
r12 -> r11 [m28/-/ri_o22/ri_i1], r13 -> fu3.cadd_o1 [m29/-/ri_o32/fu3_o1],
r14 -> fu3.cadd_o2 [m30/-/ri_o42/fu3_o2], !b1:27 -> jump [m31/i1/ir_1/pc]; // INSTRUCTION 27 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., r17 -> fu3.cadd_t [m15/-/ri_o7/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 28 ENDS
..., ..., fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o], fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t],
r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> r12 [m9/-/fu6_r/ri_i2], ..., fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o],
fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t], rv -> fu5.add_t [m13/-/ri_o1/fu5_t],
fu2.cmul_r -> r13 [m14/-/fu2_r/ri_i3], r18 -> fu3.cadd_t [m15/-/ri_o8/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
fu4.add_r -> fu10.gtu_o [m27/-/fu4_r/fu10_o], r21 -> fu10.gtu_t [m28/-/ri_o112/fu10_t],
..., ..., ...; // INSTRUCTION 29 ENDS
..., fu10.gtu_r -> b1 [m2/-/fu10_r/b_i1], fu4.add_r -> fu4.add_o [m3/-/fu4_r/fu4_o],
fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t], r17 -> fu4.add_t [m5/-/ri_o7/fu4_t],
b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t], !b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t],
fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11], fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o],
fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t], fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], ...,
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], fu2.cmul_r -> r14 [m14/-/fu2_r/ri_i4],
r19 -> fu3.cadd_t [m15/-/ri_o9/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10],
r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7],
r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42],
r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 30 ENDS

// BB4
// purpose      : Stop the pipeline for FFT
// frequency    : 5 (* 13 = 65)
// comment      : Inner loop’s epilogue
..., ..., ..., fu4.add_r -> fu1.ag_t [m4/-/fu4_r/fu1_t], ..., b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], fu2.cmul_r -> fu3.cadd_o3 [m14/-/fu2_r/fu3_o3],
r16 -> fu3.cadd_t [m15/-/ri_o6/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10],
r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7],
r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42],
r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22], r11 -> fu3.cadd_o [m27/-/ri_o12/fu3_o],
r12 -> r11 [m28/-/ri_o22/ri_i1], r13 -> fu3.cadd_o1 [m29/-/ri_o32/fu3_o1],
r14 -> fu3.cadd_o2 [m30/-/ri_o42/fu3_o2], ...; // INSTRUCTION 31 ENDS
..., ..., ..., ..., ..., b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o], fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t],
fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t],
rv -> fu5.add_t [m13/-/ri_o1/fu5_t], ..., r17 -> fu3.cadd_t [m15/-/ri_o7/fu3_t],
```

```
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 32 ENDS
..., ..., ..., ..., ..., b0:fu1.ag_r0 -> fu6.ld_t [m6/-/fu1_r0/fu6_t],
!b0:fu1.ag_r1 -> fu6.ld_t [m7/-/fu1_r1/fu6_t], fu1.ag_r1 -> r10 [m8/-/fu1_r1/ri_i11],
fu6.ld_r -> r12 [m9/-/fu6_r/ri_i2], ..., fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o],
fu5.add_r -> fu7.ld_t [m12/-/fu5_r/fu7_t], rv -> fu5.add_t [m13/-/ri_o1/fu5_t],
fu2.cmul_r -> r13 [m14/-/fu2_r/ri_i3], r18 -> fu3.cadd_t [m15/-/ri_o8/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 33 ENDS
..., ..., ..., ..., ..., ..., ..., ..., fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o],
fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t], fu5.add_r -> fu5.add_o [m11/-/fu5_r/fu5_o], ..., ...,
fu2.cmul_r -> r14 [m14/-/fu2_r/ri_i4], r19 -> fu3.cadd_t [m15/-/ri_o9/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 34 ENDS
..., ..., ..., ..., ..., ..., ..., ..., fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o],
fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t], ..., ..., ..., fu2.cmul_r -> fu3.cadd_o3 [m14/-/fu2_r/fu3_o3],
r16 -> fu3.cadd_t [m15/-/ri_o6/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9],
r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6],
r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32],
r2 -> r1 [m26/-/ri_o3/ri_i22], r11 -> fu3.cadd_o [m27/-/ri_o12/fu3_o], r12 -> r11 [m28/-/ri_o22/ri_i1],
r13 -> fu3.cadd_o1 [m29/-/ri_o32/fu3_o1], r14 -> fu3.cadd_o2 [m30/-/ri_o42/fu3_o2],
...; // INSTRUCTION 35 ENDS
..., ..., ..., ..., ..., ..., ..., ..., fu6.ld_r -> fu2.cmul_o [m9/-/fu6_r/fu2_o],
fu7.ld_r -> fu2.cmul_t [m10/-/fu7_r/fu2_t], ..., ..., ..., ..., r17 -> fu3.cadd_t [m15/-/ri_o7/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 36 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., fu2.cmul_r -> r13 [m14/-/fu2_r/ri_i3],
r18 -> fu3.cadd_t [m15/-/ri_o8/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9],
r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6],
r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32],
r2 -> r1 [m26/-/ri_o3/ri_i22], ..., ..., ..., ..., ...; // INSTRUCTION 37 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., fu2.cmul_r -> r14 [m14/-/fu2_r/ri_i4],
r19 -> fu3.cadd_t [m15/-/ri_o9/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9],
r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6],
r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32],
r2 -> r1 [m26/-/ri_o3/ri_i22], ..., ..., ..., ..., ...; // INSTRUCTION 38 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
fu2.cmul_r -> fu3.cadd_o3 [m14/-/fu2_r/fu3_o3], r16 -> fu3.cadd_t [m15/-/ri_o6/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
r11 -> fu3.cadd_o [m27/-/ri_o12/fu3_o], r12 -> r11 [m28/-/ri_o22/ri_i1],
r13 -> fu3.cadd_o1 [m29/-/ri_o32/fu3_o1], r14 -> fu3.cadd_o2 [m30/-/ri_o42/fu3_o2],
...; // INSTRUCTION 39 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
r17 -> fu3.cadd_t [m15/-/ri_o7/fu3_t], r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9],
r8 -> r7 [m20/-/ri_o92/ri_i8], r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6],
r5 -> r4 [m23/-/ri_o62/ri_i5], r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32],
r2 -> r1 [m26/-/ri_o3/ri_i22], ..., ..., ..., ..., ...; // INSTRUCTION 40 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., r18 -> fu3.cadd_t [m15/-/ri_o8/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 41 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., r19 -> fu3.cadd_t [m15/-/ri_o9/fu3_t],
r1 -> fu8.st_o [m16/-/ri_o2/fu8_o], fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t],
```

```
r10 -> r9 [m18/-/ri_o11/ri_i10], r9 -> r8 [m19/-/ri_o10/ri_i9], r8 -> r7 [m20/-/ri_o92/ri_i8],
r7 -> r6 [m21/-/ri_o82/ri_i7], r6 -> r5 [m22/-/ri_o72/ri_i6], r5 -> r4 [m23/-/ri_o62/ri_i5],
r4 -> r3 [m24/-/ri_o5/ri_i42], r3 -> r2 [m25/-/ri_o4/ri_i32], r2 -> r1 [m26/-/ri_o3/ri_i22],
..., ..., ..., ..., ...; // INSTRUCTION 42 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., r1 -> fu8.st_o [m16/-/ri_o2/fu8_o],
fu3.cadd_r -> fu8.st_t [m17/-/fu3_r/fu8_t], ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ...; // INSTRUCTION 43 ENDS

// BB5
// purpose      : Upkeeping of the outer loop
// frequency    : 5 (* 7 = 35)
..., ..., r15 -> fu4.add_o [m3/-/ri_o5/fu4_o], ..., r17 -> fu4.add_t [m5/-/ri_o7/fu4_t], ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 44 ENDS
..., ..., ..., fu4.add_r -> fu9.gtu_o [m4/-/fu4_r/fu9_o], rv -> fu9.gtu_t [m5/-/ri_o1/fu9_t],
..., ..., fu4.add_r -> r15 [m8/-/fu4_r/ri_i5], ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 45 ENDS
fu9.gtu_r -> b0 [m1/-/fu9_r/b_i1], ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ...; // INSTRUCTION 46 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., !b0:10 -> jump [m31/i1/ir_1/pc]; // INSTRUCTION 47 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 48 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 49 ENDS
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...,
..., ..., ..., ..., ..., ..., ..., ..., ..., ..., ...; // INSTRUCTION 50 ENDS
```

Appendix G

ARCHITECTURE DESCRIPTION FILE OF PROCESSOR

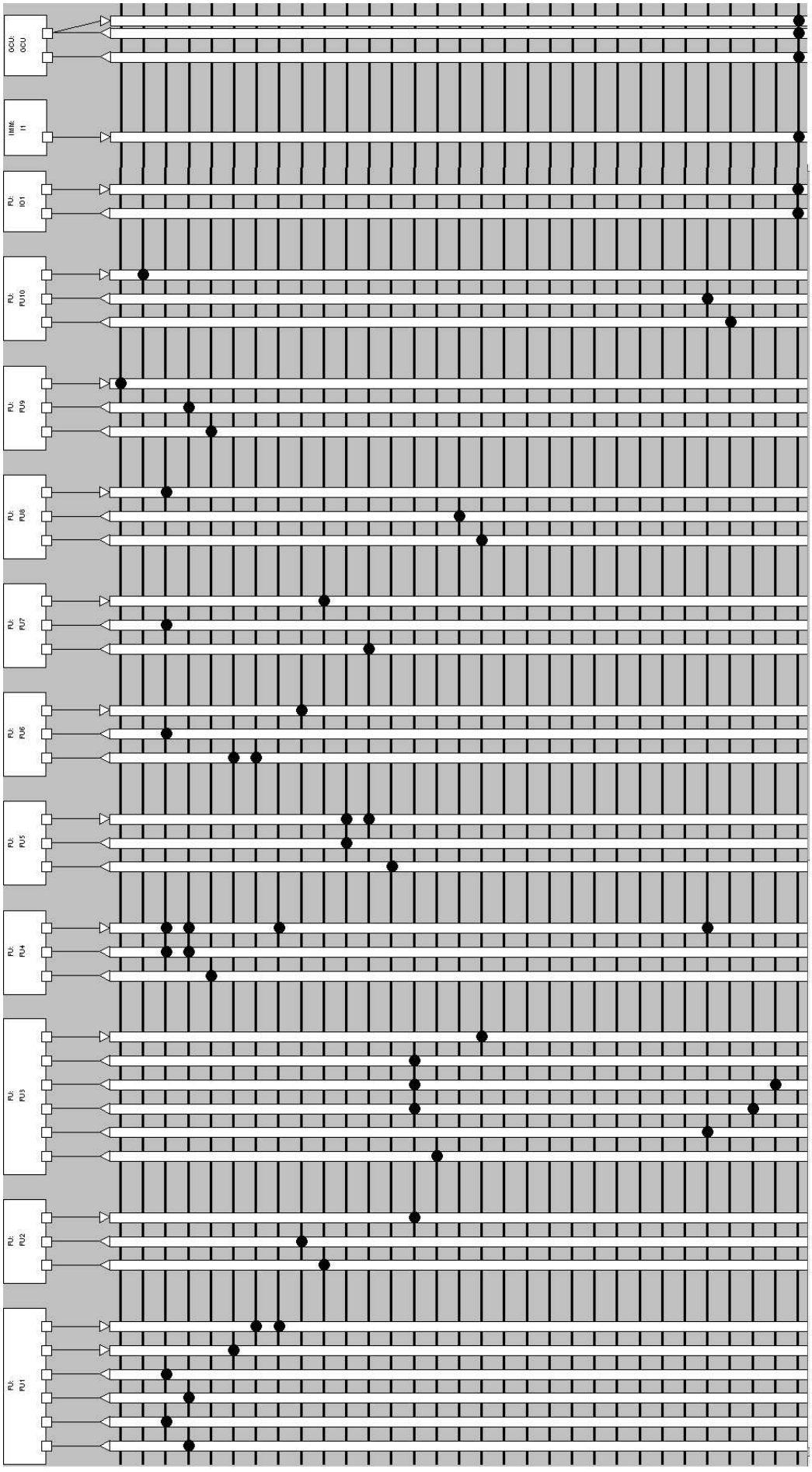
```
/*****************************************************
 * File: mach.for.man.sched.conn.opt
 *****************************************************
 * Description: A mach-file generated manually by hand
 * on the basis of manual schedulation of 1k R4 DIT FFT.
 * This mach-file provides optimal resources for the
 * manually scheduled FFT. The connectivity is as minimal
 * as possible in this processor configuration.
 *****************************************************
 * Project: FlexDSP
 * Author:  Risto Makinen <rmmakine@cs.tut.fi>
 *****************************************************/

MoveBusses
{
        m1              1,  0, unsigned;
        m2              1,  0, unsigned;
        m3              32, 0, signed;
        m4              32, 0, signed;
        m5              32, 0, signed;
        m6              32, 0, signed;
        m7              32, 0, signed;
        m8              32, 0, signed;
        m9              32, 0, signed;
        m10             32, 0, signed;
        m11             32, 0, signed;
        m12             32, 0, signed;
        m13             32, 0, signed;
        m14             32, 0, signed;
        m15             32, 0, signed;
        m16             32, 0, signed;
        m17             32, 0, signed;
        m18             32, 0, signed;
        m19             32, 0, signed;
        m20             32, 0, signed;
        m21             32, 0, signed;
        m22             32, 0, signed;
        m23             32, 0, signed;
        m24             32, 0, signed;
        m25             32, 0, signed;
        m26             32, 0, signed;
        m27             32, 0, signed;
        m28             32, 0, signed;
        m29             32, 0, signed;
        m30             32, 0, signed;
        m31             32, 0, signed;
}
```

```
Sockets
{
        fu1_o           input,  {m3};
        fu1_o1          input,  {m4};
        fu1_o2          input,  {m3};
        fu1_t           input,  {m4};
        fu1_r0          output, {m6};
        fu1_r1          output, {m7, m8};
        fu2_o           input,  {m9};
        fu2_t           input,  {m10};
        fu2_r           output, {m14};
        fu3_o           input,  {m27};
        fu3_o1          input,  {m14, m29};
        fu3_o2          input,  {m14, m30};
        fu3_o3          input,  {m14};
        fu3_t           input,  {m15};
        fu3_r           output, {m17};
        fu4_o           input,  {m3, m4};
        fu4_t           input,  {m5};
        fu4_r           output, {m3, m4, m8, m27};
        fu5_o           input,  {m11};
        fu5_t           input,  {m13};
        fu5_r           output, {m11, m12};
        fu6_o           input,  {m3};
        fu6_t           input,  {m6, m7};
        fu6_r           output, {m9};
        fu7_o           input,  {m3};
        fu7_t           input,  {m12};
        fu7_r           output, {m10};
        fu8_o           input,  {m16};
        fu8_t           input,  {m17};
        fu8_r           output, {m3};
        fu9_o           input,  {m4};
        fu9_t           input,  {m5};
        fu9_r           output, {m1};
        fu10_o          input,  {m27};
        fu10_t          input,  {m28};
        fu10_r          output, {m2};
        io1_t           input,  {m31};
        io1_r           output, {m31};
        ri_i1           input,  {m9, m28, m31};
        ri_i2           input,  {m8, m9, m30};
        ri_i22          input,  {m26};
        ri_i3           input,  {m8, m14};
        ri_i32          input,  {m25};
        ri_i4           input,  {m8, m14};
        ri_i42          input,  {m24};
        ri_i5           input,  {m8, m23, m31};
        ri_i6           input,  {m8, m22, m31};
        ri_i7           input,  {m8, m21, m31};
        ri_i8           input,  {m8, m20, m31};
        ri_i9           input,  {m8, m19, m31};
        ri_i10          input,  {m8, m18, m31};
        ri_i11          input,  {m8, m31};
        ri_o1           output, {m4, m5, m11, m13};
        ri_o12          output, {m27, m28};
        ri_o2           output, {m4, m16};
        ri_o22          output, {m28};
        ri_o3           output, {m26};
        ri_o32          output, {m29};
        ri_o4           output, {m25};
        ri_o42          output, {m30};
        ri_o5           output, {m3, m5, m24};
        ri_o6           output, {m4, m5, m10, m15, m27, m28};
        ri_o62          output, {m23};
        ri_o7           output, {m5, m15};
        ri_o72          output, {m22};
        ri_o8           output, {m15};
        ri_o82          output, {m21};
        ri_o9           output, {m15};
        ri_o92          output, {m20};
        ri_o10          output, {m3, m19};
        ri_o11          output, {m11, m18};
        ri_o112         output, {m28};
```

```
        b_i1            input,  {m1, m2};
        pc              input,  {m31};
        ra_i            input,  {m31};
        ra_o            output, {m31};
        trap            input,  {m31};
        ir_1            output, {m31};
}



FunctionUnits
{
        fu1             always, 2, { fu1_o, fu1_o1, fu1_o2 }, fu1_t, { fu1_r0, fu1_r1 }, {ag};
        fu2             always, 3, { fu2_o }, fu2_t, { fu2_r }, {cmul};
        fu3             always, 1, { fu3_o, fu3_o1, fu3_o2, fu3_o3 }, fu3_t, { fu3_r }, {cadd};
        fu4             always, 1, { fu4_o }, fu4_t, { fu4_r }, {add, sub};
        fu5             always, 1, { fu5_o }, fu5_t, { fu5_r }, {add, sub};
        fu6             always, 3, { fu6_o }, fu6_t, { fu6_r }, {ld, st};
        fu7             always, 3, { fu7_o }, fu7_t, { fu7_r }, {ld, st};
        fu8             always, 3, { fu8_o }, fu8_t, { fu8_r }, {ld, st};
        fu9             always, 1, { fu9_o }, fu9_t, { fu9_r }, {eq, gt, gtu};
        fu10            always, 1, { fu10_o }, fu10_t, { fu10_r }, {eq, gt, gtu};
        io1             always, 1, {       },  io1_t,  { io1_r },  {cntlrd, cntlwr};
}

LongImmediate
{
        Registers:
        i1              32, signed, ir_1;
        Control:
        i1 32:{31};
}

RegisterUnits
{
        Integer                 3, {ri_i1}, {ri_o1, ri_o12};
        Integer                 2, {ri_i2, ri_i22}, {ri_o2, ri_o22};
        Integer                 2, {ri_i3, ri_i32}, {ri_o3, ri_o32};
        Integer                 2, {ri_i4, ri_i42}, {ri_o4, ri_o42};
        Integer                 2, {ri_i5}, {ri_o5};
        Integer                 2, {ri_i6}, {ri_o6, ri_o62};
        Integer                 2, {ri_i7}, {ri_o7, ri_o72};
        Integer                 2, {ri_i8}, {ri_o8, ri_o82};
        Integer                 2, {ri_i9}, {ri_o9, ri_o92};
        Integer                 2, {ri_i10}, {ri_o10};
        Integer                 2, {ri_i11}, {ri_o11, ri_o112};

        Boolean                 2, {b_i1};
}

InstructionUnit
{
        JumpLatency     4;
        BoolLatency     1;
        BoolExprSize    1;
        ProgramCounter  pc;
        TrapRegister    trap;
        ReturnAddress   ra_i, ra_o;
}
```

Appendix H

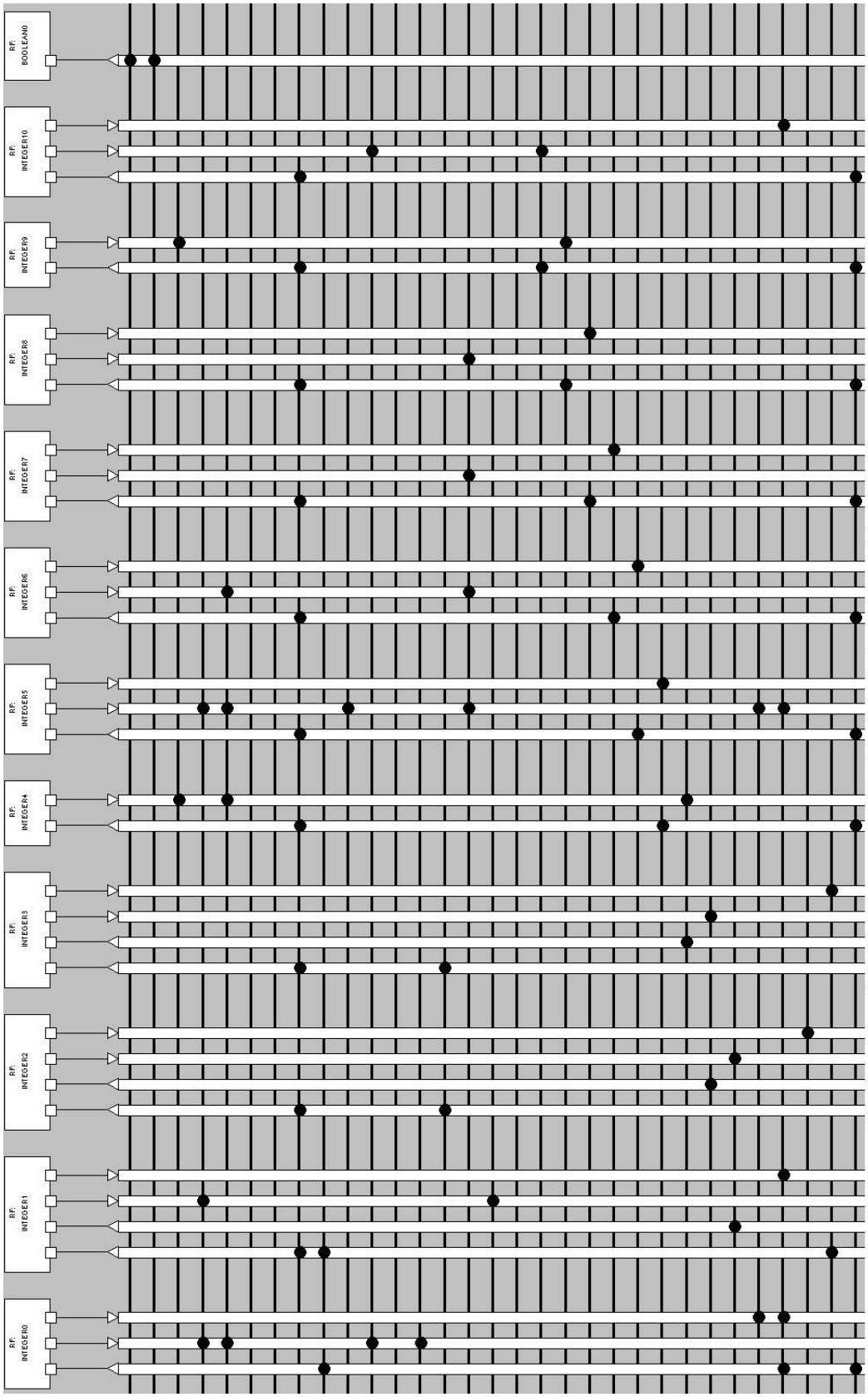
CORE: FUNCTION UNITS



The buses are marked with horizontal lines in the above figure. The uppermost bus is bus1 (m1) and the lowermost bus is bus31 (m31).

Appendix I

CORE: REGISTER FILES



The buses are marked with horizontal lines in the above figure. The uppermost bus is bus1 (m1) and the lowermost bus is bus31 (m31).