Static Instruction Scheduling for Transport Triggered Architectures

Master Thesis

Melvin Richard John

Embedded Systems Group
Computer Science Department
University of Kaiserslautern

Examiner
Prof. Klaus Schneider
Supervisor
M.Sc. Anoop Bhagyanath

Examiner
Prof. Wolfgang Kunz
Electronic Design Automation Group
Electrical and Computer Engineering Department
University of Kaiserslautern

Oct 2016

Declaration

I declare that the enclosed Master thesis has been prepared solely by me without the help of third parties except the support of my supervisor. I have not used sources or means without declaration in the thesis text. Any thoughts from others or literal quotations are clearly marked. The source is always given where I have consulted the published work of others.

Melvin Richard John

————————————-, Kaiserslautern

Abstract

Transport Triggered architecture (TTA) is an exposed datapath architecture, where the compiler is responsible to schedule data movements between the processing units and/or register files, in addition to scheduling computations to the processing units. TTAs are programmed by move instructions that transports data from the output of a processing unit/register file to the input of the same or another processing unit/register file. Execution on processing units is triggered as a side-effect of data transports. Abacus refers to a family of processor architectures that implements MIPS-like reduced instruction sets, used for educational and research purposes. This thesis work implements a code generator that generates parallel TTA code for execution on a given TTA machine, from a given Abacus program. The code generator first translates the Abacus instructions to a sequence of TTA move instructions for an abstract TTA architecture. It then packs the move instructions in every basic block using operation based list scheduling for execution on a given TTA machine. Finally it utilizes the capability of TTA architectures to move data between processing units thus bypassing the use of registers leading to a reduced register file pressure.

Abstract

Transport Triggered Architektur (TTA) ist ein belichteter Datenweg-Architektur, wobei der Compiler ist verantwortlich Datenverkehrs zwischen den Verarbeitungseinheiten zu planen und / oder Dateien zu registrieren, zusätzlich zu Scheduling Berechnungen zu den Verarbeitungseinheiten. TTAs werden durch Bewegungsbefehle programmiert, die Daten aus dem Ausgang einer Verarbeitungseinheit / Registerdatei mit dem Eingang der gleichen oder einer anderen Verarbeitungseinheit transportiert / Registerdatei. Ausführung auf Verarbeitungseinheiten als ein Nebeneffekt der Datentransporte ausgelöst. Abacus bezieht sich auf eine Familie von Prozessorarchitekturen, die MIPS-ähnliche Reduced Instruction Sets, die für Lehr- und Forschungszwecke implementiert. Diese Diplomarbeit implementiert einen Code-Generator, der für die Ausführung auf einem bestimmten TTA-Maschine, von einem bestimmten Abacus Programm parallel TTA-Code generiert. Der Code-Generator übersetzt zuerst die Abacus Anweisungen zu einer Folge von TTA Bewegungsanweisungen für eine abstrakte TTA-Architektur. Es packt dann die Bewegungsbefehle in jedem Grundsatzbetrieb basierend Liste Planung für die Ausführung auf einem bestimmten TTA-Maschine. Schließlich nutzt es die Fähigkeit von TTA Architekturen Daten zwischen Verarbeitungseinheiten so die Verwendung von Registern Umgehung zu bewegen, um eine reduzierte Registerdatei Druck führt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's world is becoming a highly mobile and interconnected hub. The role of embedded systems in this is becoming evident in its increased application in everyday life like consumer electronics, medical devices, industrial robots, automotive and airplanes etc. An embedded system continuously interacts with the environment in which it is embedded in. It reads inputs via sensors, processes the read inputs and provides the processed output via actuators. Many embedded systems are compute intensive demanding a high performance processing unit. At the same time, majority of embedded systems are real-time systems, where certain timing constraints or deadlines must be guaranteed for the correctness of the system. Therefore the processing units are also expected to be time-predictable so that the execution times can be accurately predicted offline, guaranteeing any deadlines.

1.1 Motivation

The processing unit in an embedded system is generally synthesized following classic hardware methodology for a particular application or by following classic software methodology that can handle any application. In classic hardware methodology, the computations in an application is synthesized directly into an application specific integrated circuit (ASIC) or as a soft core in a field programmable gate array (FPGA). To run another application, one has to stop the system and reprogram the FPGA or integrate an entirely new ASIC. In classic software methodology, the computations in an application is compiled to machine code that is then executed by any conventional processor architecture. To run another application, one simply has to compile the application in a host device (generally a desktop computer) and reflash the program memory in embedded device, so changes in hardware is required. Conventional processors can be broadly classified into statically scheduled and dynamically scheduled architectures. In dynamically scheduled processors like superscalar processors, the processor computes data dependencies and schedules instructions to processing units during runtime. Compiler simply has to order the instructions in an application. In statically scheduled processors like very long instruction word (VLIW) processors, compiler schedules instructions to processing units considering data dependencies between these instructions while the hardware simply executes the scheduled instructions. Most conventional architectures are load/store architectures where memory accesses are only allowed via the load or store instructions and all operand values for any computation must reside in the registers in the processor. The load/store architectures usually use reduced instruction set (RISC) instructions.

Within the class of statically scheduled architectures, belong certain exposed datapath architectures. In exposed datapath architectures, the compiler is not only responsible to schedule instructions to processing units, but also takes care of moving values between processing units and/or register files. This enables the compiler to bypass registers in certain situations thus reducing the register file pressure and improving the degree of exploited instruction level parallelism (ILP). This improves the overall efficiency of embedded system. Transport triggered architectures (TTA) [1, 2, 3] fall in this category of processor architectures. In TTA, the processing units are equipped with register ports at their inputs and outputs. All output ports are connected to input ports via an interconnection network. TTAs are programmed by a sequence of move instructions whose semantic is to move a value from an output port to an input port. Execution in processing units is triggered as a side-effect of data transports. Compiler packs independent data move instructions into parallel bundles, which is then executed by the hardware in one step. Furthermore usage of move instructions enables TTA to include processing units that may implement any arbitrary functionality with an arbitrary number of inputs and outputs. Therefore, in addition to a classic software embedded system synthesis technique, TTAs also support a synthesis technique that falls in between the classic hardware and classic software. Most frequently occurring computation patterns could be included as a dedicated processing unit in TTA to improve the performance of embedded system. Moreover static scheduling in TTAs enable easier execution time analysis making it suitable for use in real-time embedded systems [4].

1.2 Thesis Contribution

In this thesis, we generate move program for a given TTA machine from an assembly program that contains a sequence of RISC instructions. Clearly register assignment is already performed by the code generator that produced the RISC instructions. We simply consider the same register assignment in our TTA code generator. Then the compilation of RISC instructions to parallel TTA move code proceeds as follows: First basic control flow analysis is performed where the assembly program is divided into basic block. Then every instruction in a basic block is translated to a sequence of moves for an abstract TTA architecture consisting of one universal processing unit. A basic block of abstract TTA moves is then scheduled on the given TTA machine by using the operation based list scheduling technique. First fit assignment is used to map computations to the processing units and moves to the transport resources. We consider a simple set of buses as transport resources. Finally we utilize the software bypassing capability of TTA architectures, i.e. to bypass register accesses by moving values directly from the output of a processing unit to the input of the same or different processing unit. Experimental results show the performance improvement by basic block scheduling and software bypassing. Specifically we consider the Abacus RISC instruction set [5]. Abacus refers to a family of processor architectures (including single-cycle, pipelined and dynamically scheduled implementations) that implements a particular RISC instruction set. It is developed for both educational and research purposes in the Embedded Systems Chair in Computer Science Department at the University of Kaiserslautern.

The outline of this report is as follows: Chapter 2 explains in detail the Abacus instruction set and then briefly introduces various processor architectures implementing this Abacus instruction set. In Chapter 3, we explain the organization and functionality of TTA architectures followed by the move instructions considered in our compiler. Chapter 4 includes the main contribution of this thesis: design of a code generator that takes an Abacus assembly program and a TTA machine description as its input

and generates a sequence of packed move instructions to run the program in the given TTA machine. TTA machine description format is explained followed by simple translation from Abacus instructions to TTA moves. Then operation based list scheduling of basic block of TTA moves is described followed by inclusion of software bypassing. Chapter 5 describes the implementation of code generator in Java programming language [6] and experimental results. In the final section, we conclude by mentioning important future work.

Chapter 2

Abacus

Abacus [5] is a processor architecture family developed at the Chair of Embedded Systems for purposes of research and use in academia. An overview of the Abacus instruction set and possible hardware architecture implementations is described in the following sections.

2.1 Instruction Set

The Abacus instruction set is similar to the MIPS instruction set [7]. The architecture supports 8 registers $R0 - R7$. The $R0$ register is a special register always holding the value zero. Writing to $R0$ causes no change to its value. There is an overflow register $ovflw$ that holds the 8 most significant bits of any arithmetic/logic (ALU) operation. During divide (DIV) operations, the $ovflw$ holds the remainder of the division. Although the Abacus supports both scalar and vector instructions, our compiler is designed to handle only scalar instructions. Therefore in future references in this report, all instructions unless specified refer to scalar instructions.

The scalar instruction set of the Abacus processor family can be broadly classified into four classes namely *R-type*, *I-type*, *S-type* and *J-type* as shown in Table 2.1. The instruction set supports operations with two input operands and one output operand. Among the two input operands, one operand may be implicit. The implicit operand is either the program counter or the overflow value of a function unit. The Abacus instructions are 16 bits long (x_{15},...,x_0). These instructions may operate on signed or unsigned operands. For signed instructions, the operands are considered as 2's complement numbers. Similarly for unsigned instructions, the operands are encoded as radix-2 numbers.

The *R-type* or the Register type instructions refer to the register-register operations where two input operands are registers on which some operation is performed and the result is stored in a destination register (rd). The two source registers are labeled as left register(rl) and right register(rr). The R-type instructions supported by the Abacus processor family are listed in Table 2.3. To understand the description of the listed instructions refer the Table 2.2.

The *I-type* or the Immediate type instructions refer to the register-immediate operations where one operand is a register and another operand is an immediate value. The immediate value is a 4 bit constant value c. The I-type instructions supported by the Abacus processor family are listed in Table 2.4.

The *S-type* instructions are the operations that have a destination register as its operand. Some S-type instructions may have an additional operand, a 7 bit constant c. The constant is explicitly mentioned as an operand or is implicit as seen in Table 2.5.

	$x_{15}...x_{10}$	$x_9...x_7$	$x_6...x_4$	$x_3...x_1$	x_0
R-type	$opcode$	rd	rl	rr	vector flag
I-type	$opcode$	rd	rl	4 - bit constant c	
S-type	$opcode$	rd	7 - bit constant c		
J-type	$opcode$	10 - bit constant c			

Table 2.1: Classes of Abacus instructions

The *J-type* instructions or the Jump type instruction only has one operand - a 10 bit constant c. This type of operations that are part of the Abacus instruction set are listed in Table 2.6.

Conventions and Registers	
bv2nat(b)	interpret bitvector b as a radix-2 number
bv2int(b)	interpret bitvector b as a 2's complement number
Reg[r]	scalar register r (r $\in$ {0,...,7})
$ovflw$	register for double precision arithmetic
pc	program counter

Table 2.2: Abacus language reference conventions and registers

2.2 Possible Processor Architectures

The Abacus processor family has multiple possible processor architecture implementations:

- Single Cycle
- Pipelined (Stalling and Data Forwarding)
- Dynamically Scheduled (with Out-of-order Execution)
- Statically Scheduled (Very Long Instruction Word (VLIW))
- Vector processor

A brief overview of all the variations of Abacus processor architectures is given in the following. The single cycle Abacus executes exactly one instruction in one cycle. The advantage is that the hardware architecture and compiler design are very simple. The disadvantage is that the processor is very slow as the cycle time is equal to the sum of times that the instruction takes to complete each stage in execution, namely instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MA) and write back (WB). The single cycle Abacus architecture is shown in figure 2.1.

To maximize the utilization of hardware resources across different stages of instruction execution, the pipelined Abacus architecture is used. Here the so called pipeline registers are introduced between each consecutive stage so that at any point of time, independent instructions can reside in any pipeline stage.

Scalar Arithmetic Instructions			
add	rd,rl,rr	add signed	Reg[rd] = bv2int(Reg[rl]) + bv2int(Reg[rr])
addu	rd,rl,rr	add unsigned	Reg[rd] = bv2nat(Reg[rl]) + bv2nat(Reg[rr])
sub	rd,rl,rr	subtract signed	Reg[rd] = bv2int(Reg[rl]) $-$ bv2int(Reg[rr])
subu	rd,rl,rr	subtract unsigned	Reg[rd] = bv2nat(Reg[rl]) $-$ bv2nat(Reg[rr])
mul	rd,rl,rr	multiply signed	Reg[rd] = bv2int(Reg[rl]) * bv2int(Reg[rr])
mulu	rd,rl,rr	multiply unsigned	Reg[rd] = bv2nat(Reg[rl]) * bv2nat(Reg[rr])
div	rd,rl,rr	divide signed	Reg[rd] = bv2int(Reg[rl]) / bv2int(Reg[rr])
divu	rd,rl,rr	divide unsigned	Reg[rd] = bv2nat(Reg[rl]) / bv2nat(Reg[rr])
Scalar Comparison Operations			
slt	rd,rl,rr	set less-than	Reg[rd] = bv2int(Reg[rl]) $<$ bv2int(Reg[rr])
sltu	rd,rl,rr	set less-than unsigned	Reg[rd] = bv2nat(Reg[rl]) $<$ bv2nat(Reg[rr])
sle	rd,rl,rr	set less-than-equal	Reg[rd] = bv2int(Reg[rl]) <= bv2int(Reg[rr])
sleu	rd,rl,rr	set less-than-equal unsigned	Reg[rd] = bv2nat(Reg[rl]) <= bv2nat(Reg[rr])
seq	rd,rl,rr	set equal	Reg[rd] = Reg[rl] == Reg[rr]
sne	rd,rl,rr	set not equal	Reg[rd] = Reg[rl] $\neq$ Reg[rr]
Scalar Bitwise Logic Operations			
and	rd,rl,rr	bitwise and	Reg[rd] = Reg[rl] $\wedge$ Reg[rr]
or	rd,rl,rr	bitwise or	Reg[rd] = Reg[rl] $\vee$ Reg[rr]
xor	rd,rl,rr	bitwise xor	Reg[rd] = Reg[rl] $\oplus$ Reg[rr]
neg	rd,rl	bitwise nor	Reg[rd] = $\neg$Reg[rl]
sft	rd,rl,rr	bitwise shift left	Reg[rd] $\ll$ Reg[rl] by Reg[rr] bits
Scalar Load/Store Instructions			
ld	rd,rl,rr	load word	Reg[rd] = Mem[bv2nat(Reg[rl]) + bv2nat(Reg[rr])]
st	rd,rl,rr	store word	Mem[bv2nat(Reg[rl]) + bv2nat(Reg[rr])] = Reg[rd]

Table 2.3: R-type instructions

Scalar Arithmetic Instructions			
addi	rd,rl,c	add signed immediate	Reg[rd] = bv2int(Reg[rl]) + bv2int(c)
addiu	rd,rl,c	add unsigned immediate	Reg[rd] = bv2nat(Reg[rl]) + bv2nat(c)
subi	rd,rl,c	subtract signed immediate	Reg[rd] = bv2int(Reg[rl]) $-$ bv2int(c)
subiu	rd,rl,c	subtract unsigned immediate	Reg[rd] = bv2nat(Reg[rl]) $-$ bv2nat(c)
muli	rd,rl,c	multiply signed immediate	Reg[rd] = bv2int(Reg[rl]) * bv2int(c)
muliu	rd,rl,c	multiply unsigned immediate	Reg[rd] = bv2nat(Reg[rl]) * bv2nat(c)
divi	rd,rl,c	divide signed immediate	Reg[rd] = bv2int(Reg[rl]) / bv2int(c)
diviu	rd,rl,c	divide unsigned immediate	Reg[rd] = bv2nat(Reg[rl]) / bv2nat(c)
Scalar Load/Store Instructions			
ldi	rd,rl,c	load word immediate	Reg[rd] = Mem[bv2nat(Reg[rl]) + bv2nat(c)]
sti	rd,rl,c	store word immediate	Mem[bv2nat(Reg[rl]) + bv2nat(c)] = Reg[rd]

Table 2.4: I-type instructions

The Abacus architecture supports a 5 stage pipeline i.e. the processor can now complete execution of 5 instructions in the same time where only one instruction is possible with single cycle Abacus.

The pipelined architecture further supports two variations namely stalling and data forwarding. Both the variations resolves data dependencies that occur with the pipelines architecture differently with the

Register Move Instructions		
mov rd,c	move signed constant to register	Reg[rd] = bv2int(c)
ovf rd	move overflow to register	Reg[rd] = ovflw
Branch/Jump Instructions		
bez rd,c	branch if zero	if (Reg[rd] == 0) pc = pc + bv2int(c)
bnz rd,c	branch if not zero	if (Reg[rd] $\neq$ 0) pc = pc + bv2int(c)
jmp rd	jump to Reg. address	pc = pc + Reg[rd]

Table 2.5: S-type instructions

Branch/Jump Instructions			
j	c	jump to immediate address	pc = pc + bv2int(c)

Table 2.6: J-type instructions

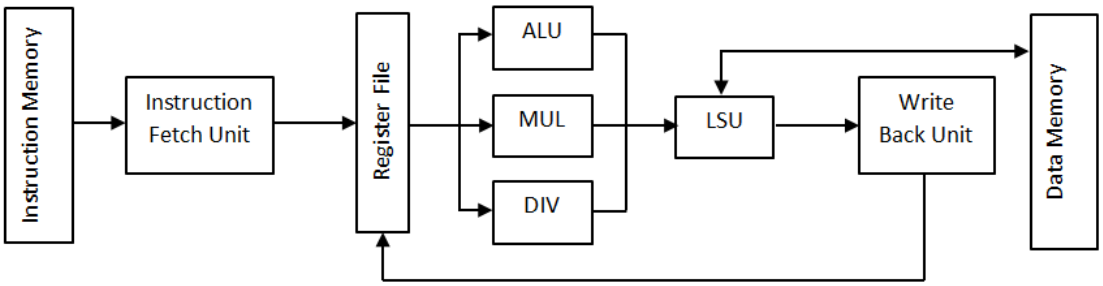


Figure 2.1: Single cycle Abacus architecture

latter providing an improved performance. There are three types of data dependencies

- Read After Write (RAW) : RAW data dependency occurs when an instruction *b* has an input operand that must be *read* from a register and some preceding instruction *a* has to *write* its result to the same register. Then instruction *a* must write to the register before instruction *b* reads from it.
- Write After Read (WAR) : WAR data dependency occurs when an instruction *b* *write* its result to a register and a preceding instruction *a* has to *read* an operand from the same register. Then instruction *a* must read from the register before instruction *b* writes to it.
- Write After Write (WAW) : WAW data dependency occurs when an instruction *b* *write* its result to a register and a preceding instruction *a* also has to *write* its result to the same register. Then instruction *a* must write to the register before instruction *b* writes to it.

Delaying the write back to the final stage resolves both WAW and WAR conflicts in the pipeline. However RAW conflicts still needs to be resolved. In the stalled version of pipelined architecture, RAW conflicts are resolved by simply stalling the pipeline when instruction *b* is in decode stage until the

relevant data is written to the register by instruction *a* in write back stage. In the data forwarded version of pipelined architecture, the relevant data (after execution/memory accesss stage of instruction *a*) is forwarded to the decode stage from the execution/memory access stages. Pipelined Abacus with stalling is shown in Figure 2.2 and the one with data forwarding is shown in Figure 2.3.

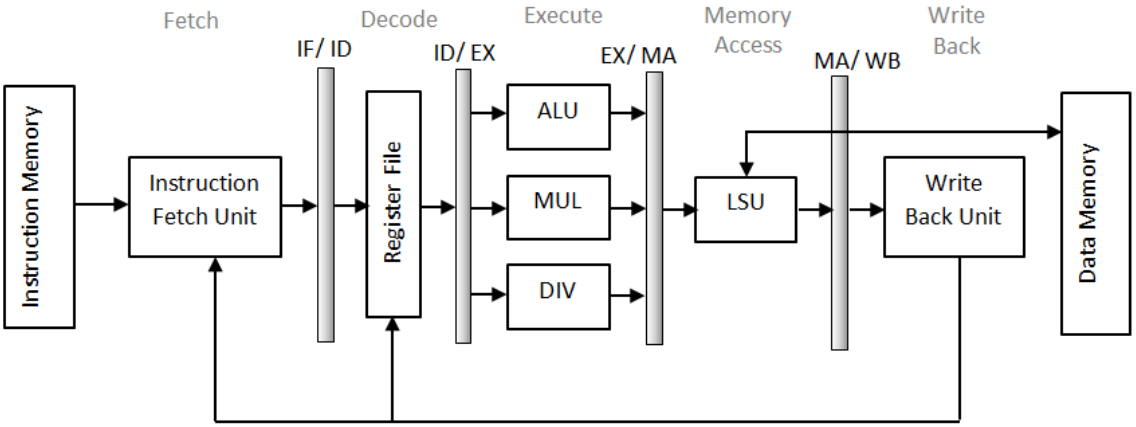


Figure 2.2: Abacus pipeline with stalling

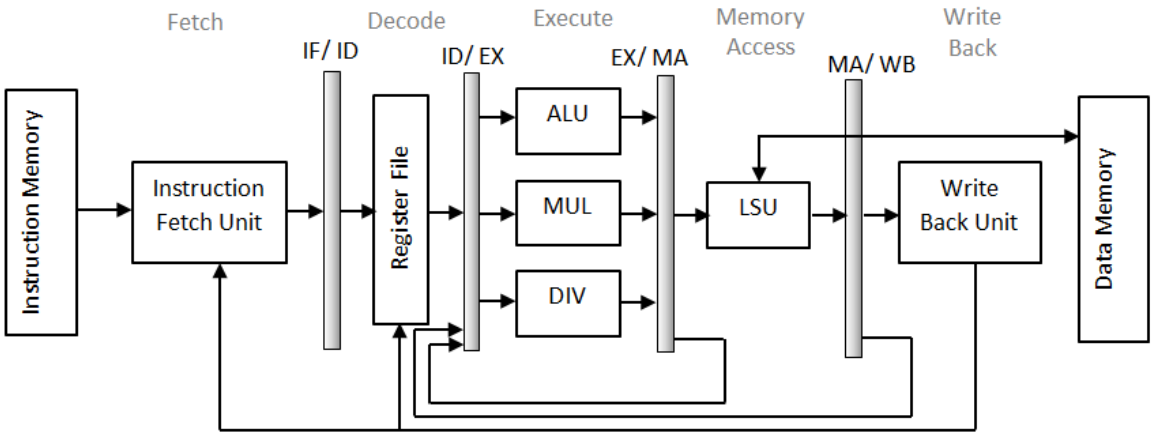


Figure 2.3: Abacus pipeline with data forwarding

In dynamically scheduled Abacus version, Tomasulo's algorithm [8] is used with a single reservation unit. Here the data dependencies between the instructions is analyzed at runtime by the processor using complicated control circuitry involving reservation station, common data bus, forward references and reorder buffer. There is a clear performance improvement compared to pipelined Abacus versions since the function units in the processor are utilized more efficiently by exploiting instruction level parallelism contained in a window of instructions rather than simply considering the instructions one after another in a sequence as in pipelined case. However this performance improvement comes with extremely complex hardware that is difficult to scale.

Finally we have the statically scheduled VLIW Abacus architecture version shown in Figure 2.4.

Here the data dependencies are analyzed statically by the VLIW compiler [9]. The compiler is aware of the number and latencies of function units in the processor. With this information, the compiler packs independent Abacus instructions to parallel bundles (or very long instructions). Each instruction in the packed very long instruction is then dispatched to a predetermined function unit for execution. Compared to the dynamically scheduled version, there is even more possibility to exploit instruction level parallelism since the compiler has access to all instructions to form parallel bundles while the processor only exploits parallelism within a window of instructions that fit into the reservation station. Of course the dynamically scheduled Abacus version is capable of utilizing the information that is available only at runtime (like memory reference disambuigation) to exploit parallelism, while the statically scheduled version cannot have access to this information.

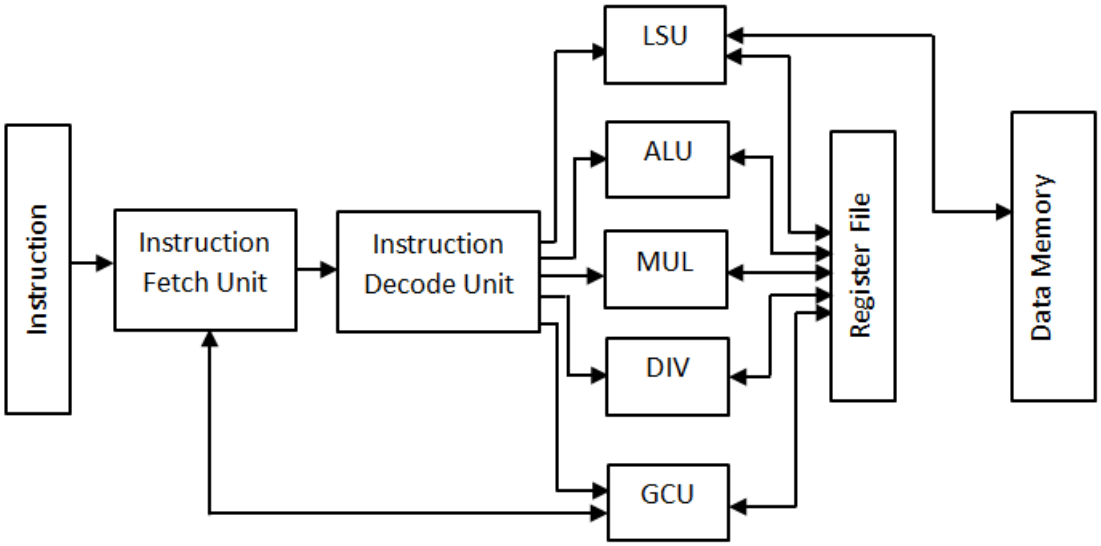


Figure 2.4: Abacus VLIW

Chapter 3

Transport Triggered Architectures

The Transport Triggered Architecture (TTA) [1, 2, 3] is an exposed datapath architecture. The compiler is not only responsible to map operations to function units, but also controls the movement of data between these function units and/or register files. This is an extreme case of static scheduling. The execution of function units is triggered as a side-effect of data transports, hence the name Transport Triggered Architecture. In conventional or Operation Triggered Architectures, function unit execution is explicitly triggered by operations. This section introduces the TTA architecture and describes the instruction set of TTA that we have considered in this work.

3.1 Architectural Details

As shown in Figure 3.1, a TTA consists of function units, register files and main memory that are interconnected using any network. Function units have registers at their input and output ports. TTA is programmed by a sequence of packed *move* instructions whose semantic is to transport a value from the output of a function unit/register file to the input of the same or different function unit/register file. While the move instructions are generated by the compiler, they are executed by the interconnect network. All moves in a packed move instruction set is executed in one cycle by the interconnect network. In other words, TTA is an exposed datapath architecture that exposes its datapath to the compiler. This enables the compiler to directly move values from the output of a function unit to the input of the same or different function unit bypassing use of registers to store intermediate values unless required. This leads to a reduced register pressure on the register files in TTA contrary to VLIW architectures, where both register pressure and register port requirement is high since the reduced instruction set (RISC) instruction format demands that each instruction read its operand(s) from register(s) and write its result to a register. This is one major drawback of VLIW architectures since high register port requirement complicates wiring in a register file adversely affecting its scalability..

Although any network may be used as interconnection network in TTAs, a simple set of buses is commonly used. Buses are connected to the ports of function units and register files using sockets. Figure 3.2 shows a partially interconnected where only some buses are connected to some ports. While in the case of fully interconnected TTAs, all ports are connected to all buses. We only support fully interconnected TTAs in our compiler design explained in the next chapter. Usage of move instructions allows TTA to have any arbitrary function units implementing any functionality with an arbitrary number of inputs and

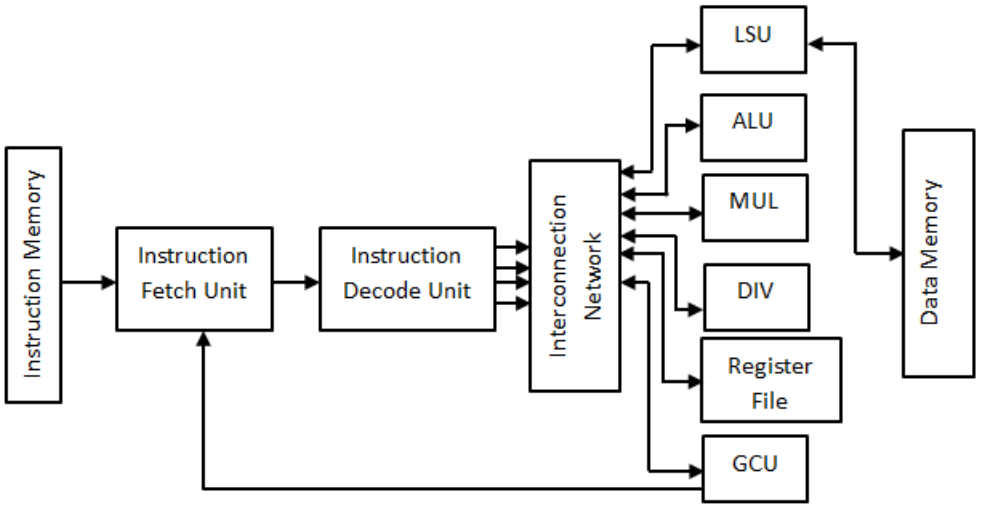


Figure 3.1: A transport triggered architecture processor

outputs. For simplicity, only two input one output function units are shown in the figures here. This makes TTA an application-specific architecture where frequently appearing computations in applications can be implemented as function units in TTA to extract better performance. Doing the same in case of RISC architectures will require change in instruction set and consequently in implementation of different stages in processor.

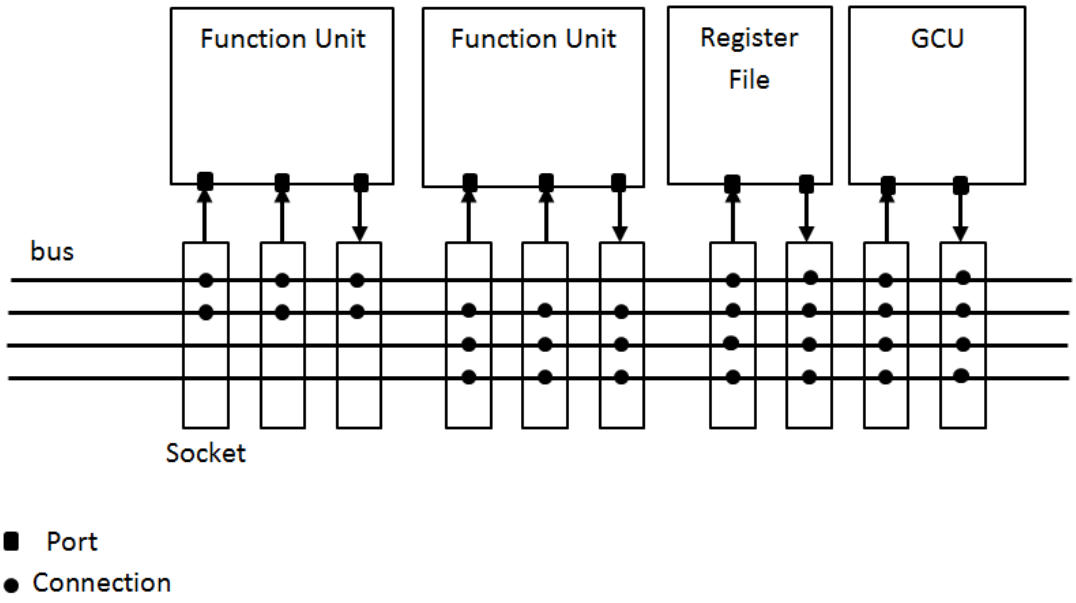


Figure 3.2: A partially interconnected transport triggered architecture with a set of buses as interconnect network

One of the input ports in every function unit is designated as a trigger input port *tr*, while the other input ports are called operand input ports *op*. The function unit execution commences when data is transported to its trigger port. Once triggered, the function unit reads all input port values, executes the function and writes the result to the output port. Therefore for any particular operation, the compiler must ensure that all the operand ports receive the data before data is transported to the trigger port. Not all function units may have the operand input port, for example, a function unit implementing simple negation operation. A function unit is said to be busy while executing its function and free otherwise. In our TTA compiler design, only two input one output function units are supported that may implement one or more functionalities directly encoded in the Abacus instruction set. However there maybe a second output port to capture the most significant bits in case of an overflow in arithmetic operations or the remainder in the case of division.

Note that there is a global control unit (GCU) in the set of function units in Figure 3.1 to handle the control flow in the TTAs. To start fetching the next instruction bundle from a different address, simply move the target address value to the GCU input port. Both conditional and unconditional branches are supported as described in the following section.

3.2 Instruction Set

As already explained, TTA is a one instruction set processor architecture containing only move instructions. For the TTA considered in our compiler design, the instruction set is shown in table 3.1. MOV, LOAD and BRANCH are simply different flavours of the move instruction set. Although the instruction set itself is small, it covers all the different types of operations a TTA processor is expected to perform.

TTA instruction set	
Instruction Set	Description
MOV	moves data from *source* port to *destination* port
LOAD	moves *constant* value to *destination* port
BRANCH	branches to location given by a *constant* value if *predicate* is set
NOP	no operation

Table 3.1: TTA instruction set

The MOV instruction specifies a source port address and a destination port address and its semantic is to move a value from the source port (output port of any function unit/register) to the destination port (input port of any function unit/register). The LOAD instruction specifies an immediate (constant) value and a destination port (input port of any function unit/register) and its semantic is to move the constant value to the specified destination port. The BRANCH instruction specifies a predicate (output port of any function unit/register or negation of output port of any function unit/register) and an immediate (constant) value. Its semantic is to move/ write the constant value to the program counter (PC residing in the GCU) if the predicate is true. However if the predicate is false, PC will be written with the constant value unconditionally. There is also a NOP instruction that is used when the compiler cannot schedule any relevant moves in a clock cycle.

Chapter 4

Compiler Design

An overview of the design of our TTA Compiler is shown in Figure 4.1. The compiler accepts TTA hardware description and the Abacus assembly (.asm) code as inputs. The TTA translator block simply translates every Abacus assembly instruction to a set of one or more TTA move instructions assuming a generic TTA machine containing only one arithmetic logic unit (alu) and one load store unit (lsu). That is, the translator block does not take into consideration the TTA hardware description and assumes that all the instructions are mapped to one alu and lsu.

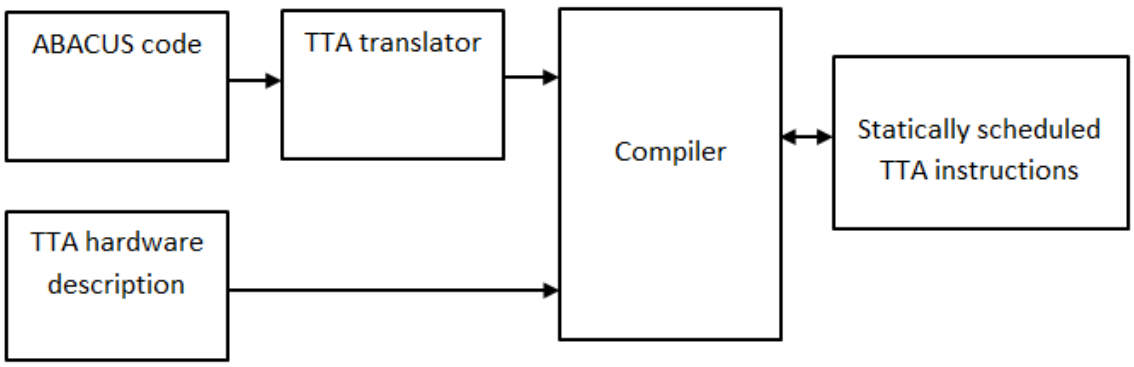


Figure 4.1: TTA compiler design overview

The TTA compiler internally is made up of a basic block generator, data dependency graph generator, a TTA hardware description file reader and most importantly a scheduler. The basic block generator is explained in Section 4.3. The TTA hardware description file reader parses the architecture definition file(*.adf) and the function unit definition files (*.fudf) and stores them internally in a TTA hardware description data structure that also include the connection details of various Function units, register files and memory to the interconnection network of buses. This is explained in detail in the next section. Finally, the scheduler generates the parallel TTA code taking into consideration the hardware resource restrictions as explained in Section 4.3. The final Section 4.4 of this chapter describes how the software bypassing capability (ie the ability to move values from the output of function unit to the input of same or different function unit without having to store the value in a register) of TTA hardware is utilized in our compiler.

4.1 TTA Machine Description

The TTA hardware description file reader parses the hardware description and creates the necessary data structures storing details regarding function units, register files, memory and their interconnection. Basic format checking is also performed. The files are expected to be in a specific format as given in the examples in Appendix A. The file format in a way enforces the TTA architecture structural details including the details of structure of individual components in the architecture. There are two types of TTA hardware description files in general. First, the Function Unit Interface (FUI) description files. They are identified with .fudf (Function Unit description file) extension. Each function unit contains one .fudf file. They describe the functionalities supported by that function unit and also the input output ports. They are named after the individual function unit, for example: Adder.fudf. Second, the overall TTA architecture definition file identified by the .adf (architecture definition file) extension. Only one architecture definition file exists for one TTA machine description. It describes the number of buses in TTA and the connections of these buses to the function unit/register file ports via input and output sockets. Also the number and types of function units and register files is given. All the hardware description files are encoded in xml (extensible markup language) format[10].

4.1.1 Transport Buses

The buses are the interconnects (assumed in our compiler, although TTAs may use any interconnection network) between the function units, the register files and the main memory. Any bus is described by two characteristics shown in Table 4.1. All buses are unique and have a unique name. The architecture definition file contains the description of the buses.

Characteristics of TTA buses	
name	unique name of the bus
width	bus width in number of bits

Table 4.1: Bus characteristics

4.1.2 Sockets

Sockets enable interconnection between buses and function unit ports as shown in Figure 3.2. Our compiler design only considers fully interconnected TTA architecture where all ports are connected to all buses. Any socket has three basic characteristics as shown in Table 4.2. All sockets are unique and have a unique name. The architecture definition file contains description of sockets too.

Characteristics of sockets in the TTA network	
name	unique name of the socket
buses	list of bus names that are connected to socket
ports	list of Function unit port names that are connected to the socket

Table 4.2: Socket characteristics

4.1.3 Ports

Ports are part of the function unit interface (FUI). Ports connect function units (FU) to buses through sockets. There are three characteristics that describe a port. They are shown in table4.3. In a given function unit, every port is unique and has a unique name. A port is differentiated as a trigger port if data arrival at that port 'triggers' the function unit execution.

Characteristics of input/output ports	
name	unique name of the socket
is Trigger	whether a port is a trigger port
sockets	list of sockets that are connected to the port

Table 4.3: Port characteristics

4.1.4 Function Unit Interfaces

Function Unit Interfaces (FUIs) defines input/output ports and input/output sockets of the TTA function units like alu, $adder$, etc. An example of the FUI of an adder unit is shown in Listing 4.1. Some FUIs have other FUIs embedded in them. This is represented by the simple block diagram in Figure 4.2 showing an alu ALU having an $adder$ *ADD* and a subtractor *SUB* modules. A function unit interface is instantiated as a function unit in the TTA architecture definition .adf file. For example, a TTA processor can have two alu units that are both instances of the defined alu function unit interface.

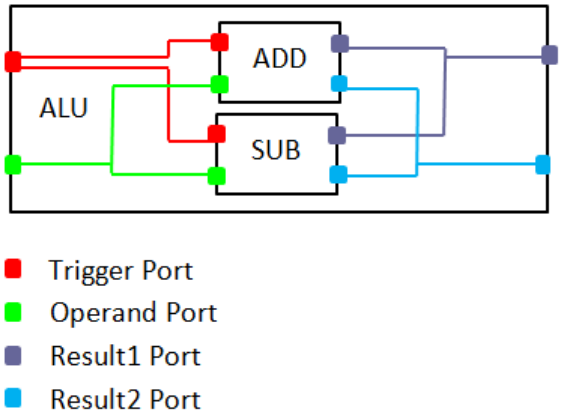


Figure 4.2: Function unit interface of alu unit

All function unit interfaces have a trigger port. The input arriving at this port is supposed to be the final input operand for any specific operation executed by that function unit since data arrival at the trigger port forces the function unit to read all input values and begin its execution. Any TTA function unit interface that includes other function unit interfaces as part of them usually have its ports connected to a multi-address input socket. The multi-address input socket basically enables the selection of the internal computation (similar to *opcode* in Abacus instruction set) triggered in that function units. For example: the alu function unit interface is shown in Listing 4.2 .

Listing 4.1: Adder.fudf

```
<?xml version="1.0"?>
<FunctionUnitInterface>
   <Ports>
      <Port trigger="true">op1</Port>
      <Port>op2</Port>
      <Port>result</Port>
      <Port>status</Port>
   </Ports>
   <Sockets>
      <InputSocket port="op1">
         <AddressOffset>0</AddressOffset>
      </InputSocket>
      <InputSocket port="op2">
         <AddressOffset>1</AddressOffset>
      </InputSocket>
      <OutputSocket port="result">
         <AddressOffset>2</AddressOffset>
      </OutputSocket>
      <OutputSocket port="status">
         <AddressOffset>3</AddressOffset>
      </OutputSocket>
   </Sockets>
</FunctionUnitInterface>
```

Listing 4.2: Alu.fudf

```
<?xml version="1.0"?>
<FunctionUnitInterface>
   <Ports>
      <Port trigger="true">op1</Port>
      <Port>op2</Port>
      <Port>result1</Port>
      <Port>result2</Port>
      <Port>status</Port>
   </Ports>
   <Sockets>
      <MultiAddressInputSocket port="op1">
         <AddressOffset name="add">0</AddressOffset>
         <AddressOffset name="substract">1</AddressOffset>
         <AddressOffset name="multiply">2</AddressOffset>
         <AddressOffset name="unsignedDivide">3</AddressOffset>
         <AddressOffset name="signedDivide">4</AddressOffset>
         <AddressOffset name="shiftLeft">5</AddressOffset>
         <AddressOffset name="shiftRight">6</AddressOffset>
         <AddressOffset name="not">7</AddressOffset>
         <AddressOffset name="and">8</AddressOffset>
         <AddressOffset name="or">9</AddressOffset>
         <AddressOffset name="xor">10</AddressOffset>
         <AddressOffset name="equal">11</AddressOffset>
         <AddressOffset name="unsignedLess">12</AddressOffset>
         <AddressOffset name="unsignedLessEqual">13</AddressOffset>
         <AddressOffset name="less">14</AddressOffset>
```

```
27          <AddressOffset name="lessEqual">15</AddressOffset>
28          <AddressOffset name="unsignedBigger">16</AddressOffset>
29          <AddressOffset name="unsignedBiggerEqual">17</AddressOffset>
30          <AddressOffset name="less">18</AddressOffset>
31          <AddressOffset name="lessBigger">19</AddressOffset>
32       </MultiAddressInputSocket>
33       <InputSocket port="op2">
34          <AddressOffset>20</AddressOffset>
35       </InputSocket>
36       <OutputSocket port="result1">
37          <AddressOffset>21</AddressOffset>
38       </OutputSocket>
39       <OutputSocket port="result2">
40          <AddressOffset>22</AddressOffset>
41       </OutputSocket>
42       <OutputSocket port="status">
43          <AddressOffset>23</AddressOffset>
44       </OutputSocket>
45    </Sockets>
46    <Dependencies>
47       <Module>Arithmetic.Adder</Module>
48       <Module>Arithmetic.Comparator</Module>
49       <Module>Arithmetic.Multiplier</Module>
50       <Module>Arithmetic.SignedDivider</Module>
51       <Module>Arithmetic.UnsignedDivider</Module>
52       <Module>Arithmetic.Subtractor</Module>
53       <Module>Bitwise.ShiftLeft</Module>
54       <Module>Bitwise.ShiftRight</Module>
55       <Module>Bitwise.And</Module>
56       <Module>Bitwise.Or</Module>
57       <Module>Bitwise.Xor</Module>
58       <Module>Bitwise.Not</Module>
59    </Dependencies>
60 </FunctionUnitInterface>
```

4.1.5 Function Units

Function Units are defined in the architecture definition file. It is a unique instance of a Function Unit Interface. There can be one or more Function Units having the same Function Unit Interface. A Function Unit has a unique name but the other characteristics it inherits from a function unit interface. The interconnects of sockets are also uniquely defined. A portion of the TTA description file containing the hardware description of the ALU function unit is listed in Listing 4.3.

The architecture definition file contains the details of the interconnections between the Function Unit ports and the sockets. As different function units of the same function unit interface are unique they can have different delays. The delay of a Function Unit is independent of the FUI it is based on.

Listing 4.3: TTA.adf

```
1    </function-unit>
2    <function-unit name="Alu">
3       <delay>3</delay>
```

```
4       <module>Arithmetic.Alu</module>
5       <port name="op1">
6           <connects-to>AluOp1</connects-to>
7       </port>
8       <port name="op2">
9           <connects-to>AluOp2</connects-to>
10      </port>
11      <port name="result1">
12          <connects-to>AluResult1</connects-to>
13      </port>
14      <port name="result2">
15          <connects-to>AluResult2</connects-to>
16      </port>
17      <port name="status">
18          <connects-to>AluStatus</connects-to>
19      </port>
20  </function-unit>
```

4.1.6 Memory

The memory module is designed similar to any function unit interface of the TTA processor. It has a trigger port which accepts the address of the memory to be accessed. The value can be read from or written to memory. This is described by a named address offset shown in the Table 4.4.

Memory socket address offset	
name	address offset value
read	0
write	1

Table 4.4: Memory socket address offset

4.1.7 Register File

The register files are like any other function unit in TTA, but with multiple ports so that any number of registers can be read from and written to in the same clock cycle. Our TTA hardware contains 32 registers. For this reason, The TTA hardware description files include for the Register files (RFs), a parallel multi address socket with 32 registers having unique names from $register0$ to $register31$. Meanwhile the Abacus instruction set allows to encode only up to 8 registers. As already explained, we simply take the register allocation performed already by the Abacus code generator and only perform TTA move code generation and scheduling in our TTA compiler Therefore for simplifying the compiler design, a one to one mapping of the 8 Abacus Registers ($R0$ to $R7$) to the first eight registers of the TTA processor is used. The register naming convention of the TTA code later discussed in this document will use the same naming conventions of the Abacus assembly code, i.e $R0$ will implicitly mean $register0$, $R1$ will implicitly mean $register1$ and so on.

4.2 Abacus Instruction to TTA Moves Translator

As briefly explained earlier, the TTA translator block in Figure 4.1 simply translates every Abacus assembly instruction to a set of one or more TTA move instructions assuming a generic TTA machine containing only one arithmetic logic unit (alu) and one load store unit (lsu). That is, the translator block does not take into consideration the TTA hardware description and assumes that all the instructions are mapped to one alu and lsu.Consider the below Abacus immediate addition instruction for example:

Abacus instruction:

addi $R2, R2, 1$

translated TTA moves:

LOAD $1, alu.addi.op$;

MOV $R2, alu.addi.tr$;

MOV $alu.result1, R2$;

The Abacus instruction adds the immediate value 1 to the value in register $R2$ and stores the result in register $R2$. Same functionality is reflected in the generic TTA hardware using three move instructions. First move the immediate value 1 to the operand input of alu. Then move the content of register $R2$ to the trigger input and finally, move the result of alu addition from the output port to register $R2$. The TTA translator block consists of an Abacus code reader that reads the Abacus code, verifies the format and translates the code to the TTA move instructions. Few more example translations are given in Table 4.5.

Abacus instructions	TTA move instructions

addu rd, rl, rr	MOV $rl, alu.addu.op$;
	MOV $rr, alu.addu.tr$;
	MOV $alu.result1, rd$;

mov rd, c[1]	LOAD c, rd;

ld rd, rl, rr	MOV $rr, alu.add.op$;
	MOV $rl, alu.add.tr$;
	MOV $alu.result1, lsu.read$;
	MOV $lsu.result, rd$;

bnz rd, c[1]	LOAD $0, alu.seq.op$; MOV $rd, alu.seq.tr$; BRANCH $!alu.result1, c$;

Table 4.5: Abacus instruction to TTA move instructions conversion

An Abacus instruction is translated to one or more TTA move instructions depending on the operation performed. Any operation in Abacus must be mapped to a function unit having two inputs. If an Abacus instruction has only one input operand then the second operand is an implicit operand. For example, in the case of jump operation, the program counter is the implicit input operand as shown in Table 2.6. A function unit in TTA has two input ports i.e. an operand port and a trigger port. The operands for an Abacus instruction n are mapped as TTA move instructions to these input ports, namely n_{op} and n_{tr} respectively. All function units generate at least one result, which is moved to the destination of the corresponding Abacus instruction. Let the move instruction n_{res1} denote this. Some function units generate an overflow value as an additional result. Let n_{res2} denote the move instruction associated with

[1] c is a constant value, which is a line number in case of bnz instruction, that the program counter must be updated with, if the branching condition is satisfied.

this additional result. The dependency among these moves, namely $n_{op}, n_{tr}, n_{res1}, n_{res2}$, is shown in Figure 4.3. This states that the operand move must be scheduled before or in the same clock cycle in which the trigger move is scheduled. Furthermore the result moves must be scheduled at least δ clock cycles after the scheduling of trigger move, where δ is the latency associated with that function unit to which the operation is mapped. The dotted lines in the figure indicate that n_{res2} move instruction is not always generated by the translator block.

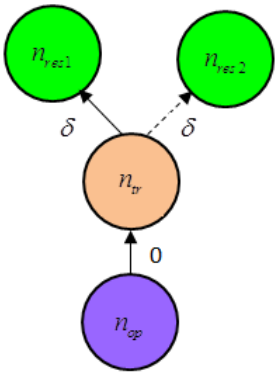
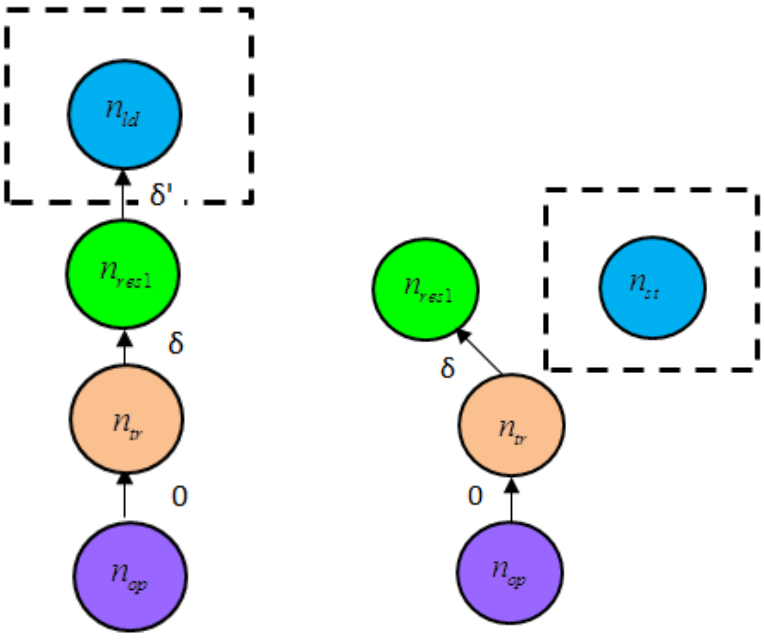


Figure 4.3: Dependency graph of TTA move instructions corresponding to an non-memory access Abacus instruction



(a) Memory load (b) Memory store

Figure 4.4: Dependency graph of TTA move instructions corresponding to a memory access Abacus instruction (load and store)

While Figure 4.3 shows the move instructions corresponding to an Abacus instruction that does not access memory, Figure 4.4 shows the move instructions corresponding to an Abacus instruction that

accesses memory, specifically load instruction in Figure 4.4a and store instruction in Figure 4.4b. Firstly note that the additional (overflow) result move n_{res2} does not exist in case of memory access instruction translations. The operand move (n_{op}) and the trigger move (n_{tr}) triggers the function unit to execute addition to evaluate the effective memory address. The evaluated address value is transported to the address port of the load store unit (lsu) by the result move n_{res1}. In case of load Abacus instruction, the n_{ld} move transports the value loaded from the memory and residing at the output port of lsu to the destination register of the load instruction. Clearly δ' is the latency of a memory read performed by the lsu. Similarly in case of Abacus store instruction, the n_{st} move transports the value from the destination register of the store instruction to the data port of the lsu unit. Memory write operation then writes this data to the target memory address. Both n_{ld} and n_{st} are shown in dotted box since these moves are associated with the lsu unit and not the function unit that evaluates effective memory address.

4.3 Basic Block Scheduler

A basic block *bb* is a set of instructions with a single entry and a single exit point. Given an Abacus assembly code, the basic block generator *BBG* module in the compiler block generates the set of basic blocks. The BBG determines the start of a basic block as follows:

1. first instruction line of code **or**
2. instruction line has a label to which some instruction will jump to **or**
3. instruction line has a preceding branch or jump instruction

Consider the Abacus code in Listing A.1 in Appendix A to multiply two vectors. The move code after translation is shown in Listing A.2. The sync Abacus operation synchronizes the values in the cache with the values in the main memory. As we do not consider use of caches in TTA, we simply translate the sync instruction to TTA NOP instruction. The Abacus instructions (constituting the Abacus program) are then split into basic blocks. For the vector multiply example, the Abacus assembly program is split into five basic blocks by the *BBG* module, namely `BB0` to `BB4`. Along with identifying the basic blocks, the control flow among these basic blocks is also identified. The resulting control flow graph for the vector multiply example is shown in Figure 4.5.

Next, data dependencies (RAW, WAR and WAW) between Abacus instructions within a basic block is identified and the resulting data flow graph is constructed for each basic block. For the vector multiply example, these data flow graphs for basic blocks `BB0` to `BB4` are shown in Figures 4.6 to 4.10 respectively. Note that the Abacus instruction are identified by the numbers corresponding to these instructions commented in the move code in Listing A.2. Data flow graph is a directed acyclic graph, where an as soon as possible (ASAP) schedule classifies the nodes into levels. Data flow from the leaf nodes (nodes at level 0) to nodes at level 1 to nodes at level 2 and so on. The leaf nodes represent the load instructions and the non-leaf nodes represent some computations.

While Figure 4.7 shows the data flow graph of basic block `BB1` of the Abacus program, Figure 4.11 shows the data dependency graph of basic block `BB1` of the move code. We use the notation introduced in previous section to show the dependencies among TTA move instructions corresponding to each Abacus instruction. Also dependencies among move instructions associated with different Abacus instructions is shown, that is derived from the data flow graph of the same basic block `BB1` of Abacus instructions.

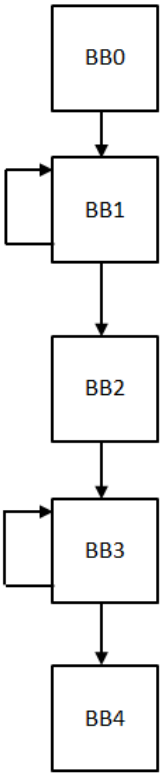


Figure 4.5: Control flow graph of the Abacus program vectorMultiply.asm

Figure 4.6: Data flow graph of basic block `BB0`

For scheduling of move instructions in a basic block, we utilize the well known list scheduling technique. In the context of instruction scheduling, list scheduling technique schedules a set of dependent instructions in clock cycles. The instructions are scheduled in the order determined by the dependencies among them. That is, an instruction is scheduled when all its predecessor instructions are scheduled. Such an instruction is referred to as a *ready* instruction. There are two basic variations of list scheduling: First where the scheduler tries to schedule as many instructions as possible in current clock cycle. When no more instructions can be scheduled in the current clock cycle, it moves on to schedule instructions in the next clock cycle. Second, where the scheduler every time picks a ready to schedule instruction and schedules it in the earliest possible clock cycle. Then moving on to the next ready instruction and so on. We implement the second list scheduling technique in our TTA compiler.

Algorithm 1 describes the list Scheduling technique used in TTA compiler. To this end, list scheduling is applied on the Abacus instructions in the basic block and not directly on the TTA move instructions. Once an Abacus instruction is selected for scheduling, all the move instructions associated with (or translated from) this Abacus instructions are scheduled in one shot using ScheduleAbacusInstr() function. In case we directly use the list scheduling on TTA moves, it is highly probable that different moves (operand, trigger and result) associated with an Abacus operation gets scheduled in clock cycles far

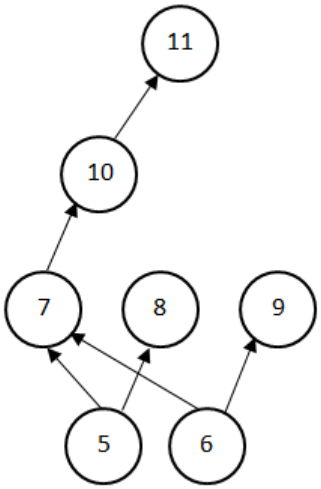


Figure 4.7: Data flow graph of basic block `BB1`

Figure 4.8: Data flow graph of basic block `BB2`

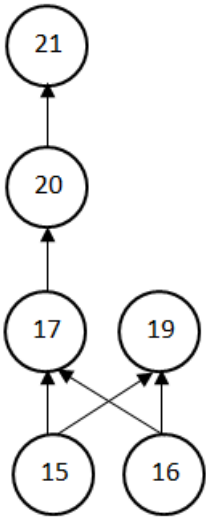


Figure 4.9: Data flow graph of basic block `BB3`

away from one another. This will make the corresponding function unit *busy* for a longer period of time unnecessarily. Scheduling all TTA moves associated with an Abacus operation in one shot ensures that the moves are scheduled close to one another, freeing the function unit as soon as possible so that other operations may be mapped to that function unit.

List scheduling now proceeds as follows: Abacus Instructions (Operations) with no data dependencies are ready to be scheduled and are added to the *ready* set. From this *ready* set of instructions, an instruction

Figure 4.10: Data flow graph of basic block `BB4`

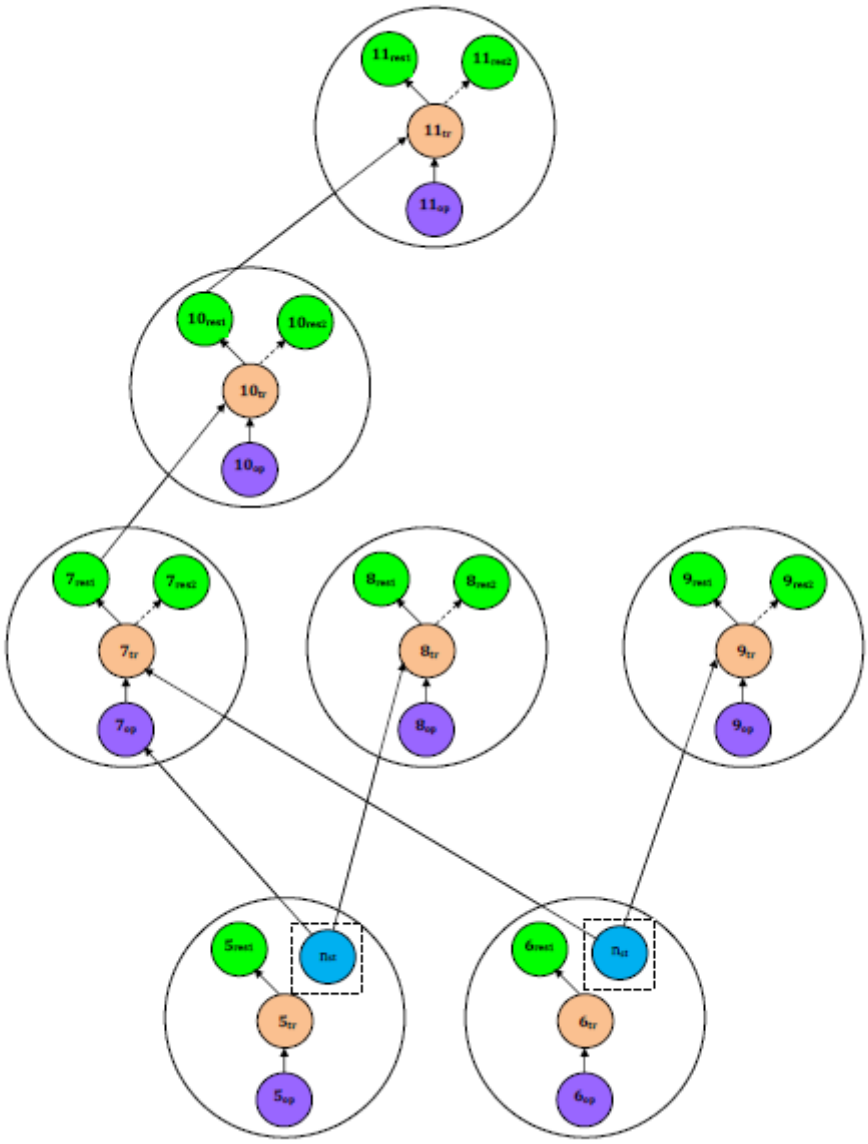


Figure 4.11: Data dependency graph of TTA moves in basic block `BB1`

is selected for scheduling using the SelectAbacusInstr() functtion. After all Abacus instructions in the current *ready* set are scheduled, they are marked as *scheduled* and all instructions that are dependent on the *scheduled* set of instructions is added to the next set of *ready* instructions. The basic block to be scheduled is represented by $G = (V, E)$ where V is the set of all Abacus instructions in the basic block and E is the set of directed edges showing the data dependencies of these instructions v where $\{v \in V\}$. The directed edge $E_{i,j}$ between two instructions v_i and v_j where $v_i, v_j \in V$ shows that the instruction v_j is dependant on instruction v_i. This algorithm is used on every basic block of the Abacus code.

Algorithm 1: List scheduling of an Abacus program basic block represented by $G = (V, E)$

$ready \leftarrow \{v \in V \mid \neg\exists(v_i, v) \in E,\ v_i \in V\}$
if $\exists v_i \in ready \mid i == 0$ **then**
  $clk \leftarrow 0$
else
  $clk \leftarrow clk\ of\ last\ scheduled\ move\ instruction\ of\ previous\ basic\ block + 1$
end
$scheduled \leftarrow \emptyset$
while $scheduled \neq V$ **do**
  while $ready \neq \emptyset$ **do**
    v = SelectAbacusInst($ready$)
    ScheduleAbacusInst(v, clk)
    $scheduled \leftarrow scheduled \cup \{v\}$
    $ready \leftarrow \{ready \setminus v \mid v \in scheduled\}$
  end
  $clk \leftarrow clk\ of\ last\ scheduled\ move\ instruction$
  $ready \leftarrow \{v \in V \setminus scheduled \mid \forall(v_i, v) \in E,\ v_i \in scheduled\}$
end

A simple algorithm implementing the SelectAbacusInstr() function is given in Algorithm 2. It selects an instruction v where $v \in V$, from a set of $ready$ instructions. If there is an instruction has a label (in other words, an instruction to where some instruction branches to in the Abacus program), this instruction is first selected from the ready set of instructions so that we could keep associating the label information with this instruction. If no such instruction exists in the *ready* set, the algorithm simply selects the first instruction that have the least delay. To determine the delay, simple first-fit assignment is used to check for any function unit availability that can be mapped to the considered Abacus instruction. Note that function unit assignment is not explicitly stated in the algorithm.

Algorithm 2: Selecting an Abacus instructions from a set of $ready$ instructions

if $hasLabel(v_i) == true\ \{v_i \in ready\}$ **then**
  return v_i
else
  return $\{v_i \in ready \mid \delta_{v_i} < \delta_{v_j} \forall v_j \in \{ready \setminus v_i\})\}$
end

Algorithm 3 describes the scheduling of TTA instructions of a selected Abacus instruction (Operation) v where $v \in ready$, given a clock cycle clk. Clearly only as many move instructions can be scheduled in a single clock cycle as the number of buses in TTA processor. For every move instruction, if no more slots are available in the current clock cycle, we simply keep incrementing clk until an empty slot is found. TTA moves corresponding to an Abacus instruction is composed of operand input moves, one trigger input move, result moves and memory load data/memory store data moves (only in case of memory access Abacus instructions). All operand input moves (n_{op}) are scheduled before scheduling the trigger input move (n_{tr}). Next the result moves (that transports the output produced by the function unit) is scheduled at least δ clock cycles after scheduling n_{tr}, where δ is the latency of the function unit to which the Abacus instruction is mapped to. If the Abacus instruction write to main memory, then the TTA move

that transports the register content (to be stored to main memory) to the data input of lsu unit, i.e n_{st} is scheduled in the next available slot. However in case the Abacus instruction reads from main memory, we wait for δ' clock cycles before scheduling the TTA move n_{ld} that transports the loaded data from lsu output to the destination register. δ' is the memory load latency. Note that, although not stated in the algorithm, a function unit (alu or lsu) is marked as $busy$ in that clock cycle, where the first operand move is scheduled and the function unit is marked as $free$ in the clock cycle where the final result move is scheduled.

4.4 Software Bypassing

The term software bypassing refers to bypassing the use of registers (and consequently the main memory) by transporting values from the output of a function unit to the input of the same/different function unit. Clearly TTA architecture provides its compiler this opportunity. Software bypassing reduces code size, improves performance and reduces the register file pressure in the hardware. To apply software bypassing, our TTA compiler simply passes through the schedule of packed move instructions taking note of consequent occurrences of a result move to a register followed by an operand/trigger move that transports this register content to the input of any function unit. For example, if the move instruction $alu1.result \rightarrow R1$ in clock cycle clk is followed by another move instruction $R1 \rightarrow alu2.addiu.op$ in clock cycle $clk + i$, then we first replace the latter move instruction by $alu1.result \rightarrow alu2.addiu.op$. Second, we try to schedule this instruction in a clock cycle as near as possible to clk. This way reading the register file $R1$ is bypassed. Also depending on the packed move instructions, we improve the performance. Furthermore if the value (from $alu1.result$) moved to register $R1$ is never used in any succeeding move instructions, we may even remove the former instruction $alu1.result \rightarrow R1$. This requires a liveness analysis of all values in the generated TTA move schedule and is currently not part of our TTA compiler.

To see a complete example of the use of software bypassing, refer to a simple MiniC [11] program listed in 4.4 that evaluates two expressions. MiniC is a concise language with C-like syntax developed at the Chair of Embedded systems at the University of Kaiserslautern and it is used for both educational and research purposes. The corresponding Abacus assembly program is listed in 4.5 with the basic blocks marked. The basic block `bb1` is represented as a data flow graph (data dependency graph) in Figure 4.12. The nodes are Abacus instructions with the same line number as the number in the node.

Listing 4.4: A MiniC program

```
1 bool bl;
2 nat a,b,c,d,e,f,g;
3
4 if (bl)
5     g = (e-5)+(f+6);
6 else
7     d = ((a+2)+(b*3))-(c-4);
```

Algorithm 3: Scheduling of move instructions set T associated with an Abacus instruction v given a clock cycle clk

```
1  n ← number of buses
2  index ← number of moves scheduled in clk
3  T_scheduled ← ∅
4  while T_scheduled ≠ T do
5      while index >= n do
6          clk ← clk + 1
7          index ← number of moves scheduled in clk
8      end
9      if ∃t | t ∈ T, t ⊂ operandInMove then
10         AddToSchedule(clk, t)
11         T_scheduled ← t ∪ T_scheduled
12     else if ∃t | t ∈ T, t ⊂ triggerInMove, ∄ t_op ∈ T| t_op ⊂ operandInMove, t_op ∉ T_scheduled
        then
13         AddToSchedule(clk, t)
14         T_scheduled ← t ∪ T_scheduled
15     else if ∃t | t ∈ T, t ⊂ resultMove, ∄ t_tr ∈ T| t_tr ⊂ triggerInMove, t_tr ∉ T_scheduled then
16         clk = clk + δ
17         index ← number of moves scheduled in clk
18         while index >= n do
19             clk ← clk + 1
20             index ← number of moves scheduled in clk
21         end
22         AddToSchedule(clk, t)
23         T_scheduled ← t ∪ T_scheduled
24     else if ∃t | t ∈ T, t ⊂ memoryStoreDataMove, ∄ t_res ∈ T, t_res ⊂ resultMove, t_res ∉
        T_scheduled then
25         AddToSchedule(clk, t)
26         T_scheduled ← t ∪ T_scheduled
27     else if
        ∃t | t ∈ T, t ⊂ memoryLoadResMove, ∄ t_res ∈ T, t_res ⊂ resultMove, t_res ∉ T_scheduled
        then
28         clk = clk + δ'
29         index ← number of moves scheduled in clk
30         while index >= n do
31             clk ← clk + 1
32             index ← number of moves scheduled in clk
33         end
34         AddToSchedule(clk, t)
35         T_scheduled ← t ∪ T_scheduled
36     else
37         Do nothing
38     end
39 end
```

Listing 4.5: The Abacus assembly program

```
0  bb0:    mov     $0,0
1          ldi     $1,$0,7
2          bnz     $1,bb2
3  bb1:    ldi     $1,$0,4
4          ldi     $2,$0,5
5          subiu   $1,$1,5
6          addiu   $2,$2,6
7          addu    $1,$1,$2
8          sti     $1,$0,6
9          j       end
10 bb2:    ldi     $1,$0,0
11         ldi     $2,$0,0
12         addiu   $1,$1,2
13         muliu   $2,$2,3
14         ldi     $3,$0,2
15         subiu   $3,$3,4
16         addu    $1,$1,$2
17         subu    $1,$1,$3
18         sti     $1,$0,3
19 bb3:    nop
```

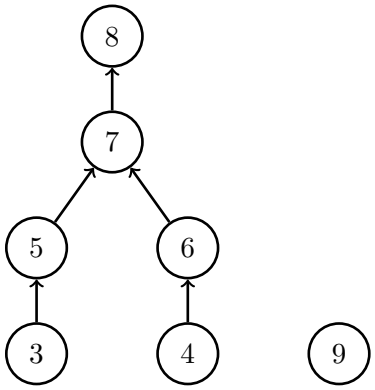


Figure 4.12: Data flow graph of basic block $bb1$ that computes the expression $g = (e - 5) + (f + 6)$ in the MiniC program in Listing 4.4

For simplicity, consider a TTA architecture with unlimited number of function units and registers, but only 4 transport buses. All function units are universal and have three inputs (unlike two input function units considered in other sections in this report). Apart from inputs $op1$ and $op2$, the trigger input tr stores an execution $opcode$, that decides the computation in the function unit. Translation of relevant Abacus instructions to TTA move instructions is then shown in Table 4.6. Note that only for memory write operation, we have a fourth input $op3$ that stores the data to be written to the main memory. The arithmetic/logic operations are assumed to take 2 clock cycles to complete, while the memory load/store operations are assumed to have unit latencies.

Now the Figure 4.13 shows the data dependency among the move instructions obtained by translating Abacus instructions in basic block `bb1`, following the mapping in Table 4.6. Running the scheduling algorithms in 1, 2 and 3 on the move instructions in basic block `bb1` gives the schedule shown in Table 4.7. Note that the total execution time of the basic block is 11 clock cycles. Software bypassing is now

Abacus Instr	Corresponding TTA Move Instructions			
n	n_{op1}	n_{op2}	n_{tr}	n_{res}/n_{op3}
mov rd,c			*c→rd*;	
ldi rd,rl,c	*rl→uni.op1*;	*c→uni.op2*;	*ld→uni.f*;	*uni.res→rd*
sti rd,rl,c	*rl→uni.op1*;	*c→uni.op2*;	*st→uni.f*;	*rd→uni.res*
addiu rd,rl,c	*rl→uni.op1*;	*c→uni.op2*;	*addu→uni.f*;	*alu.uni→rd*
muliu rd,rl,c	*rl→uni.op1*;	*c→uni.op2*;	*mulu→uni.f*;	*uni.res→rd*
subiu rd,rl,c	*rl→uni.op1*;	*c→uni.op2*;	*subu→uni.f*;	*uni.res→rd*
addu rd,rl,rr	*rl→uni.op1*;	*rr→uni.op2*;	*addu→uni.f*;	*uni.res→rd*
subu rd,rl,rr	*rl→uni.op1*;	*rr→uni.op2*;	*subu→uni.f*;	*uni.res→rd*
j end		*end→uni.op2*;	*j→uni.f*;	
bnz rd,tgt	*rd→uni.op1*;	*tgt→uni.op2*;	*bnz→uni.f*;	

Table 4.6: Mapping Abacus Instructions to move Instructions for a TTA Machine with one universal function unit

applied on this schedule to generate the new improved schedule of packed move instructions in Table 4.8 that only takes 9 clock cycles to execute the basic block. The pair of move instructions on which software bypassing is applied are underlined in the new schedule.

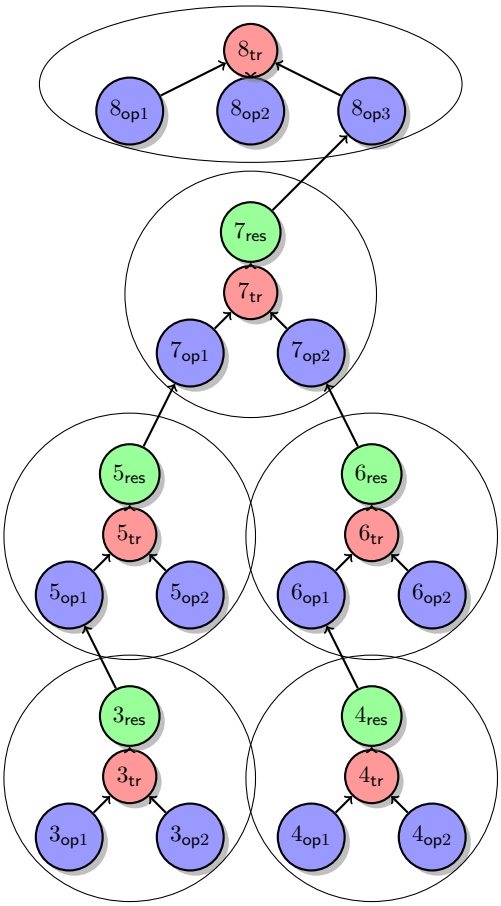


Figure 4.13: Dependency graph of TTA move instructions for basic block $bb1$

Clk	Scheduled TTA Move Instructions			
	bus 1	bus 2	bus 3	bus 4
0	3_{op1}	3_{op2}	3_{tr}	5_{op2}
1	3_{res}	4_{op1}	4_{op2}	4_{tr}
2	4_{res}	5_{op1}	5_{tr}	6_{op2}
3	6_{op1}	6_{tr}	8_{op1}	8_{op2}
4				
5	5_{res}			
6	6_{res}	7_{op1}		
7	7_{op2}	7_{tr}		
8				
9				
10	7_{res}			
11	8_{op3}	8_{tr}		

Table 4.7: Schedule of basic block $bb1$ without software bypassing

Clk	Scheduled TTA Move Instructions			
	bus 1	bus 2	bus 3	bus 4
0	3_{op1}	3_{op2}	3_{tr}	5_{op2}
1	3_{res}	4_{op1}	4_{op2}	4_{tr}
2	$\underline{4_{res}}$	5_{op1}	5_{tr}	$\underline{6_{op1}}$
3	6_{op2}	6_{tr}	8_{op1}	8_{op2}
4				
5	$\underline{5_{res}}$	$\underline{7_{op1}}$		
6	$\underline{6_{res}}$	$\underline{7_{op2}}$	7_{tr}	
7				
8				
9	$\underline{7_{res}}$	$\underline{8_{op3}}$	8_{tr}	

Table 4.8: Schedule of basic block $bb1$ with software bypassing

Chapter 5

Implementation and Experimental Results

This chapter explains the implementation details of our TTA compiler followed by experimental results obtained by compiling few benchmarks for predefined TTA hardware.

5.1 Implementation

The Java programming language Version 1.8.04 [6] is used to implement the Abacus to TTA Compiler. Java is selected as the programming language because it supports an object oriented programming (OOP) paradigm and is widely used. One of the advantages of OOP is intuitive and efficient mapping of real world objects. The Java programming language provides *classes* which are blueprints. An object is an instance of the *class* and inherits the properties of the class. To understand this concept in the TTA compiler context, let us consider the example of a TTA processor with two homogeneous ALUs, namely ALU_1 and ALU_2. To model this in Java, one can create a *class* ALU with all the properties of the ALU. As ALU_1 and ALU_2 are homogeneous, we may simply create instances of the ALU class to create objects ALU_1 and ALU_2.

The TTA Compiler is modeled taking advantage of these properties of Java. To describe the modelling of the TTA compiler, an overview of the same is shown in Figure 5.1. In this model, the functionality of the TTA compiler is split into manageable modules that perform some functionality. The functionality is represented as a set of nodes. The nodes of the model are shown as tabular blocks with two or three sub-blocks. The upper sub-block represents the module modelled. The middle one is the java file name that contains the code of the implementation of the block. The lower optional sub block represents either other modules that are triggered or new data that has been computed. The directed arrows represent either the flow of control in the model or one module being the sub-module of another module. Different notations are not used as during flow of control, control can move from one module to a sub-module.

The compiler model has three main modules. There is an additional module that represents the architectural definition file reader. This additional module parses the architecture definition files. The parsed data is used to generate the hardware architecture map shown in Figure 5.3. The three main modules are the User Interface *UI* module, the TTA hardware map generator *HMG* module and Compiler code map generator and scheduler *CCMGS* module.

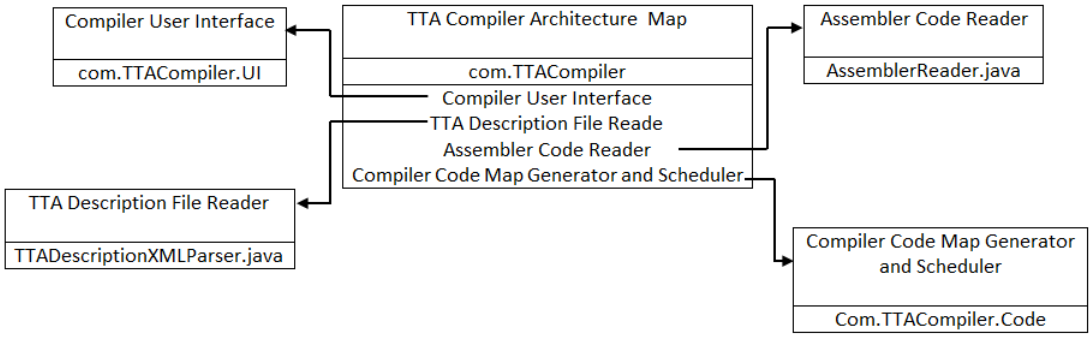


Figure 5.1: Overview of implementation model of compiler

The UI module is shown in Figure 5.2. The main frame is the module that represents the window generated for user access. The window allow the user to select the TTA description files .adf (architecture definition file) and *.fudf (function unit defintion file)[1].

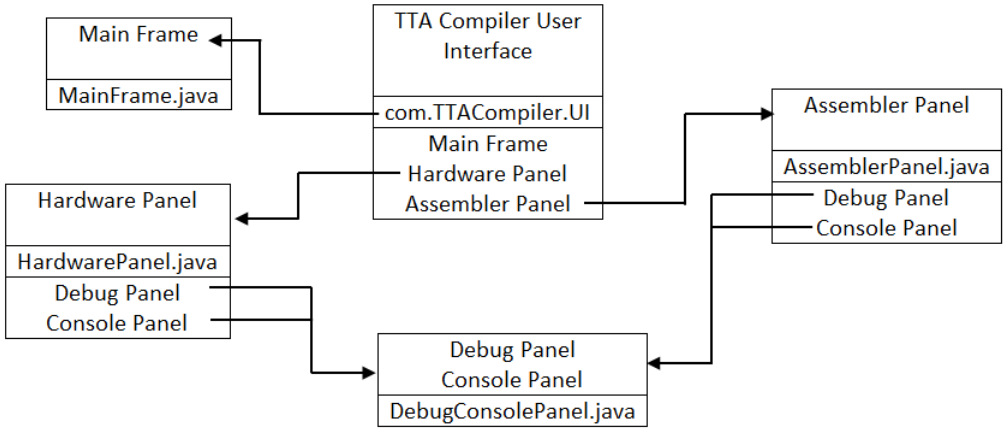


Figure 5.2: Overview of implementation of compiler user interface

The overview of the *HMG* module shown in the Figure 5.3. It maps the hardware of the TTA to a set of Java classes. The hardware map is a copy of the TTA processor architecture along with its datapath and function unit dependencies. In the Figure 5.3 to prevent overlapping of the directed arrows which describe structural dependencies and the Abacus operations supported a connector $\otimes$ is used.

A simplified overview of the *CCMGS* module that generates the packed TTA moves is shown in 5.4. This module reads the Abacus instruction and generates a code map. The code map links the various branch operations in the code to the respective labels. This module triggers the TTA translator module. In Figure 4.1 the TTA Translator is shown as a external module. This is because the Compiler is designed with the TTA Translator as a modifiable external module. This is to handle future changes in the TTA Instruction Set.

[1]The TTA hardware description has one .adf file describing the specific TTA processor for which the TTA Compiler must generate a schedule and multiple .fudf files describing the different function units that the TTA processor can have.

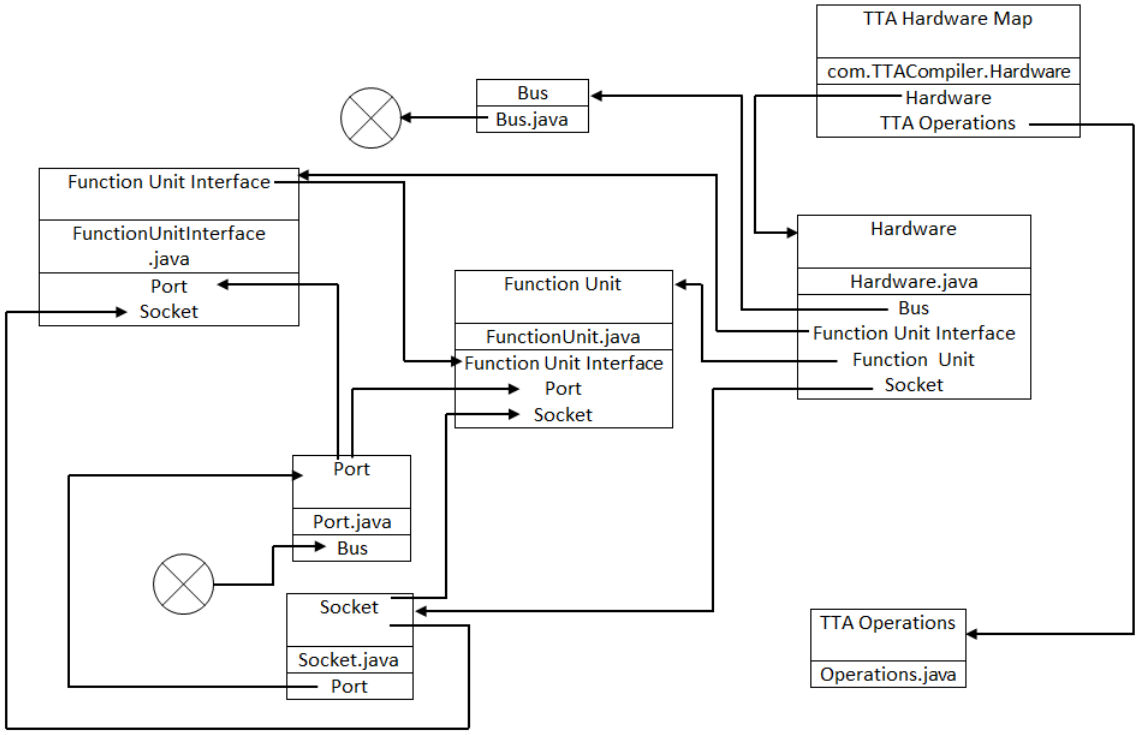


Figure 5.3: Overview of implementation model of TTA hardware map in the compiler

The *CCMGS* module also triggers the Basic Block with Control Flow and Dependency Graph Generator. This generates the basic blocks by analyzing the data dependencies of instructions of the input Abacus code. It also generates the Control Flow Graph *CFG*. The *CFG* maps the flow of control from one basic block to another. The instruction dependencies are computed for the three data dependency types RAW, WAR and WAW. This is used to generate a Data Dependency Graph *DDG*. The *DDG* maps the data dependencies of Abacus instructions and not the individual TTA move instructions. This is because compiler scheduler uses Operation based List Scheduling for scheduling TTA move instructions. In Operation based List Scheduling, the scheduler schedules the TTA instructions as operations they represent and not as independent individual TTA instructions. An operation in TTA is mapped to a set of one or more moves. In this case the operations are the Abacus instruction operations.

The basic blocks, *DFG* and *CFG* are inputs to the Basic Block Scheduler *BBS* along with the translated TTA move instructions. The *BBS* generates the scheduled TTA instructions. The instructions are packed as the *BBS* generates VLIW instructions. The scheduled TTA instructions are then given as a input to the Software Bypass sub module to perform software bypassing. The Software Bypass sub module is triggered by the *CCMGS* sub module after *BBS* has completed generating the scheduled TTA instructions.

5.2 Experimental Results

The TTA Compiler designed was tested with three benchmarks. They are:

1. Vector Multiply - multiplication of two Vectors
2. Square Root of a natural number

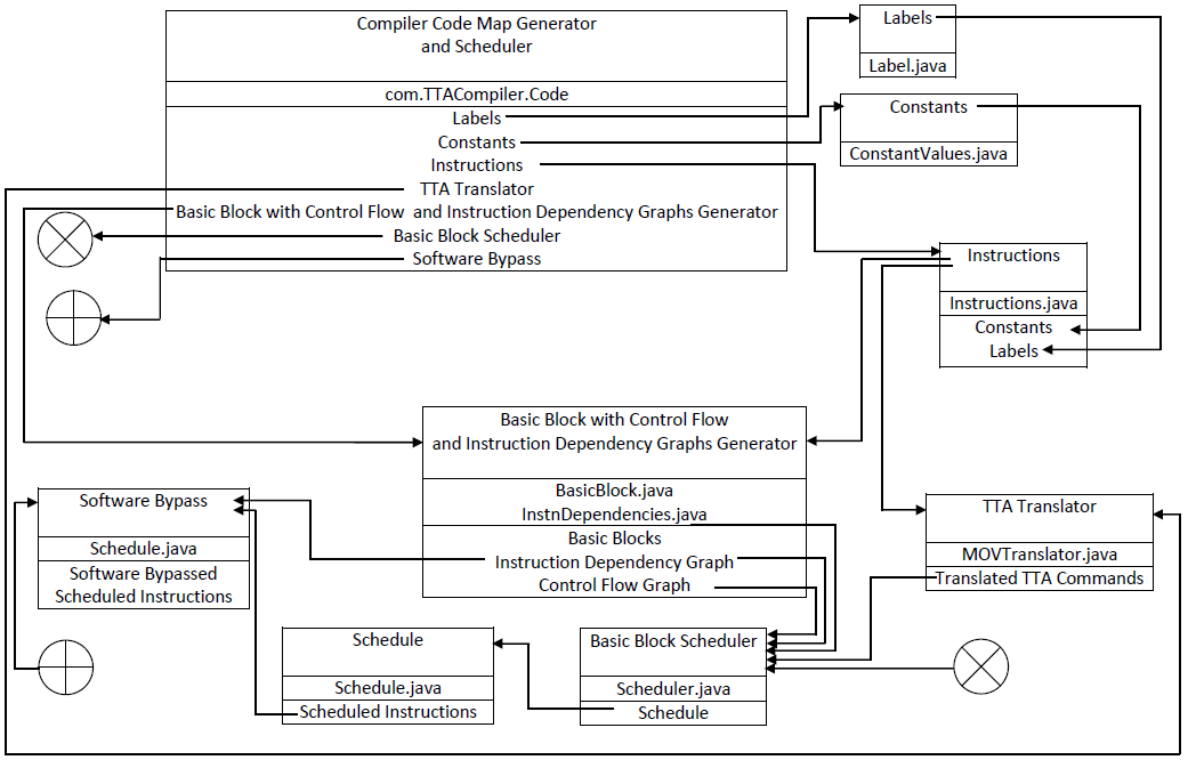


Figure 5.4: Overview of implementation model of Abacus to TTA move instruction scheduler in the compiler

3. Bubble Sort of an array of natural numbers

The Abacus assembly program, translated sequence of TTA move instructions, scheduled TTA parallel instruction bundles and the software bypassed TTA parallel instruction bundles for the first two benchmarks, vector multiply and square root are listed in Appendix A as examples. The listings have the name of the benchmark appended with .asm for Abacus source, _movSequence.asm for translated sequence of moves, _movPacked for scheduled parallel TTA code and _movPackedSb for software bypassed scheduled parallel TTA code. For the square root benchmark, the MiniC [11] source code used to generate the Abacus program is also listed.

		number of TTA instructions	
	benchmark	before scheduling	after scheduling
1	Vector Multiplication	55	24
2	Square Root	58	21
3	Bubble Sort	100	44

Table 5.1: Number of TTA instructions before and after scheduling

The number of move instructions before basic block scheduling and the number of packed move instruction bundles after basic block scheduling by the TTA Compiler is given in Table 5.1. Each move instruction in the sequence of moves before basic block scheduling as well as each instruction bundle after basic block scheduling are executed by the TTA hardware in one step or one clock cycle. From Table 5.1, clearly basic block scheduling results in lower execution times. The number of instruction bundles is

less than half the number of individual move instructions. This implies that the performance is more than doubled by efficient packing of individual moves. The number of instruction bundles remain the same after software bypassing as the TTA hardware considered for these experiments consists of only one alu unit. The alu considered has a delay δ of 3 clock cycles.

Chapter 6

Conclusion and Future Work

Transport triggered architectures (TTA) is a statically scheduled exposed datapath architecture. 'Exposed datapath' because TTA exposes its data path to the compiler so that the compiler is not only responsible to schedule operations on functional units, but also takes care of moving values between function units. 'Statically scheduled' because the compiler packs independent instruction in parallel bundles, where all instructions in one bundle is executed by the hardware in one clock cycle. In TTA, function units have registers at their input and output ports. Function units, register files and main memory are connected to one another via an interconnection network, which is usually a set of buses. TTAs are programmed by move instructions whose semantic is to move a value from the output port of a function unit / register to the input port of the same or different function unit / register. Static scheduling at data transport level makes TTA a time-predictable architecture, where worst case execution times of applications can be accurately predicted. This is an important feature of hardware architectures in real-time embedded systems. Furthermore, usage of move instructions enables TTA to include / plug in any arbitrary function unit that implements any arbitrary functionality with any number of inputs and outputs, without having to change the instruction set. This makes TTA an application specific architecture, where frequently occuring computations in applications can be implemented directly as a function unit in TTA, to improve the performance. In thesis work, a compiler is developed that generates parallel TTA move code for a given TTA machine from a sequence of reduced instruction set (RISC) instructions. Specifically assembly programs using RISC instructions from Abacus instruction set is used. Abacus is a family of processor architectures that implements MIPS-like RISC instruction set. Different implementations (single cycle, pipelined, VLIW, superscalar) of Abacus instruction set are possible.

In the designed TTA compiler, first the Abacus instructions are simply translated to a sequence of move instructions for a generic TTA architecture containing only one arithmetic logic unit (alu) and one load store unit (lsu) connected to main memory. Basic blocks are identified and dependency graphs of both Abacus instructions and translated TTA move instructions in basic blocks are generated. Then the move instructions in basic blocks are scheduled using the famous list scheduling technique, where simple first-fit resource assignment is used to map the computations to available function units and also to map the data transports to available buses. In a final step, the register/software bypassing capability of TTA is utilized by the compiler, resulting in a reduced register pressure and improved instruction level parallelism. Register/ software bypassing refers to the ability of TTA compilers to move values from the output port of a function unit to the input port of the same or different function unit, without having to store the value in

a register. We show experimentally the performance improvement both by packing of move instructions in basic blocks and by utilizing software bypassing.

In the future work, it is important to extend the compiler by implementing global (extended basic block) scheduling [9] and cyclic scheduling (software pipelining) [9]. Both these techniques facilitates the compiler to find enough independent instructions across the basic blocks and in loops, to maximize the utilization of available hardware resources (function units and buses) improving the performance. In the current TTA compiler version, since we generate parallel TTA move code from RISC instructions where the variables are already assigned to registers, register allocation is not performed. In future it is desirable to generate parallel TTA code from the any intermediate representation so that then the compiler can explicitly allocate variables to registers. This will make way for the well known phase ordering problem where we must explore the advantages and disadvantages of different orderings of the register allocation phase and instruction scheduling phase in our TTA compiler.

Appendix A

Examples

To test the compiler, we use a TTA processor architecture, that consists of one arithmetic logic unit (alu) and one load-store unit (lsu) connected to the main memory. Here the delay of the alu unit is set to 3 clock cycles and the delay of the lsu unit is set to 1 cycle (assuming some on-chip scratchpad memory that can be read and written to in one clock cycle) for simplicity.

First test case is a simple program that performs multiplication of two vectors. The Abacus program that computes the inner product of two vectors $a = (1, 3, ..., 2N - 1)$ and $b = (2, 4, ..., 2N)$ for some parameter N is shown in Listing A.1. For testing we assume that $N = 20$. This Abacus assembly program file is the input selected in the assembler panel of the TTA Compiler to generate the scheduled TTA instructions. The translated sequence of TTA move instructions are shown in Listing A.2 and finally the packed TTA moves after basic block scheduling for the considered TTA processor architecture is shown in Listing A.3. The empty lines in this final schedule (lines without any move instructions) are clock cycles in which no move instructions could be scheduled.

Listing A.1: vectorMultiply.asm

```
// ---------------------------------------------------------
// The following Abacus program computes the innner product of the vectors
// a=(1,3,...,2N-1) and b=(2,4,...,2N) for some parameter N. These vectors are
// first generated and are stored in memory, where vector a will start at
// address 0 and vector b will start at address N. After the vectors have been
// generated, their inner product is computed and will be finally available in
// register $7. Assume N = 20.
// ---------------------------------------------------------

// ---------------------------------------------------------
// first, generate the argument vectors in memory
// ---------------------------------------------------------
      mov $0,0          // Reg[0] := 0
      mov $1,20          // Reg[1] := N
      mov $2,0          // Reg[2] := 0 (loop variable)
      mov $3,1          // Reg[3] := 1 (elements of a)
      mov $4,2          // Reg[4] := 2 (elements of b)
l1 :  st  $3,$0,$2      // Mem[Reg[2]+0] := Reg[3]
      st  $4,$1,$2      // Mem[Reg[2]+N] := Reg[4]
      addi  $2,$2,1     // Reg[2] := Reg[2] + 1
      addi  $3,$3,2     // Reg[3] := Reg[3] + 2
      addi  $4,$4,2     // Reg[4] := Reg[4] + 2
      slt   $5,$2,$1    // Reg[5] := Reg[2] < Reg[1]
      bnz $5,l1         // if Reg[5]!=0 goto l1
      sync
// ---------------------------------------------------------
// compute the inner product of the generated vectors
// ---------------------------------------------------------
      mov $2,0          // Reg[2] := 0 (loop variable)
      mov $7,0          // Reg[7] := 0 (preliminary inner product)
l2 :  ld  $3,$0,$2      // Reg[3] := Mem[Reg[2]+0] (enumerates elements of a)
      ld  $4,$1,$2      // Reg[4] := Mem[Reg[2]+N] (enumerates elements of b)
      mulu $6,$3,$4     // Reg[6] := Reg[3] * Reg[4]
      addu $7,$7,$6     // Reg[7] := Reg[7] + Reg[6]
      addi  $2,$2,1     // Reg[2] := Reg[2] + 1
      slt   $5,$2,$1    // Reg[5] := Reg[2] < Reg[1]
      bnz $5,l2         // if Reg[5]!=0 goto l2
      sync
// ---------------------------------------------------------
```

Listing A.2: vectorMultiply_movSequence.asm

```
// 0 : mov $0,0
0 :  LOAD 0, R0;

// 1 : mov $1,20
1 :  LOAD 20, R1;

// 2 : mov $2,0
2 :  LOAD 0, R2;

// 3 : mov $3,1
3 :  LOAD 1, R3;

// 4 : mov $4,2
4 :  LOAD 2, R4;

// 5 : l1: st $3,$0,$2
5 :  l1  :  MOV R0, alu.add.op;
6 :  l1  :  MOV R2, alu.add.tr;
7 :  l1  :  MOV alu.result1,  ram.write;
8 :  l1  :  MOV R3, ram.result;

// 6 : st $4,$1,$2
9 :  MOV R1, alu.add.op;
10 :  MOV R2, alu.add.tr;
11 :  MOV alu.result1,  ram.write;
12 :  MOV R4, ram.result;

// 7 : addi $2,$2,1
13 :  LOAD 1, alu.addi.op;
14 :  MOV R2, alu.addi.tr;
15 :  MOV alu.result1,  R2;

// 8 : addi $3,$3,2
16 :  LOAD 2, alu.addi.op;
17 :  MOV R3, alu.addi.tr;
18 :  MOV alu.result1,  R3;

// 9 : addi $4,$4,2
19 :  LOAD 2, alu.addi.op;
20 :  MOV R4, alu.addi.tr;
21 :  MOV alu.result1,  R4;

// 10 : slt $5,$2,$1
22 :  MOV R2, alu.slt.op;
23 :  MOV R1, alu.slt.tr;
24 :  MOV alu.result1,  R5;

// 11 : bnz $5,l1
25 :  LOAD 0, alu.seq.op;
26 :  MOV R5, alu.seq.tr;
```

```
27 :  BRANCH !alu.result1, l1 ;

// 12 : sync
28 :  NOP;

// 13 : mov $2,0
29 :  LOAD 0, R2;

// 14 : mov $7,0
30 :  LOAD 0, R7;

// 15 : l2: ld $3,$0,$2
31 :  l2  :  MOV R0, alu.add.op;
32 :  l2  :  MOV R2, alu.add.tr ;
33 :  l2  :  MOV alu.result1 ,  ram.read ;
34 :  l2  :  MOV ram.result,  R3;

// 16 : ld $4,$1,$2
35 :  MOV R1, alu.add.op;
36 :  MOV R2, alu.add.tr ;
37 :  MOV alu.result1 ,  ram.read ;
38 :  MOV ram.result,  R4;

// 17 : mulu $6,$3,$4
39 :  MOV R3, alu.mulu.op;
40 :  MOV R4, alu.mulu.tr;
41 :  MOV alu.result1 ,  R6;

// 18 : addu $7,$7,$6
42 :  MOV R7, alu.addu.op;
43 :  MOV R6, alu.addu.tr ;
44 :  MOV alu.result1 ,  R7;

// 19 : addi $2,$2,1
45 :  LOAD 1, alu.addi . op;
46 :  MOV R2, alu.addi. tr ;
47 :  MOV alu.result1 ,  R2;

// 20 : slt $5,$2,$1
48 :  MOV R2, alu. slt . op;
49 :  MOV R1, alu. slt . tr ;
50 :  MOV alu.result1 ,  R5;

// 21 : bnz $5,l2
51 :  LOAD 0, alu.seq . op;
52 :  MOV R5, alu.seq. tr ;
53 :  BRANCH !alu.result1, l2 ;

// 22 : sync
54 :  NOP;
```

Listing A.3: vectorMultiply_movPacked.asm

```
BB0

0 : LOAD 0, R0; LOAD 20, R1; LOAD 0, R2; LOAD 1, R3;
1 : LOAD 2, R4;

--------------------------------------------

BB1

2 : MOV R0, alu.add.op; MOV R2, alu.add.tr;
3 :
4 :
5 : MOV alu.result1, ram.write;
6 : MOV R3, ram.result; MOV R1, alu.add.op; MOV R2, alu.add.tr;
7 :
8 :
9 : MOV alu.result1, ram.write;
10 : MOV R4, ram.result; LOAD 1, alu.addi.op; MOV R2, alu.addi.tr;
11 :
12 :
13 : MOV alu.result1, R2; LOAD 2, alu.addi.op; MOV R3, alu.addi.tr;
14 :
15 :
16 : MOV alu.result1, R3; LOAD 2, alu.addi.op; MOV R4, alu.addi.tr;
17 :
18 :
19 : MOV alu.result1, R4; MOV R2, alu.slt.op; MOV R1, alu.slt.tr;
20 :
21 :
22 : MOV alu.result1, R5; LOAD 0, alu.seq.op; MOV R5, alu.seq.tr;
23 :
24 :
25 : BRANCH !alu.result1, l1;

--------------------------------------------

BB2

26 : NOP; LOAD 0, R2; LOAD 0, R7;

--------------------------------------------

BB3

27 : MOV R0, alu.add.op; MOV R2, alu.add.tr;
28 :
29 :
30 : MOV alu.result1, ram.read;
31 : MOV ram.result, R3; MOV R1, alu.add.op; MOV R2, alu.add.tr;
32 :
33 :
```

```
34 : MOV alu.result1, ram.read;
35 : MOV ram.result, R4; MOV R3, alu.mulu.op; MOV R4, alu.mulu.tr;
36 :
37 :
38 : MOV alu.result1, R6; LOAD 1, alu.addi.op; MOV R2, alu.addi.tr;
39 :
40 :
41 : MOV alu.result1, R2; MOV R7, alu.addu.op; MOV R6, alu.addu.tr;
42 :
43 :
44 : MOV alu.result1, R7; MOV R2, alu.slt.op; MOV R1, alu.slt.tr;
45 :
46 :
47 : MOV alu.result1, R5; LOAD 0, alu.seq.op; MOV R5, alu.seq.tr;
48 :
49 :
50 : BRANCH !alu.result1, l2;

----------------------------------------

BB4

51 : NOP;

----------------------------------------
```

Listing A.4: vectorMultiply_movPackedSb.asm

```
BB0

0 : LOAD 0, R0; LOAD 20, R1; LOAD 0, R2; LOAD 1, R3;
1 : LOAD 2, R4;

--------------------------------------------

BB1

2 : l1 : MOV R0, alu.add.op; MOV R2, alu.add.tr;
3 :
4 :
5 : MOV alu.result1, ram.write;
6 : MOV R3, ram.result; MOV R1, alu.add.op; MOV R2, alu.add.tr;
7 :
8 :
9 : MOV alu.result1, ram.write;
10 : MOV R4, ram.result; LOAD 1, alu.addi.op; MOV R2, alu.addi.tr;
11 :
12 :
13 : MOV alu.result1, R2; LOAD 2, alu.addi.op; MOV R3, alu.addi.tr;
14 :
15 :
16 : MOV alu.result1, R3; LOAD 2, alu.addi.op; MOV R4, alu.addi.tr;
17 :
18 :
19 : MOV alu.result1, R4; MOV R2, alu.slt.op; MOV R1, alu.slt.tr;
20 :
21 :
22 : MOV alu.result1, R5; LOAD 0, alu.seq.op; MOV alu.result1, alu.seq.tr;
23 :
24 :
25 : BRANCH !alu.result1, l1;

--------------------------------------------

BB2

26 : NOP; LOAD 0, R2; LOAD 0, R7;

--------------------------------------------

BB3

27 : l2 : MOV R0, alu.add.op; MOV R2, alu.add.tr;
28 :
29 :
30 : MOV alu.result1, ram.read;
31 : MOV ram.result, R3; MOV R1, alu.add.op; MOV R2, alu.add.tr;
32 :
33 :
```

```
34 : MOV alu.result1, ram.read;
35 : MOV ram.result, R4; MOV R3, alu.mulu.op; MOV ram.result, alu.mulu.tr;
36 :
37 :
38 : MOV alu.result1, R6; LOAD 1, alu.addi.op; MOV R2, alu.addi.tr;
39 :
40 :
41 : MOV alu.result1, R2; MOV R7, alu.addu.op; MOV R6, alu.addu.tr;
42 :
43 :
44 : MOV alu.result1, R7; MOV R2, alu.slt.op; MOV R1, alu.slt.tr;
45 :
46 :
47 : MOV alu.result1, R5; LOAD 0, alu.seq.op; MOV alu.result1, alu.seq.tr;
48 :
49 :
50 : BRANCH !alu.result1, l2;

_____________________________________________

BB4

51 : NOP;

_____________________________________________
```

Next test case to verify the TTA Compiler functionality is shown in Listing A.6. This Abacus code, generated from the MiniC code in Listing A.5 implements Herons iteration to compute the square root of a natural number. MiniC [11] is a simple language with C-like syntax developed at the Chair of Embedded systems at the University of Kaiserslautern for educational and research purposes. Again the sequence of moves translated from the Abacus code is shown in Listing A.7 and final packed move schedule generated is Listed in A.8.

Listing A.5: squareroot.c

```
// ---------------------------------------------------------
// The function below computes the integer approximation to the square root
// of a given natural number a. It is known as Heron's algorithm, but is also
// derived by the Newton-Raphson iteration of f(x) = x^2-a to compute roots.
// ---------------------------------------------------------

function heron(nat a) : nat {
    nat xold,xnew;
    xnew = a;
    do {
        xold = xnew;
        xnew = (xold + a/xold)/2;
    } while(xnew < xold);
    return xold;
}

thread Heron {
    nat z;
    z = heron(121); // compute the square root of 121
}
```

Listing A.6: squareroot.asm

```
      mov $2,0          // xnew_FC1 := 121
l1 :  mov $0,6          //
      sft  $2,$2,\$0    //
      mov $7,7          // mov $0,7
      or  $2,$2,$0      //
      mov $0,4          //
      sft  $2,$2,$0     //
      addiu  $2,$2,9    //
      addiu  $1,$2,0    // xold_FC1 := xnew_FC1
      mov $2,0          // _t2 := 121
l2 :      mov $0,6      //
      sft  $2,$2,$0     //
      mov $0,7          //
      or  $2,$2,$0      //
      mov $0,4          //
      sft  $2,$2,$0     //
      addiu  $2,$2,9    //
      divu  $2,$2,$1    // _t1 := _t2 / xold_FC1
      addu $2,$1,$2     // _t3 := xold_FC1 + _t1
      diviu  $2,$2,2    // xnew_FC1 := _t3 / 2
      sltu  $3,$2,$1    // _t4 := xnew_FC1 < xold_FC1
      bnz $3,l2         // if _t4 goto 1
      addiu  $1,$1,0    // _t0 := xold_FC1
      j  l2             // goto 10
      addiu  $1,$1,0    // z := _t0
      sync              // sync
```

Listing A.7: squareroot_movSequence.asm

```
// 0 : mov $2,0
0  :  LOAD 0, R2;

// 1 : l1: mov $0,6
1  :  l1  :  LOAD 6, R0;

// 2 : sft $2,$2,$0
2  :  MOV R2, alu.sft.op;
3  :  MOV R0, alu.sft. tr ;
4  :  MOV alu.result1 ,  R2;

// 3 : mov $7,7
5  :  LOAD 7, R7;

// 4 : or $2,$2,$0
6  :  MOV R2, alu.or.op;
7  :  MOV R0, alu.or. tr ;
8  :  MOV alu.result1 ,  R2;

// 5 : mov $0,4
9  :  LOAD 4, R0;

// 6 : sft $2,$2,$0
10  :  MOV R2, alu.sft.op;
11  :  MOV R0, alu.sft. tr ;
12  :  MOV alu.result1 ,  R2;

// 7 : addiu $2,$2,9
13  :  LOAD 9, alu.addiu.op;
14  :  MOV R2, alu.addiu. tr ;
15  :  MOV alu.result1 ,  R2;

// 8 : addiu $1,$2,0
16  :  LOAD 0, alu.addiu.op;
17  :  MOV R2, alu.addiu. tr ;
18  :  MOV alu.result1 ,  R1;

// 9 : mov $2,0
19  :  LOAD 0, R2;

// 10 : l2: mov $0,6
20  :  l2  :  LOAD 6, R0;

// 11 : sft $2,$2,$0
21  :  MOV R2, alu.sft.op;
22  :  MOV R0, alu.sft. tr ;
23  :  MOV alu.result1 ,  R2;

// 12 : mov $0,7
24  :  LOAD 7, R0;
```

```
// 13 : or $2,$2,$0
25 : MOV R2, alu.or.op;
26 : MOV R0, alu.or.tr ;
27 : MOV alu.result1 , R2;

// 14 : mov $0,4
28 : LOAD 4, R0;

// 15 : sft $2,$2,$0
29 : MOV R2, alu.sft.op;
30 : MOV R0, alu.sft.tr ;
31 : MOV alu.result1 , R2;

// 16 : addiu $2,$2,9
32 : LOAD 9, alu.addiu.op;
33 : MOV R2, alu.addiu.tr ;
34 : MOV alu.result1 , R2;

// 17 : divu $2,$2,$1
35 : MOV R2, alu.divu.op;
36 : MOV R1, alu.divu.tr ;
37 : MOV alu.result1 , R2;

// 18 : addu $2,$1,$2
38 : MOV R1, alu.addu.op;
39 : MOV R2, alu.addu.tr ;
40 : MOV alu.result1 , R2;

// 19 : diviu $2,$2,2
41 : MOV R2, alu.diviu.op;
42 : LOAD 2, alu.diviu.tr ;
43 : MOV alu.result1 , R2;

// 20 : sltu $3,$2,$1
44 : MOV R2, alu.sltu.op;
45 : MOV R1, alu.sltu.tr ;
46 : MOV alu.result1 , R3;

// 21 : bnz $3,l2
47 : LOAD 0, alu.seq.op;
48 : MOV R3, alu.seq.tr ;
49 : BRANCH !alu.result1, l2 ;

// 22 : addiu $1,$1,0
50 : LOAD 0, alu.addiu.op;
51 : MOV R1, alu.addiu.tr ;
52 : MOV alu.result1 , R1;

// 23 : j l2
53 : BRANCH true, l2;
```

```
//24 : addiu $1,$1,0
54 :  LOAD 0, alu.addiu.op;
55 :  MOV R1, alu.addiu.tr ;
56 :  MOV alu.result1 ,  R1;

//25 : sync
57 :  NOP;
```

Listing A.8: squareroot_movPacked.asm

```
BB0

0 : LOAD 0, R2;

--------------------------------------------

BB1

1 : l1 : LOAD 6, R0; LOAD 7, R7; MOV R2, alu.sft.op; MOV R0, alu.sft.tr;
2 :
3 :
4 : MOV alu.result1, R2; MOV R2, alu.or.op; MOV R0, alu.or.tr;
5 :
6 :
7 : MOV alu.result1, R2; LOAD 4, R0; MOV R2, alu.sft.op; MOV R0, alu.sft.tr;
8 :
9 :
10 : MOV alu.result1, R2; LOAD 9, alu.addiu.op; MOV R2, alu.addiu.tr;
11 :
12 :
13 : MOV alu.result1, R2; LOAD 0, alu.addiu.op; MOV R2, alu.addiu.tr;
14 :
15 :
16 : MOV alu.result1, R1; LOAD 0, R2;

--------------------------------------------

BB2

17 : l2 : LOAD 6, R0; MOV R2, alu.sft.op; MOV R0, alu.sft.tr;
18 :
19 :
20 : MOV alu.result1, R2; LOAD 7, R0; MOV R2, alu.or.op; MOV R0, alu.or.tr;
21 :
22 :
23 : MOV alu.result1, R2; LOAD 4, R0; MOV R2, alu.sft.op; MOV R0, alu.sft.tr;
24 :
25 :
26 : MOV alu.result1, R2; LOAD 9, alu.addiu.op; MOV R2, alu.addiu.tr;
27 :
28 :
29 : MOV alu.result1, R2; MOV R2, alu.divu.op; MOV R1, alu.divu.tr;
30 :
31 :
32 : MOV alu.result1, R2; MOV R1, alu.addu.op; MOV R2, alu.addu.tr;
33 :
34 :
35 : MOV alu.result1, R2; MOV R2, alu.diviu.op; LOAD 2, alu.diviu.tr;
36 :
37 :
38 : MOV alu.result1, R2; MOV R2, alu.sltu.op; MOV R1, alu.sltu.tr;
```

```
39 :
40 :
41 : MOV alu.result1 , R3; LOAD 0, alu.seq.op; MOV R3, alu.seq.tr ;
42 :
43 :
44 : BRANCH !alu.result1, l2 ;

----------------------------------------

BB3

45 : LOAD 0, alu.addiu.op; MOV R1, alu.addiu.tr ;
46 :
47 :
48 : MOV alu.result1 , R1; BRANCH true, l2;

----------------------------------------

BB4

49 : LOAD 0, alu.addiu.op; MOV R1, alu.addiu.tr ;
50 :
51 :
52 : MOV alu.result1 , R1; NOP;

----------------------------------------
```

Listing A.9: squareroot_movPackedSb.asm

```
BB0

0 : LOAD 0, R2;

--------------------------------------------

BB1

1 : l1 : LOAD 6, R0; LOAD 7, R7; MOV R2, alu.sft.op; MOV R0, alu.sft.tr;
2 :
3 :
4 : MOV alu.result1, R2; MOV alu.result1, alu.or.op; MOV R0, alu.or.tr;
5 :
6 :
7 : MOV alu.result1, R2; LOAD 4, R0; MOV alu.result1, alu.sft.op; MOV R0, alu.sft.tr;
8 :
9 :
10 : MOV alu.result1, R2; LOAD 9, alu.addiu.op; MOV alu.result1, alu.addiu.tr;
11 :
12 :
13 : MOV alu.result1, R2; LOAD 0, alu.addiu.op; MOV alu.result1, alu.addiu.tr;
14 :
15 :
16 : MOV alu.result1, R1; LOAD 0, R2;

--------------------------------------------

BB2

17 : l2 : LOAD 6, R0; MOV R2, alu.sft.op; MOV R0, alu.sft.tr;
18 :
19 :
20 : MOV alu.result1, R2; LOAD 7, R0; MOV alu.result1, alu.or.op; MOV R0, alu.or.tr;
21 :
22 :
23 : MOV alu.result1, R2; LOAD 4, R0; MOV alu.result1, alu.sft.op; MOV R0, alu.sft.tr;
24 :
25 :
26 : MOV alu.result1, R2; LOAD 9, alu.addiu.op; MOV alu.result1, alu.addiu.tr;
27 :
28 :
29 : MOV alu.result1, R2; MOV alu.result1, alu.divu.op; MOV R1, alu.divu.tr;
30 :
31 :
32 : MOV alu.result1, R2; MOV R1, alu.addu.op; MOV alu.result1, alu.addu.tr;
33 :
34 :
35 : MOV alu.result1, R2; MOV alu.result1, alu.diviu.op; LOAD 2, alu.diviu.tr;
36 :
37 :
38 : MOV alu.result1, R2; MOV alu.result1, alu.sltu.op; MOV R1, alu.sltu.tr;
```

```
39 :
40 :
41 : MOV alu.result1, R3; LOAD 0, alu.seq.op; MOV alu.result1, alu.seq.tr;
42 :
43 :
44 : BRANCH !alu.result1, l2;

----------------------------------------

BB3

45 : LOAD 0, alu.addiu.op; MOV R1, alu.addiu.tr;
46 :
47 :
48 : MOV alu.result1, R1; BRANCH true, l2;

----------------------------------------

BB4

49 : LOAD 0, alu.addiu.op; MOV R1, alu.addiu.tr;
50 :
51 :
52 : MOV alu.result1, R1; NOP;

----------------------------------------
```

Bibliography

[1] H. Corporaal. TTAs: Missing the ILP complexity wall. *Journal of Systems Architecture*, 45(12-13):949–973, June 1999.

[2] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 1996. PhD.

[3] J. Janssen. *Compiler Strategies for Transport Triggered Architectures*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2001. PhD.

[4] A. Bhagyanath and K. Schneider. TTA as predictable architecture for real-time applications. In *International Conference on Science, Engineering, Research and Management (ICSEMR)*, Chennai, India, 2014. IEEE Computer Society.

[5] N. Bhardwaj, M. Senftleben, and K. Schneider. Abacus – a processor family for education. In M. Törngren and M.E. Grimheden, editors, *Workshop on Embedded and Cyber-Physical Systems Education (WESE'14)*, pages 2:1–2:8, New Delhi, India, 2015. ACM.

[6] Java programming language. `https://www.oracle.com/java/index.html`.

[7] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill. MIPS: A microprocessor architecture. In *Microarchitecture (MICRO)*, pages 17–22, Palo Alto, California, USA, 1982. IEEE Computer Society.

[8] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[9] J.A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[10] Extensible markup language. `https://www.w3.org/XML/`.

[11] Minic language. `http://es.informatik.uni-kl.de/tools/teaching/MiniC.html`.