Bachelor Thesis

Code Generation for Transport Triggered Architecture

Mario Reder August 31, 2014

Department of Computer Science,
University of Kaiserslautern,
D 67653 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
M. Sc. Anoop Bhagyanath

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema “Code Generation for Transport Triggered Architecture” selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 31. August 2014

_____
Vorname Name

Abstract

This thesis describes an implementation of a compiler for Transport Triggered Architectures (TTAs). TTAs consist of Function Units (FU), Register Files (RF) and memory connected using a set of 'MOV' (data transport) buses. They are programmed by a set of statically scheduled 'MOV' instructions that contain data transports between FUs and/or RFs. Operations are triggered on FUs as a side effect of data transports. Given a TTA hardware description file, the code generator generates parallel TTA code from an Abacus assembly program file (Abacus is a tiny MIPS-like processor architecture developed in the Chair of Embedded Systems for educational and research purposes).

Zusammenfassung

Diese Arbeit beschreibt die Implementierung eines Compilers für Transport Triggered Architectures (TTAs). TTAs bestehen aus Function Units (FU), Register Files (RF) und Speichereinheiten, welche durch 'MOV' (Datenübertragungs-) Busse verbunden sind. Sie werden durch statically scheduled 'MOV' Anweisungen gesteuert, welche Datenübertragungen zwischen FUs und/oder RFs beinhalten. Operationen werden als Nebenwirkung durch Datenübertragungen ausgelöst. Indem man dem Compiler eine Hardwarebeschreibungsdatei übergibt, kann dieser parallelen TTA Code aus einem Abacus Assemblerprogramm erzeugen (Ababus ist eine MIPS-ähnliche Prozessorarchitektur, welche in der AG Eingebettete Systeme für schulische und Forschungszwecke entwickelt wurde)

Contents

1 Introduction

1.1 Motivation

Embedded systems are application-specific computer systems, often dedicated for a particular task or a set of tasks. Today, embedded systems have become very common in our everyday life ranging from simple MP3 players to large complex avionics systems. Thus varying in the required computational power. Majority of these embedded systems are real-time systems, requiring the task(s) to be completed within specified time deadline(s). It is also important to keep the energy consumption of these systems as small as possible, since many of them shall be deployed in such environments where battery is the only usable source of energy. For example, wireless sensor systems.

Design methodology for embedded systems can range from classic hardware to classic software designs. In classic hardware design, the entire functionality is implemented in hardware, used either in the form of Application-Specific Integrated Chips (ASICs) or prototyped in Field Programmable Gate Arrays (FPGAs). In classic software design, the functionality is achieved by programming an off-the-shelf selected conventional microprocessor. Designers can also develop an application-specific processor and program the same in order to implement the embedded system. This approach falls between classic hardware and classic software design approaches. Conventional processors used in classic software design methodology can be broadly classified into dynamically scheduled and statically scheduled processors. In dynamically scheduled processors, software or program instructions are scheduled during program runtime by the hardware, while the compiler is responsible to schedule instructions for statically scheduled processors. Simple MIPS-like pipelined hardware architecture to complex out-of-order executing superscalar architectures fall in the dynamically scheduled category. VLIW (Very Long Instruction Word) processors are most common in statically scheduled category.

Transport Triggered Architectures (TTAs) are an extreme case of VLIW processors, where pure static scheduling is used. They consist of Function Units (FU), Register Files (RFs) and the main memory inter-connected using a set of data transport buses. TTAs has a different approach to execute operations compared to other conventional processors. It is a One Instruction Set Computer (OISC) programmed using a set of 'MOV' instructions that transport data between FUs and/or RFs. FUs execute as a side effect of data transport to their respective input storage units. Furthermore, a FU in TTA can be any hardware block with an arbitrary number of inputs and outputs. This generic nature of FUs allows creating a TTA processor specific to an application, thus allowing an application-specific embedded system design methodology to be followed. Furthermore, many researches are being carried out to minimize the energy requirements of TTAs.

1.2 About this document

This thesis is about code generation for TTAs. A TTA compiler was implemented, that, given a TTA hardware description file and an input assembly program written for Abacus hardware architecture, generates a set of scheduled data transports or TTA 'MOV' instructions for execution on the particular configured TTA hardware. '*Abacus*' is a tiny MIPS-like processor architecture developed in the Chair of Embedded Systems for educational and research purposes. Abacus hardware architecure is described in the next chapter, followed by TTA hardware architecture description. Then the architecture and design of code generator is discussed follwed by its implementation details. Final chapter summarizes the work done and mentions the probable future work.

2 Abacus

The term '*Abacus*' [BhSS14] is used to refer to a range of processor architectures, that has been developed indigenously in the Chair of Embedded Systems for both educational and research purposes. Thus it keeps the number of instructions as small as possible and hides as many implementation details as possible.

Following architectural versions of Abacus processor are available -

- Single cycle
- Pipelined (Both with Stalling and Data Forwarding)
- Out-of-order executing
- Very Long Instruction Word (VLIW)
- Vector Processor

2.1 Hardware Specification

An Abacus processor has separate data memory and instruction memory. Instruction memory stores the instructions corresponding to a program targeted to execute in Abacus and the data memory is used by the program to store and load values. There are eight general purpose registers R0 to R7. Register R0 has the special role to hold the constant zero. Writing to R0 is allowed, but will have no effect. Another special register is the overflow register *Ov* which stores the upper half of the result of ALU operations. In case of DIV operations, *Ov* stores the remainder value from the division performed. In the vector version of Abacus processor, there are also eight general purpose vector registers, which makes it possible for the instruction set to have both scalar and vector operations of the same type. Of course, there also exists a program counter PC to control the program flow.

Instruction execution is divided into five steps - Fetch (IF), Decode (ID), Execute (EX), Memory Access (MEM) and Writeback (WB). In the IF stage, an instruction fetch unit fetches the next instruction to execute, from the instruction memory at the address defined by the current value of the PC. The ID stage gets the operand values stored in registers from the register file and passes them on to the EX stage. The EX stage consists of the function units ALU, MUL and DIV capable of performing arithmetic/logic operations, multipication and division respectively. The next stage, memory access MEM, has a load-store unit (LSU), which is connected to the data memory. LSU loads/stores data from/to the data memory in case of load/store instructions. The last stage WB stores the output value from the EX and MEM stage back to the register file.

Figure 2.1 shows a single cycled implementation of an Abacus processor. In the *single cycle*

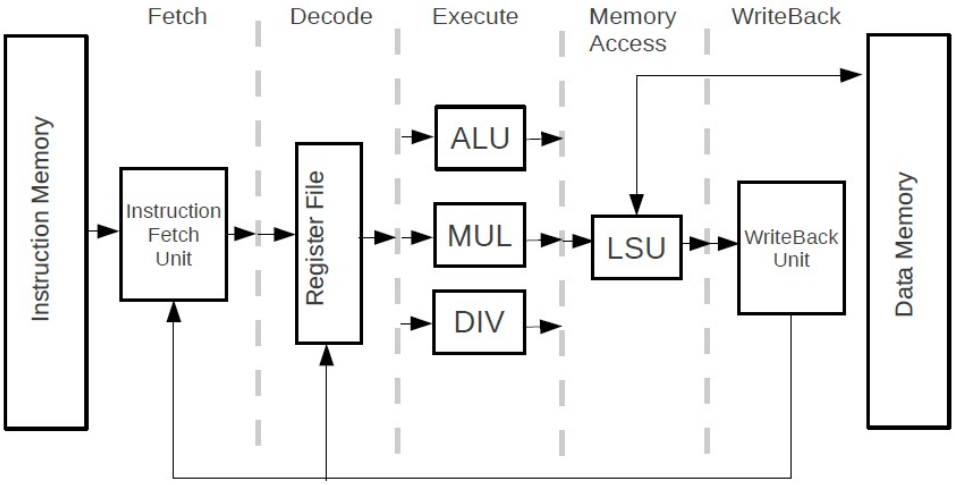

Figure 2.1: Abacus Single Cycle Architecture

Abacus version, it takes one cycle for an instruction to complete all the five steps. This has the problem, that the calculations become slow, since it requires a slow clock so that all steps can finish in one clock cycle.

In *pipelined Abacus* architecture, pipelining is introduced to speed up the execution of programs. Hardware resource(s) used for each instruction execution step is used by a different instruction in the program order. Also additional pipeline storage units are included between each instruction execution step so that one pipeline stage can forward data to the next pipeline stage. Pipelined architecture can be run at a higher clock speed since now only each pipeline stage need to finish in one or more (if the stage is further pipelined) clock cycles instead of all five stages as in the case of single cycle version. Figure 2.2 shows the pipelined Abacus architecture.

Abacus pipelined version resolves RAW (Read After Write) data dependencies between instructions either by stalling the pipeline or using data forwarding. RAW data dependency occurs when an instruction succeeding a previous instruction requires the result of execution of the latter to execute itself. This is explained below.

Assume the below assembler code:

```
(1) ADDI R1,R0,5;
(2) ADD R4,R0,R1;
```

where ADDI and ADD are Abacus instructions explained in detail in the next section. There is a RAW dependency between both instructions, because instruction 2 requires the output of instruction 1, which will be stored in register R1. Instruction 1 will be one pipeline stage ahead of instruction 2. In the ID stage, the operand registers are read. When instruction

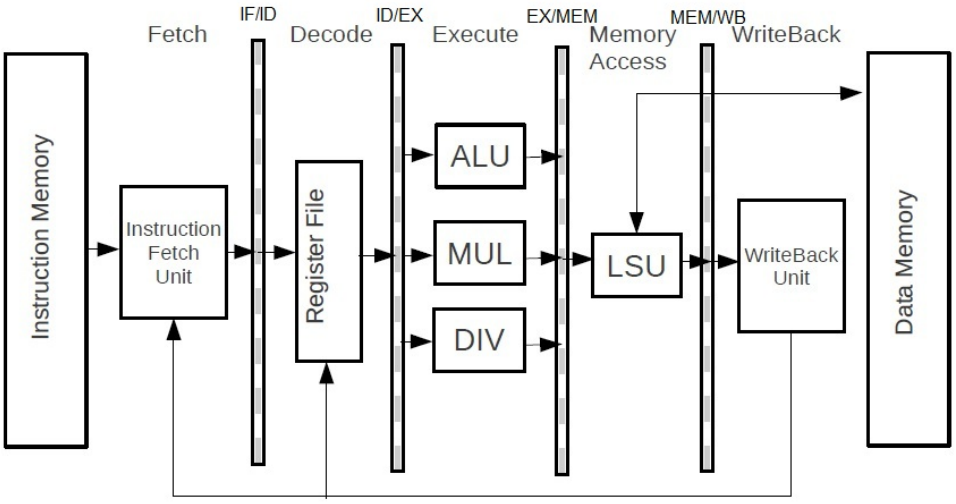

Figure 2.2: Abacus Pipelined Architecture

2 reads data in R1, instruction 1 will be in EX stage, thus it did not yet write back the data to register R1 and hence the second instruction ends up reading the older value in register R1. The writeback will take place two pipeline stages later. Therefore, in stalled version of pipelined Abacus architecture, the instruction 2 is stalled in ID stage until the first instruction writes output data to register R1. Thus ID, EX and MEM stages hardware shall do no operation (NOP) while pipeline is stalled. The following figure illustrates the example.

Another way to resolve the above RAW dependency in the pipeline is by using data forwarding. In data forwarding, the data required for an instruction execution in a pipeline stage is forwarded from one of the following pipeline stages as soon as it becomes available. Therefore in our example, instead of waiting for the first instruction to write back output data to register R1, the same data available at the end of EX stage and is forwarded to the beginning of EX stage, which will then utilize the data in the next cycle. Therefore no stalling is required in this case. This is illustrated in figure 2.4.

It is generally bad practice to stall, because it increases the execution time of a program. Data forwarding in contrast can reduce the number of stalls, but still in some cases, the pipeline has to be stalled.

Assume the following assembler code:

```
(1) ADDI R1,R0,5;
(2) LD R2,R0,100;
(3) ADD R4,R1,R2;
```

Instruction 2 can get the data which will be written in R2 at the end of MEM stage, but the data is required at the beginning of EX stage for instruction 3. Therefore the third instruction

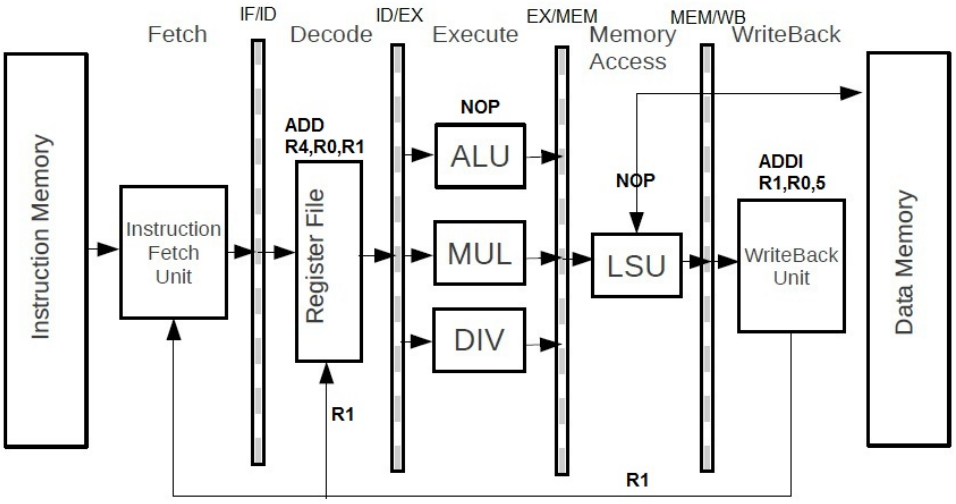

Figure 2.3: Abacus Pipelined with stalling

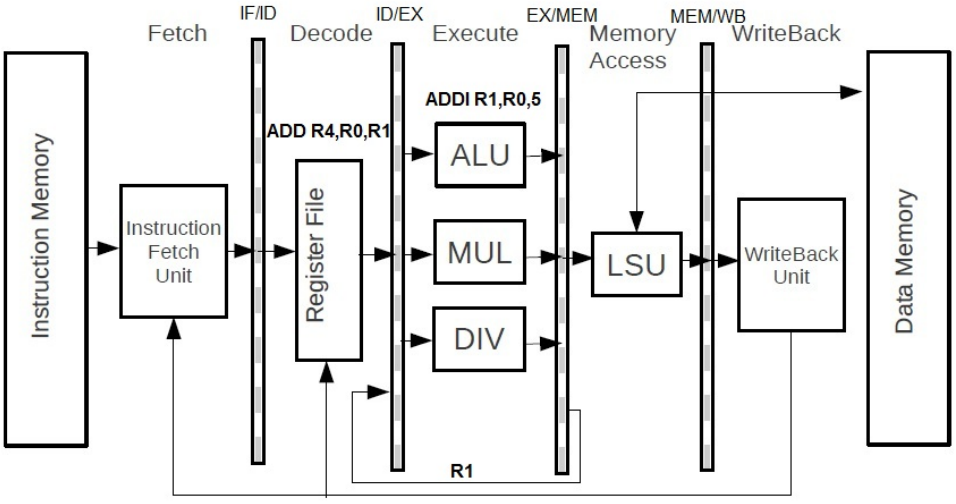

Figure 2.4: Abacus Pipelined with data forwarding

will have to be stalled for one cycle before the data can be forwarded from the end of MEM stage to the beginning of EX stage. An approach to reduce the number of stalls is reordering of the code. In this case toggling instructions 1 and 2 is sufficient to reduce the number of stalls to zero.

The *out-of-order executing Abacus processor* version introduces improvements beyond pipelining. Mainly, the processor is capable to change the order of instructions dynamically during runtime in order to reduce the number of stalls required and maximize hardware utilization. Furthermore, other features such as branch prediction, speculative execution are also implemented. However, these concepts shall not be explained in detail in this document.

There also exists a *VLIW version of Abacus*. VLIW stands for Very Long Instruction Word. In a VLIW processor, an instruction consists of a set of packed operations, that are executed in parallel in hardware. Instructions are statically scheduled. The number of instructions, that can execute in parallel is determined by the total number of functional units, e.g. if there are 4 functional units, the VLIW instruction can have a length of $4 \cdot 16 = 64$ bit, since each Abacus instruction is encoded in 16 bits. One drawback of VLIW architectures is that the code that has been compiled for a specific VLIW architecture can only run on processors of the same type. In other words, the compiled code is not retargetable. Figure 2.5 shows VLIW version of Abacus.

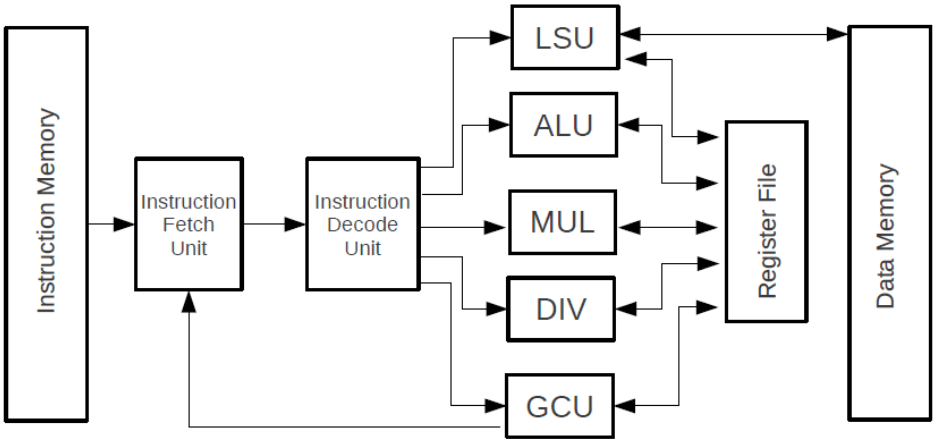

Figure 2.5: Abacus VLIW version

2.2 Instruction Set

The Abacus instruction set is similar to the MIPS instruction set [BhSS14]. Operand values are taken from registers and output values will also be stored in registers. This is called a load-store architecture.

There are four different types of instructions for Abacus, which can be seen in Table 2.1. All of them are stored as 16 bit words $x_{15}...x_0$ in the instruction memory. The opcode determines the type of instruction and is stored in the bits $x_{15}...x_10$. The meaning of the remaining bits $x_9...x_0$ depends on the opcode. In most cases they store register references or immediate values. Register indexes are stored in 3-bits that encode a number between 0 and 7. The different types of instructions supported by the TTA compiler implementation are listed. A notation must first be introduced to distinguish between signed and unsigned operations. Signed values are stored as *2-complement numbers* and unsigned values are stored as *binary radix numbers*.

- $\langle [b_{n-1}, ..., b_0 \rangle_{\mathbb{N}} := \sum_{i=0}^{n-1} b_i \cdot 2^i$ (binary radix-2 number)
- $\langle [b_{n-1}, ..., b_0 \rangle_{\mathbb{Z}} := -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$ (2-complement number)

	$x_{15}...x_{10}$	$x_9...x_8$	$x_7...x_6$	$x_4...x_3$	$x_1...x_0$
R-type	opcode	register index R1	register index R2	register index R3	vector flag
I-type	opcode	register index R1	register index R2	4-bit constant C	
S-type	opcode	register index R1	7-bit constant C		
J-type	opcode	10-bit constant C			

Table 2.1: Classes of Abacus Instructions

R-type instructions always refer to three registers. Table 2.2 lists down the supported R-type instructions, where the register R1 is the result register and the other registers R2 and R3 store the operand values for the instruction. An R-type instruction always result in a state change, since it changes the register content of the processor. The bit x_0 determines if the instruction is of scalar type or vector type. This bit is hardcoded to 0.

I-type instructions consist of a destination register R1, an input operand register R2 and a 4-bit immediate operand C_4. The semantics is very similar to R-type instructions with the difference, that the second operand is stored as an immediate value instead of in register. Also there is no room for a vector flag. Also an I-type instruction always result in a state change, since it changes the register content of the processor. Table 2.3 lists all supported I-type instructions.

S-type instructions only have a destination register R1 and a 7-bit immediate value C_7. The semantics of individual S-type instructions is listed in Table 2.4. S-type instructions also change the processor state by writing to a register.

J-type instructions only consist of a 10-bit immediate value C_{10} as its operand. Therefore they cannot write to one of the general purpose registers and can only be used to change the processors state by changing the value of the program counter PC.

arithmetic operations			
ADD	R1,R2,R3	add signed	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} + \langle \text{Reg[R3]} \rangle_{\mathbb{Z}}$
ADDU	R1,R2,R3	add unsigned	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} + \langle \text{Reg[R3]} \rangle_{\mathbb{N}}$
SUB	R1,R2,R3	subtract signed	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} - \langle \text{Reg[R3]} \rangle_{\mathbb{Z}}$
SUBU	R1,R2,R3	subtract unsigned	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} - \langle \text{Reg[R3]} \rangle_{\mathbb{N}}$
MUL	R1,R2,R3	multiply signed	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} \cdot \langle \text{Reg[R3]} \rangle_{\mathbb{Z}}$
MULU	R1,R2,R3	multiply unsigned	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} \cdot \langle \text{Reg[R3]} \rangle_{\mathbb{N}}$
DIV	R1,R2,R3	divide signed	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} \,/\, \langle \text{Reg[R3]} \rangle_{\mathbb{Z}}$
DIVU	R1,R2,R3	divide unsigned	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} \,/\, \langle \text{Reg[R3]} \rangle_{\mathbb{N}}$
comparison operations			
SLT	R1,R2,R3	set less-than	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} < \langle \text{Reg[R3]} \rangle_{\mathbb{Z}}$
SLTU	R1,R2,R3	set less-than unsigned	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} < \langle \text{Reg[R3]} \rangle_{\mathbb{N}}$
SLE	R1,R2,R3	set less-than-equal	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} \leq \langle \text{Reg[R3]} \rangle_{\mathbb{Z}}$
SLEU	R1,R2,R3	set less-than-equal unsigned	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} \leq \langle \text{Reg[R3]} \rangle_{\mathbb{N}}$
SEQ	R1,R2,R3	set equal	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle = \langle \text{Reg[R3]} \rangle$
SNE	R1,R2,R3	set not equal	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle \neq \langle \text{Reg[R3]} \rangle$
bitwise logic operations			
AND	R1,R2,R3	bitwise and	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle \wedge \langle \text{Reg[R3]} \rangle$
OR	R1,R2,R3	bitwise or	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle \vee$ ¡Reg[R3]¿
XOR	R1,R2,R3	bitwise xor	$\text{Reg[R1]} = \langle \text{Reg[R2]} \rangle \oplus \langle \text{Reg[R3]} \rangle$
load/store instructions			
LD	R1,R2,R3	load from memory	$\text{Reg[R1]} = \text{Mem}[\langle \text{Reg[R2]} \rangle_{\mathbb{N}} + \langle \text{Reg[R3]} \rangle_{\mathbb{N}}]$
ST	R1,R2,R3	store to memory	$\text{Mem}[\langle \text{Reg[R2]} \rangle_{\mathbb{N}} + \langle \text{Reg[R3]} \rangle_{\mathbb{N}}] = \text{Reg[R1]}$

Table 2.2: R-type instructions

arithmetic operations			
ADDI	R1,R2,C_4	add signed immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} + \langle C_4 \rangle_{\mathbb{Z}}$
ADDIU	R1,R2,C_4	add unsigned immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} + \langle C_4 \rangle_{\mathbb{N}}$
SUBI	R1,R2,C_4	subtract signed immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} - \langle C_4 \rangle_{\mathbb{Z}}$
SUBIU	R1,R2,C_4	subtract unsigned immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} - \langle C_4 \rangle_{\mathbb{N}}$
MULI	R1,R2,C_4	multiply signed immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} \cdot \langle C_4 \rangle_{\mathbb{Z}}$
MULIU	R1,R2,C_4	multiply unsigned immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} \cdot \langle C_4 \rangle_{\mathbb{N}}$
DIVI	R1,R2,C_4	divide signed immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{Z}} \,/\, \langle C_4 \rangle_{\mathbb{Z}}$
DIVIU	R1,R2,C_4	divide unsigned immediate	$(\text{Ov,Reg[R1]}) = \langle \text{Reg[R2]} \rangle_{\mathbb{N}} \,/\, \langle C_4 \rangle_{\mathbb{N}}$
load/store instructions			
LDI	R1,R2,C_4	load from memory immediate	$\text{Reg[R1]} = \text{Mem}[\langle \text{Reg[R2]} \rangle_{\mathbb{N}} + \langle C_4 \rangle_{\mathbb{N}}]$
STI	R1,R2,C_4	store to memory immediate	$\text{Mem}[\langle \text{Reg[R2]} \rangle_{\mathbb{N}} + \langle C_4 \rangle_{\mathbb{N}}] = \text{Reg[R1]}$

Table 2.3: I-type instructions

move instructions			
MOV	R1,C_7	move operation	Reg[R1] = $\langle C_7 \rangle_{\mathbb{Z}}$
OVF	R1	move from overflow	Reg[R1] = Ov
branch instructions			
BEZ	R1,C_7	branch if zero	if(Reg[R1]==0) $PC_{next} = PC_{current} + \langle C_7 \rangle_{\mathbb{Z}}$
BNZ	R1,C_7	branch if not zero	if(Reg[R1]!=0) $PC_{next} = PC_{current} + \langle C_7 \rangle_{\mathbb{Z}}$
jump instructions			
JMP	R1,C_7	jump indirect	$PC_{next} = PC_{current} + \langle Reg[R1] \rangle_{\mathbb{Z}} + \langle C_7 \rangle_{\mathbb{Z}}$

Table 2.4: S-type instructions

jump instructions			
J	C_{10}	jump direct	$PC_{next} = PC_{current} + \langle C_{10} \rangle_{\mathbb{Z}}$

Table 2.5: J-type instructions

3 Transport Triggered Architecture

This chapter introduces '*Transport Triggered Architecture*' (TTA) [Hogg96] [CoHA99]. First, the hardware components, their interaction and the principle of execution of a TTA program is explained. It is explained how control flow in a TTA program is achieved before explaining the advantages and disadvantages of TTAs. Finally, as an example, a TTA architecture with the same hardware components as in the Abacus hardware is shown.

3.1 Principle

TTAs are an extreme case of VLIW architectures, where static scheduling is used in its purest form. It consists of function units (FUs), register files (RFs) and the main memory interconnected using a set of data transport buses. The TTA compiler is responsible to generate a sequence of a set of packed data transports, while the TTA hardware executes all data transports in a pack every cycle. FUs execute as a result of data transports. Therefore, in comparison to VLIW architecture, a TTA compiler is also aware of interconnection network details in addition to FU execution latency and RF read/write latencies. The number of parallel executable data transports depends on the number of available buses and also their connectivity to the rest of the hardware. TTA hardware can be either fully interconnected, in which case a bus can interconnect any hardware components or partially connected, in which a bus can only interconnect predefined hardware components.

Register contents are stored in register files (RFs). Any RF can have an arbitrary number of general purpose registers. The capability of a RF is defined in terms of number of reads and/or writes it can perform simultaneously. Function units (FUs) are resposible to execute the operations. A FU, that is capable of executing an operation that requires 'n' inputs and 'm' outputs, will have 'n-1' operand registers, 1 trigger register and 'm' result registers. Data transport to the trigger register of a FU triggers its execution, producing output value(s) in the result register(s) after its predefined latency. Generally a FU can execute several operations like an ALU that can execute any arithmetic or logical operation, but there can also be dedicated FUs carrying out a dedicated task like MUL used for multiplication, DIV used for division etc. To transport data via the buses to a FU/RF operand or trigger register, sockets are required. They connect buses and the ports of RFs and/or FUs. A socket can either read from or write to a port. An example of a TTA hardware components and their interconnection details can be seen in Figure 3.1.

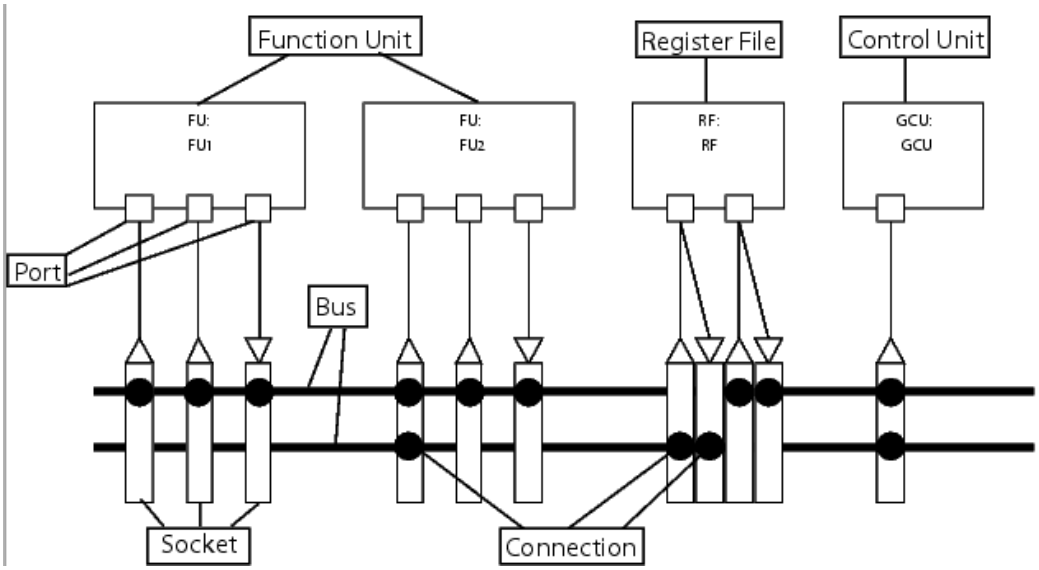

Figure 3.1: Transport Triggered Architecture

3.2 Control Flow

To control the flow of the program, there exist conditional and unconditional branches. A branch has a label to which it will jump, if it is taken. In Abacus assembly code, these labels are represented by strings, that can contain any characters. While in the generated TTA parallel code, we use numbers instead of strings to indicate the target address of a branch. Branch data transports write a new value to the program counter (PC). The PC is made available via a Global Control Unit (GCU).

An unconditional branch consists of a data transport, that writes an immediate value to the PC and is always taken. The immediate value written to the PC is the target address of the branch, assuming the branch is taken.

An conditional branch is taken or not taken depending on whether the condition evaluates to true or false respecitively. TTA programs use *guard expression* as condition to execute a data transport. A guard expression is a boolean expression constructed using only one of the result registers of FU(s). Normally, this shall be the ALU result register holding comparison output or the second result register of DIV function unit holding the remainder of division operation. Data transport with an adjacent guard expression is only executed, if the guard expression evaluates to true. For unconditional branches we shall use PC relative adressing instead of absolute adresses as immediate values.

3.3 Advantages and Disadvantages

TTA offers numerous advantages over conventional microprocessors both from the point of view of hardware architecture and compiler optimizations. TTA architecture is flexible in the

sense that new FUs can be easily plugged into the interconnection network and a function unit can be any hardware block with any number of inputs and outputs. Scheduling data transports to execute a TTA program allows a better utilization of transport resources, register read/write ports. Furthermore, scheduling data transports to execute an operation gives more scheduling freedom. With respect to compiler optimizations, the most important compiler optimization supported by TTA is software bypassing. Assuming two flow dependent moves in the same cycle, the two moves can be scheduled in the same cycle by bypassing the move to an intermediate storage unit. This optimization leads to a lower number of RF reads. It is not possible to make software bypassing work in VLIW architecture. Operand sharing is another TTA-specific optimization where two operations having a common operand and using the same FU can share one operand move. In yet another TTA-specific optimization, operand moves of a commutative operation can be swapped if it leads to a better static schedule.

The main disadvantage of TTA architecture is its low code density. This increases the code size, requiring a larger instruction memory to store TTA programs compared to dynamically scheduled processors. Of course, TTA compilers are much more complex compared to dynamically scheduled processors since in TTA, the responsibility of scheduling the instructions lies with the compiler and not the hardware. The hardware only executes the code. Furthermore, the fact that data transports are statically scheduled rather than operations increases the complexity of TTA compilers.

3.4 Example TTA with Abacus Hardware Components

Abacus hardware components can be used to build a Transport Triggered Archtitecture, which can be seen in Figure 3.2.

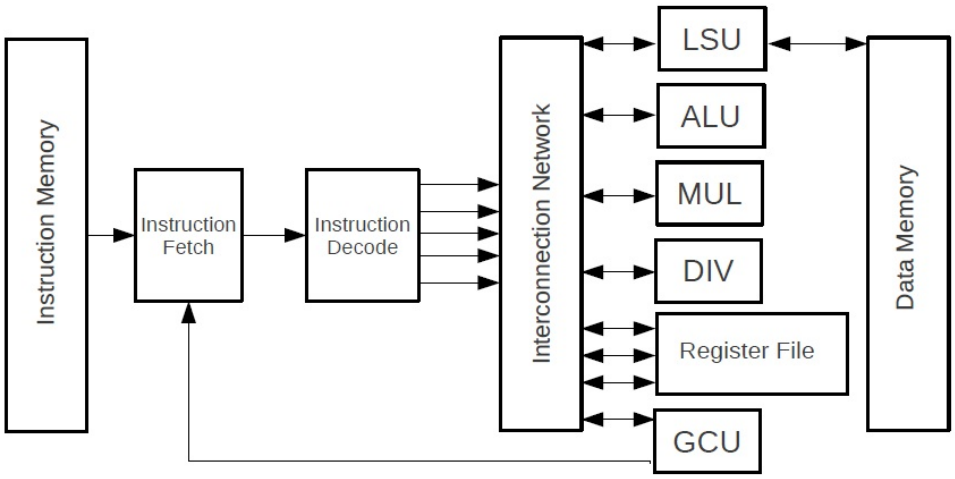

Figure 3.2: An Example Transport Triggered Architecture

ALU, MUL and DIV function units perform arithmetic/logical, multiplication and division

operations respectively as was the case in Abacus hardware architecture. It is important to mention, that there exists no overflow register. Instead the relevant function unit has two output registers. One that holds the value of the calculation and the other one holds the overflown value so that the Abacus instruction OVF can be translated into a set of TTA 'MOV' commands. There exists a Global Control Unit (GCU) for the program counter access and a RF for the eight general purpose registers. There should also be an additional RF for the vector registers, but the TTA compiler implementation is currently only capable to translate scalar Abacus instructions to a set of data transports. For memory access a load-store unit LSU is used, which is a FU from the point of view of TTA.

4 Code Generation

This section explains the architecture and design of the code generator for TTAs. Any given Abacus assembly code is compiled to 'MOV' TTA code as long as that can be run in the particular TTA hardware configured in the hardware description file. Java is used as programming language [**?**].

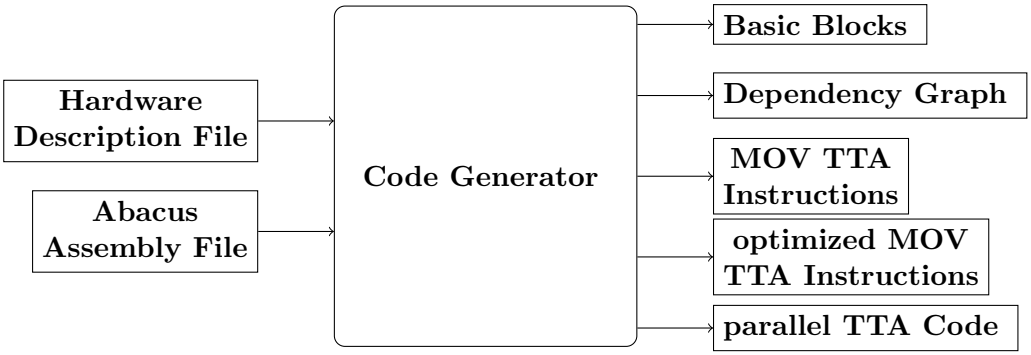

Figure 4.1: Code Generator

Figure 4.1 shows the inputs and outputs of the code generator. A hardware description file describing the target TTA processor and an Abacus assembly file to convert to parallel 'MOV' code are provided as inputs. The code generator starts by identifying the basic blocks, generating a data dependency graph for each basic block and a set of TTA 'MOV' instructions for each Abacus instruction in each basic block. Then the TTA 'MOV' code is optimized and final parallel TTA executable code is generated after applying list scheduling technique.

4.1 Hardware Description File Reader

A Hardware Description File shall be maintained and its reader implementation is required so that it is possible to generate code for any kind of TTA hardware. It should be possible to read all the important values, that the compiler needs. Thus the TTA hardware description file should make clear which components are used and how they are interconnected. Entries in the file are described. [CiSJ07] [TCE]

4.1.1 Transport Bus

Bus parameters	
B-1	Data width
B-2	Bus segmentation

Table 4.1: Bus parameters

Busses are used to connect components. So a list of all buses should be given. A bus has a data width attribute which must be a positive integer number (B-1). It can also be divided into segments (B-2), which means data can only be transported at a segment of the corresponding bus, but there can as well be multiple data transports at different segments of the same bus as long as they do not overlap.

4.1.2 Socket

Socket parameters	
S-1	Bus segment connections
S-2	Socket direction

Table 4.2: Socket parameters

Sockets are used to connect components to buses. It must also be known at which segment of the bus the socket is (S-1). They can be connected to several sources and destinations and are either input or output (S-2). Sources and destinations shall be referred to as ports. A socket does not have an explicit data width, but the busses and ports that it connects to do. It may be the case, that data widths of busses and ports are different and data could be lost. This is case if either width(bus) $>$ width(destination) or width(bus) $<$ width(source).

4.1.3 Address Space

Address Space parameters	
AS-1	Minimum address
AS-2	Maximum address

Table 4.3: Address Space parameters

An address space defines a memory unit and its accessible addresses. There only has to be define a minimum (AS-1) and maximum address (AS-2).

4.1.4 Function Unit

Function Unit parameters	
FU-1	Supported operations of instruction set
FU-2	Input/Output ports as well as socket binding
FU-3	Memory address space (optional)

Table 4.4: Function Unit parameters

Function Units FU support operations of the hardware's instruction set (FU-1). A Function Unit can support multiple operations, that it can execute. For any operation an execution time must be known. To get the operands and to pass the result, input and output ports have to be defined and their corresponding sockets, that they connect to (FU-2). A port can also be bidirectional which means, that it connects to a reading and writing socket. If a FU is a load-store unit LSU, the address space must also be specified (FU-3).

4.1.5 Hardware Operation

Hardware Operation parameters	
HO-1	Parent Function Unit
HO-2	Name of operation
HO-3	Operand/Result binding of Function Unit ports

Table 4.5: Hardware Operation parameters

The FUs operations can only belong to one parent FU (HO-1). It can still be possible, that another FU can do the same operation, but that operation must have its own parameters. An operation is specified by its name (HO-2) as well as its connection between the operands and results of the FU (HO-3), thus it is possible to specify separate FU ports for each input and output of different operations.

4.1.6 Register File

Register File parameters	
RF-1	Type of registers
RF-2	Data width of registers
RF-3	Number of registers
RF-4	Input/Output ports as well as socket binding

Table 4.6: Register File parameters

A Register File RF can be of different types (RF-1). It contains either general purpose registers or reserved registers, that can only be used for compiler optimizations and all registers contain either scalar or vector data. The data width of the registers must also be defined (RF-2) as well as the number of registers, that are contained in the RF (RF-3).

4.2 Abacus Assembler File Reader

To read Abacus assembler code, a reader is implemented that stores all relevant details from the input assembly code. All possible Abacus instruction types, explained in chapter 2.2, must be distinguished and their operands must be stored as well. Furthermore the reader must also check for the validity of the given code. This includes the check for immediate values to be in the expected range. Register allocation is not performed. Instead we assume that the target TTA hardware contains at least the same number of registers required by the Abacus assembly program. In other words, a one to one mapping is done for all register usages from Abacus assembly code to target TTA code. Therefore register allocation done while obtaining Abacus assembly code is used as such without any changes.

4.3 Code Generator

To generate MOV TTA code out of Abacus assembly code, several scheduling steps have to be done. First, control flow graph corresponding to Abacus assembly program should be generated by identifying the Basic Blocks. Next data dependency graph should be generated for each Basic Block. Then each Abacus assembly instruction in each basic block is converted to a set of MOV TTA data transports. All optimizations are then performed for data transports within a basic block followed by finally scheduling them using list scheduling technique.

4.3.1 Basic Blocks

Basic Blocks can be evaluated once every Abacus assembler instruction has been read and all relevant data stored. The beginning of a basic block (bobb) is defined as:

1. The first instruction of a program is a bobb
2. Every instruction that is the target of a branch or jump instruction is a bobb
3. Every instruction that comes immediately after a branch or jump instruction is a bobb

A Basic Block is then the code between two bobb. Once all basic blocks are identified, a control-flow graph (CFG) can be created. The CFG of a program shows the possible control flows in the program via its basic blocks. In other words, nodes in CFGs are basic blocks of the program and directed edges show the possible sequence of execution of basic blocks.

We shall now see with the help of the below example, how basic blocks can be identified and CFG created :

1.	B1	LDI R1,R0,123;
2.		BEZ R1,R0,label;
3.	B2	SUBI R1,R0,3;
4.		ADDI R2,R0,4;
5.		J end;
6.	B3	label: MOVI R5,12;
7.	B4	end: SLTI R2,R5,4;

B1 starts at line 1 due to the first rule for a bobb. Line 3 comes immediately after a branch instruction, thus it is also a bobb. Lines 6 and 7 are both labeled and are target of branch/jump instructions. So they are bobbs as well. The corresponding CFG can be seen in Figure 4.2

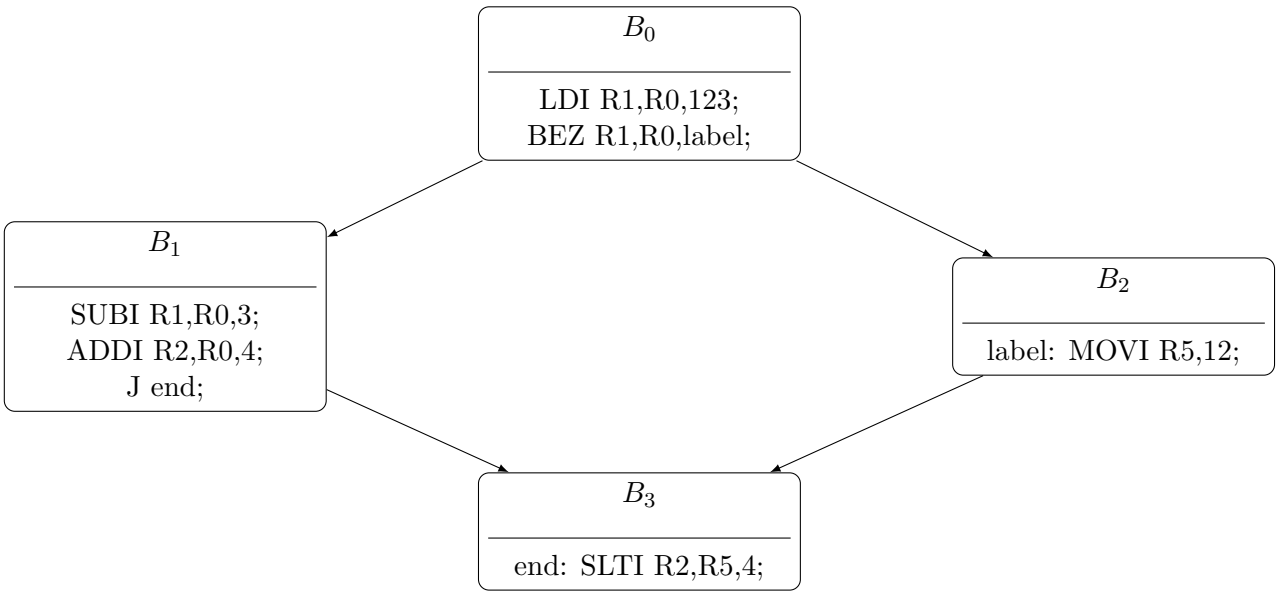

Figure 4.2: Basic Block example

A CFG is generated very easily once all the basic blocks are identified. The successor(s) of all basic blocks define the CFG. There can be between zero and two successors in our case (Abacus assembly program), while there can be any number of successors for a program with more than two-way branching possible. A basic block is a successor of another basic block if it either comes next in program order or if its bobb is the target of the branch or jump instruction, which is the last instruction in the latter basic block.

4.3.2 Dependency Graph

A Data Dependency Graph (DDG) indicates the dependencies between instructions. These instructions can be either anywhere in the program, which would give a global view of all data dependencies, or within the same basic block. Since this TTA code generator implementation does not take advantage of Extended Basic Block Scheduling, data dependencies of instructions within the same basic block are only considered.

An instruction is dependant on another instruction, if one of the following rules fulfills:

1. (RAW): An instruction reads from a register, whereat a previous instruction wants to write data to the same register
2. (WAW): An instruction writes to a register, whereat a previous instruction wants to write data to the same register
3. (WAR): An instruction writes to a register, whereat a previous instruction wants to read data from the same register

Since writes to register R0 does not have any effect on Abacus, the conflicts due to R0 can be ignored.

As an example consider the following code:

1.	B1	LDI R1,R0,123;
2.		BEZ R1,R0,label;
3.	B2	SUBI R4,R1,3;
4.		ADDI R2,R0,4;
5.		SLTI R4,R2,5;
6.		J end;
7.	B3	label: ADD R4,R0,R5;
8.		MOVI R5,12;
9.	B4	end: SLTI R2,R5,4;

In Figure 4.3, the DDG including all instructions in the above example is shown. Solid arrows are dependencies among instructions in the same basic block and dashed ones are dependencies among instructions from different basic blocks.

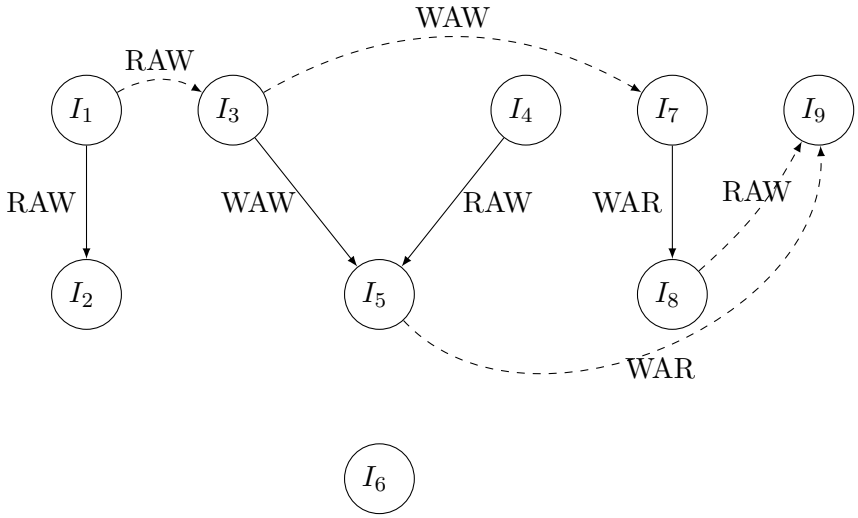

Figure 4.3: Dependency Graph example

4.3.3 Move Instructions

TTA 'MOV' instructions corresponding to any Abacus instruction can be generated at any point of time after reading the Abacus assembly file, since they are independent from basic block identification and data dependency graph generation. This step does not consider any optimizations though. Optimization will be done at a later point of time.

The below tables will show the set of TTA 'MOV' code corresponding to any Abacus instruction type.

arithmetic/comparison/logic R-type operations			
CMD R1,R2,R3;	Reg[R2].id	→	RF_RD1ADR;
	RF_R1DATA	→	FU_OP_CMD;
	Reg[R3].id	→	RF_RD2ADR;
	RD_R2DATA	→	FU_TR_CMD;
	Reg[R1].id	→	RF_WRADR;
	FU_RES1	→	RF_WDATA;

Table 4.7: arithmetic/comparison/logic R-type operations as 'MOV' TTA code

Any R-type instruction besides load and store instructions will be compiled to the code mentioned in Table 4.7. To load the content of a register into a FU, two instructions are required. This is because a RF requires first the id of the register to be loaded. A pointer will then be placed to the corresponding register, which can then be read by the data port. For this implementation it will be assumed, that a RF has at least two ports RF_R1DATA and RF_R2DATA, that can be read. For the writeback a similar approach is used. The port RF_WRADR places a pointer to the register to be written to and the port RF_WDATA writes the data to the corresponding port.
A FU also requires additional information about which operation should be executed. This information is encoded into the name of the FU port, e.g. if an ADD should be executed, the command will write to the ports FU_OP_ADD and FU_TR_ADD. This information is only required, if the FU is capable to do multiple operations.

arithmetic/comparison I-type operations			
CMD R1,R2,Immediate;	Reg[R2].id	→	RF_RD1ADR;
	RF_R1DATA	→	FU_OP_CMD;
	Immediate	→	FU_TR_CMD;
	Reg[R1].id	→	RF_WRADR;
	FU_RES1	→	RF_WDATA;

The only difference between I-type instructions and R-type instructions is, that there is no second register to be loaded into the FU. Instead the immediate value can be directly written to the FU trigger register.

Move instructions only consist of moving an immediate value to a register, thus only two MOV TTA instructions are required. One that changes the pointer to the register to be

move instructions			
MOV/MOVU R1,Immediate;	Reg[R1].id	→	RF_WRADR;
	Immediate	→	RF_WDATA;
OVF R1;	Reg[R1].id	→	RF_WRADR;
	FU_OUT2	→	RF_WDATA;

written to and the other one to write the data. The OVF instruction gets its value from the overflow register from the last used FU, therefore the compiler must keep track of that.

load/store instructions			
ST R1,R2,Immediate;	Reg[R2].id	→	RF_RD1ADR;
	RF_R1DATA	→	LSU_ADR;
	(+/-)Immediate	→	LSU_ADR;
	1	→	LSU_WR;
	Reg[R1].id	→	RF_RD1ADR;
	RF_R1DATA	→	LSU_DIN;
LD R1,R2,Immediate;	Reg[R2].id	→	RF_RD1ADR;
	RF_R1DATA	→	LSU_ADR;
	(+/-)Immediate	→	LSU_ADR;
	0	→	LSU_WR;
	Reg[R1].id	→	RF_WRADR;
	LSU_DOUT	→	RF_WDATA;

Load and store instructions first have to load their address. A register's content is loaded into LSU_ADR. The immediate value is an offset, therefore its content is added relative to the current address. LUS_WR contains a flag if the instruction is writing to the memory and is either 0 or 1. LSU_DIN is used for writing and LSU_DOUT is used for reading commands.

branch instructions			
BRANCH R1,label;	Reg[R1].id	→	RF_RD1ADR;
	(!)RF_R1DATA:(+/-)label	→	PC_TR;

Branch instructions use guarded terms. An instruction of the form "condition:immediate → port" only executes, if the condition is true. Depending on the branch condition, this condition must eventually be negated. The label is an immediate, which is added as an offset to the PC.

jump instructions			
J label;	label	→	PC_TR;
JMP R1,label;	Reg[R1].id	→	RF_RD1ADR;
	RF_R1DATA	→	ALU_OP_ADD;
	label	→	ALU_TR_ADD;
	ALU_RES1	→	PC_TR;

There also exist two different types of jump instructions. The S-type jump instruction "JMP R1,label" requires first to evaluate its jump target by an ALU first, instead the J-type jump instruction can directly write its label to the PC.

4.3.4 Software Bypassing

Software Bypassing is a TTA-specific optimization technique. Data required by a FU for execution is normally accessed from register by reading the register file, but often it is the case that this data is the result of a previous operation performed in the same/another FU and written to the read register. In such cases, if the referred register is not used later in the program, writing to the register can be bypassed and the data can be directly transferred from the output register of the FU to the input (operand or trigger) register of the other FU.

The most simple case should be demonstrated by the following assembler code:

line	assembler code	MOV TTA code	MOV TTA with software bypassing
1	ADDIU R1,R0,1;	0 → ALU_OP_ADDIU;	0 → ALU_OP_ADDIU;
2		1 → ALU_TR_ADDIU;	1 → ALU_TR_ADDIU;
3		1 → RF_WRADR;	1 → RF_WRADR;
4		ALU_RES1 → RF_WDATA;	ALU_RES1 → RF_WDATA;
5	ADDIU R2,R0,2;	0 → ALU_OP_ADDIU;	0 → ALU_OP_ADDIU;
6		2 → ALU_TR_ADDIU;	2 → ALU_TR_ADDIU;
7		1 → RF_WRADR;	
8		ALU_RES1 → RF_WDATA;	
9	ADD R3,R1,R2;	1 → RF_RD1ADR;	1 → RF_RD1ADR;
10		RF_RD1DATA → ALU_OP_ADD;	RF_RD1DATA → ALU_OP_ADD;
11		2 → RF_RD2ADR;	ALU_RES1 → ALU_OP_TR;
12		RF_RD2DATA → ALU_OP_TR;	
13		3 → RF_WRADR;	
14		ALU_OUT1 → RF_WDATA;	

Every TTA 'MOV' instruction writes its data back to the register file without any optimizations. Software bypassing checks any of the writebacks. The first assembler instruction writes to register R1 and uses the ALU as its FU. The second assembler instruction uses the same FU, thus the data of the first assembler instruction would be overwritten, if there is no write-back. Now it must be checked if there are any reads to register R1. Otherwise the data would not be used. Since the third instruction reads register R1, lines 3 and 4 must execute and cannot be saved.

The second command writes to register R2, so again the compiler has to check any future uses of register R2. If register R2 would have been read at some later point of time, it must be written back to its register. In contrast if an instruction overwrites register R2, the compiler no longer has to check any FU usages for this specific instruction. The third instruction applies to both of these requirements. In this case a writeback for register R2 is not necessary as well, since there are no more instructions, that could have used register R2. Hence lines 7 and 8 are unnecessary and can be saved.

The third instruction now reads register R2, which has not been written back. It is still in the output register of the FU, namely port ALU_RES1. Therefore lines 11 and 12 have to

be changed to the new line *ALU_RES1* → *ALU_OP_TR*. Furthermore lines 13 and 14 can be saved, because it is the end of the code and no more reads or writes to register R3 can occur.

In the previous scenario all instructions were in the same basic block, since there are not any branches or jumps. To apply software bypassing for the whole code, every possible path that the code can go through must be evaluated.

line	assembler code	MOV TTA code
1	ADDI R1,R0,1;	0 → ALU_OP_ADDI;
2		1 → ALU_TR_ADDI;
3		1 → RF_WRADR;
4		ALU_RES1 → RF_WDATA;
5	L: ADDI R1,R1,1;	1 → RF_RD1ADR;
6		RF_RD1DATA → ALU_OP_ADDI;
7		1 → ALU_TR_ADDI;
8		1 → RF_WRADR;
9		ALU_RES1 → RF_WDATA;
10		
...		loop body
N-7		
N-6	DIV R1,R2,R3;	2 → RF_RD1ADR;
N-5		RF_RD1DATA → DIV_OP;
N-4		3 → RF_RD2ADR;
N-3		RF_RD2DATA → DIV_TR;
N-2		1 → RF_WRADR;
N-1		DIV_RES1 → RF_WDATA;
N	J L;	5 → PC_TR;

Table 4.8: MOV TTA loop example

The loop example in Table 4.8 consists of two basic blocks. Both of them have the second basic block as a successor, thus software bypassing must be applyable for both of them. However the first basic block has the data from register R1 in the ALU_RES1 register, whereat the second basic block has it in DIV_RES1. Without a case distinction depending on the predecessor software bypassing is not possible.

4.3.5 List Scheduling

List scheduling is a renowned scheduling technique, especially in VLIW architectures [**?**]. We use the same technique for basic block scheduling. The goal is to run several TTA 'MOV' commands in parallel. To do the same, we must be aware of which hardware component can be used and is currently not busy doing any other tasks. Moves that correspond to the same Abacus instruction are not scheduled individually, since this could lead to a poor hardware usage, e.g. if there is a large gap between operand and trigger moves, the FU will be occupied for a very long time and no other instruction can use the same FU during this. Similarly,

data from a result register of FUs have to be moved as soon as possible for the same reason.

Instructions are scheduled at assembly level and we use the data dependency graph, that has been created before. In addition to the data dependency, it is also required to assign resources properly. Resources that have to be assigned are FUs, busses and sockets. All of them are assigned by a first-fit assignment, which means "the first free resource that fits is selected". For FUs it is also desirable to always select the most specialized one, if there exist multiple FUs that can perform the same operation, so that the more general ones can be used for other tasks. When a FU gets assigned to an instruction, it will be occupied for a time equal to the latency of the operation. This delay is specified in the hardware description file for every hardware component. Since we assume a fully interconnected TTA hardware, it is not necessary to assign busses. Instead the number of busses simply denotes the number of move instructions that can execute in parallel. For sockets we also assume that each port of each FU is connected to every bus via a separate socket. Thus we do not need to consider socket availability also while doing list scheduling.

There are two possible delays that can occur. The first one being an occupied FU and the second one being a data dependency.

line	assembler code	optimized MOV TTA code	occuring delays
1	ADDI R1,R0,1;	0 → ALU_OP_ADDI;	→ ALU reserved until trigger
2		1 → ALU_TR_ADDI;	→ ALU occupied for delay(ADDI)
3	ADDIU R2,R0,4;	0 → ALU_OP_ADDI;	← depends on ALU → ALU reserved until trigger
4		4 → ALU_TR_ADDI;	← depends on ALU → ALU occupied for delay(ADDIU)
5	DIV R3,R2,2;	ALU_RES1 → DIV_OP;	← depends on R2
6		2 → DIV_TR;	→ DIV_FU occupied for delay(DIV)

The above example makes clear:

- A FU is reserved after an operand move to the FU until the trigger move to the same FU
- A trigger move occupies a FU by the delay specified for the operation or FU execution
- Data dependencies delay an instruction until the previous instruction (on which the current instruction depends) finishes

All instructions that cannot execute will be stored in a FIFO buffer. As long as this buffer is not empty the compiler will check, if any of its instructions can be executed. If the compiler reaches a new basic block, the FIFO buffer must first be empty before proceeding. When the list scheduling is finished, the compiler should have created a list of parallel executable TTA 'MOV' code. All parallel executable move instructions are separated by a comma and a semicolon indicates, that the next instructions will be done in the next step.

5 Implementation Details

After explaining code generation in general this chapter explains the implementation details of the compiler. First the file formats have to be specified and how they are read and stored. The scheduler then uses these information and applies the techniques explained in section 4.3.

5.1 File Format Specifications

The code generator reads a hardware description file and an Abacus assembly file. The output file contains TTA 'MOV' code. These formats are explained in this section.

5.1.1 TTA Hardware Description File Format

The TTA hardware description file is maintained in an XML file format, which is used to specify the hardware, that can be used. Any content that should be read must be within ¡adf¿ tags.
Chapter 4.1 gave an introduction of what should be the content of a hardware description file. In this section the syntax and annotation shall be defined.

5.1.1.1 Transport Bus

```
<bus name="name">
  <width>\#</width>
</bus>
```

Busses have a specified data width and also a bus segmentation. However bus segmentation is ignored by the current implementation, since it is only capable to deal with fully interconnected hardware. An exception will be thrown, if the data width is not uniquely defined.

5.1.1.2 Socket

```
<socket name="name">
  [<writes-to>
    <bus>bus name</bus>
    ...
  </writes-to>]
```

```
  OR
  [<reads-from>
    <bus>bus name</bus>
    ...
  </reads-from>]
</socket>
```

Buses are connected via sockets, that either write to or read from several buses. Since bidirectional sockets are not supported, an exception must be thrown in this case.

5.1.1.3 Address Space

```
<address-space name="name">
  <min-address>\#</min-address>
  <max-address>\#</max-address>
</address-space>
```

An address space is uniquely defined by its name and has a minimum and maximum address.

5.1.1.4 Function Unit

```
<function-unit name="name">
  <port name="name" id="id">
    <connects-to>socket name</connects-to>
    <width>\#</width>
    [<triggers/>]
  </port>
  ...
  <operation>
    <name>operation name</name>
    <delay>\#</delay>
  </operation>
  ...
  <address-space>
  address space name
  </address-space>
</function-unit>
```

The FUs ports are currently limited to the implementation. The implementation requires an id for a port, which defines its type.

id		semantics
0	→	Left Operand
1	→	Right Operand
2	→	Output
3	→	Overflow

A function unit also has a list of operations. An operations name defines its type, e.g. an operation with the name ADD can execute the ADD Abacus instruction. Any operation has a delay on the FU, when it is executed. In case the FU is a load-store unit an address space has to be specified.

5.1.1.5 Register File

```
<register-file name="name">
  <size>\#</size>
  <width>\#</width>
  <port name="name" id="id">
    <connects-to>socket name</connects-to>
    <width>\#</width>
  </port>
  ...
```

A register file's size defines the number of registers. The width applies to all register's data widths. As for FUs the port id defines its type.

id		semantics
0	→	First Readable Operand's Pointer
1	→	First Readable Operand's Value
2	→	Second Readable Operand's Pointer
3	→	Second Readable Operand's Value
4	→	Writable Operand's Pointer
5	→	Writable Operand's Value

5.1.2 Abacus Assembly File Format

The Abacus file format has to be specified. These are the rules for the file format:

1. All instructions end in semicolon
2. A comment line starts with //
3. Only one instruction per line is allowed
4. The instruction format is "label: instruction". "label: " is optional and only instructions are allowed, that the compiler has implemented.
5. Operations use capital letters (e.g. ADD, SUBU, MOV)
6. NOP must be explicitly given as an instruction

7. Any number of whitespaces or tabs can be used for formatting an instruction line for readability

5.1.3 TTA MOV Assembly File Format

There are not just TTA 'MOV' instructions which are generated, but also basic blocks and a dependency graph. For these cases the output format must be specified. An instruction has the form:

1. All instructions end in semicolon
2. A comment line starts with //
3. Parallel executable instructions are separated with a comma
4. The instruction format is "source_port → destination_port" or "immediate → destination_port" or "NOP"
5. Ports use capital letters

The file that contains the basic blocks and the dependency graph contains Abacus instructions. Thus the Abacus file format is used with additional information.
Basic blocks are named and consecutively numbered as "BB#". The instructions are separated and stand below their corresponding basic block. At the beginning of the file the control flow can be seen. "BB#0 → BB#1" means BB#1 is a successor of BB#0. If BB#0 is empty the last used basic block is meant instead.
The dependency file numbers all instructions. A dependency has the form

```
instruction number
| RAW: list of RAW dependant instructions
| WAW: list of WAW dependant instructions
| WAR: list of WAR dependant instructions
```

All dependencies are listed above the code.

5.2 Reading the Hardware Description File

An XML parser is implemented in the code generator, but any XML parser can be used to read the ADF file. Figure 5.1 shows a simplified UML diagram of the classes used in the implementation to read the hardware description file.

The hardware description file reader (ADFReader) creates the hardware with its components. Any component has its own class and subcomponents are saved as attributes. The purpose of the hardware is to give the scheduler information about its usage and limitations.

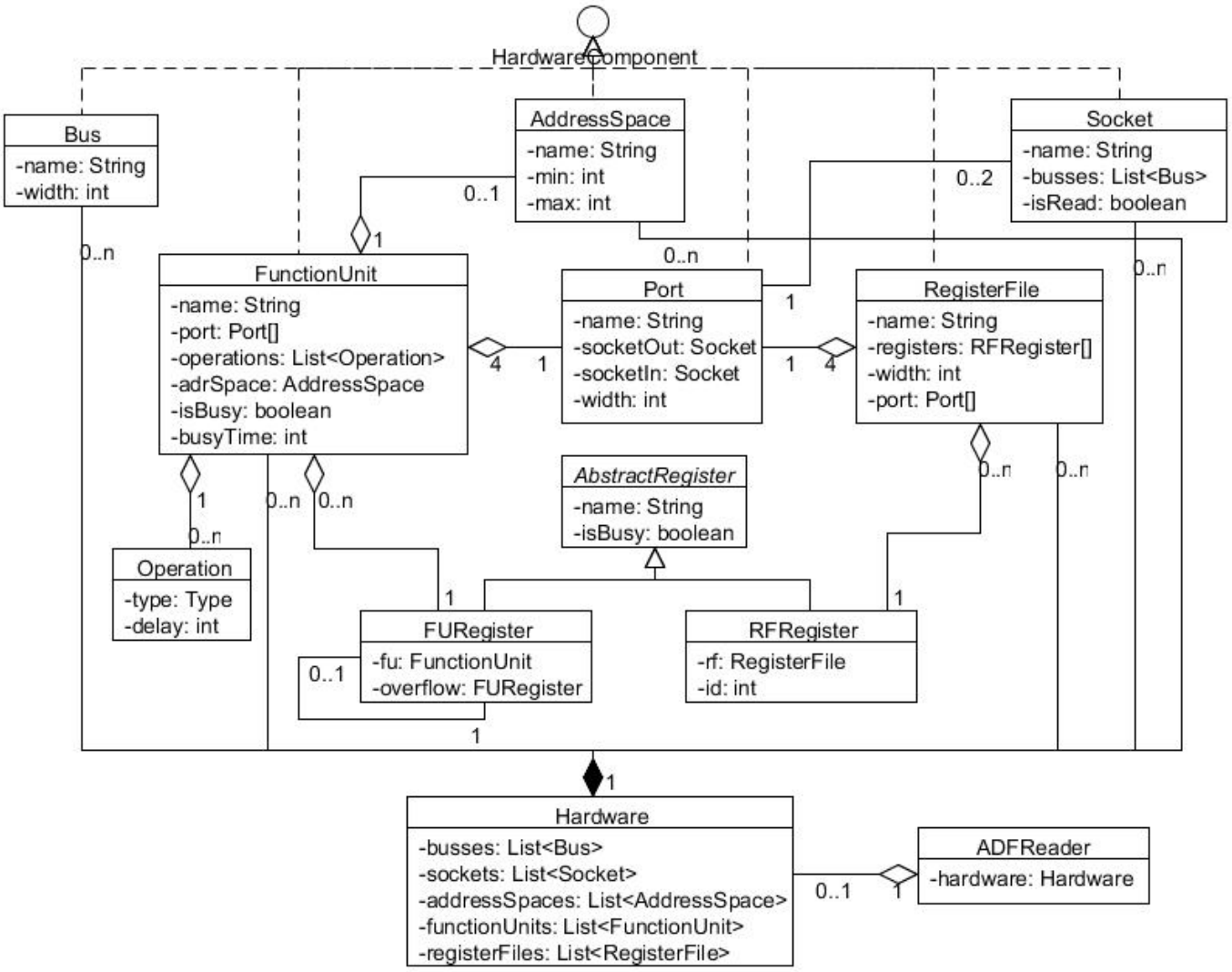

Figure 5.1: UML of Hardware Components

5.3 Reading the Abacus Assembly File

Figure 5.2 illustrates the Abacus assembly file reader in a simplified UML diagram. Instructions are stored in the AssemblerCode class, which can then be used by the scheduler. Instructions are distinguished by their type and/or by their usage. R- and I-type instructions use the same class RICommand, whereat S-type instructions use the SCommand class. Load and store instructions refer to the MemoryCommand class. Branch and jump commands additionally have a destination label.

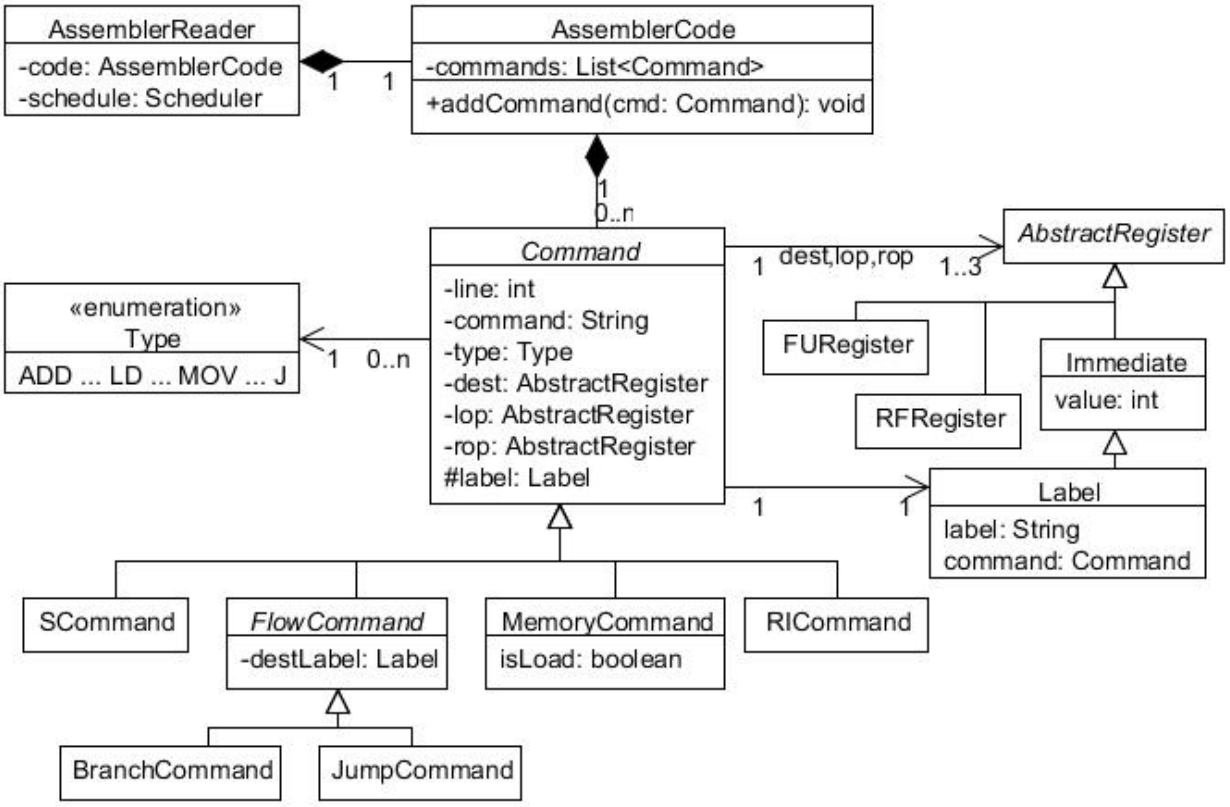

Figure 5.2: UML of Abacus Assembly Reader Components

5.4 Code Generation

This section is about the implementation details of the code generator. Figure 5.3 shows a simplified UML diagram of all important classes. The implementation of all scheduling steps are explained. All steps correspond to a subsection of section 4.3. The previous steps of reading the hardware description file and the Abacus assembly file already created all the instructions. The class Scheduler can now use the AssemblerCode object to do all scheduling steps.

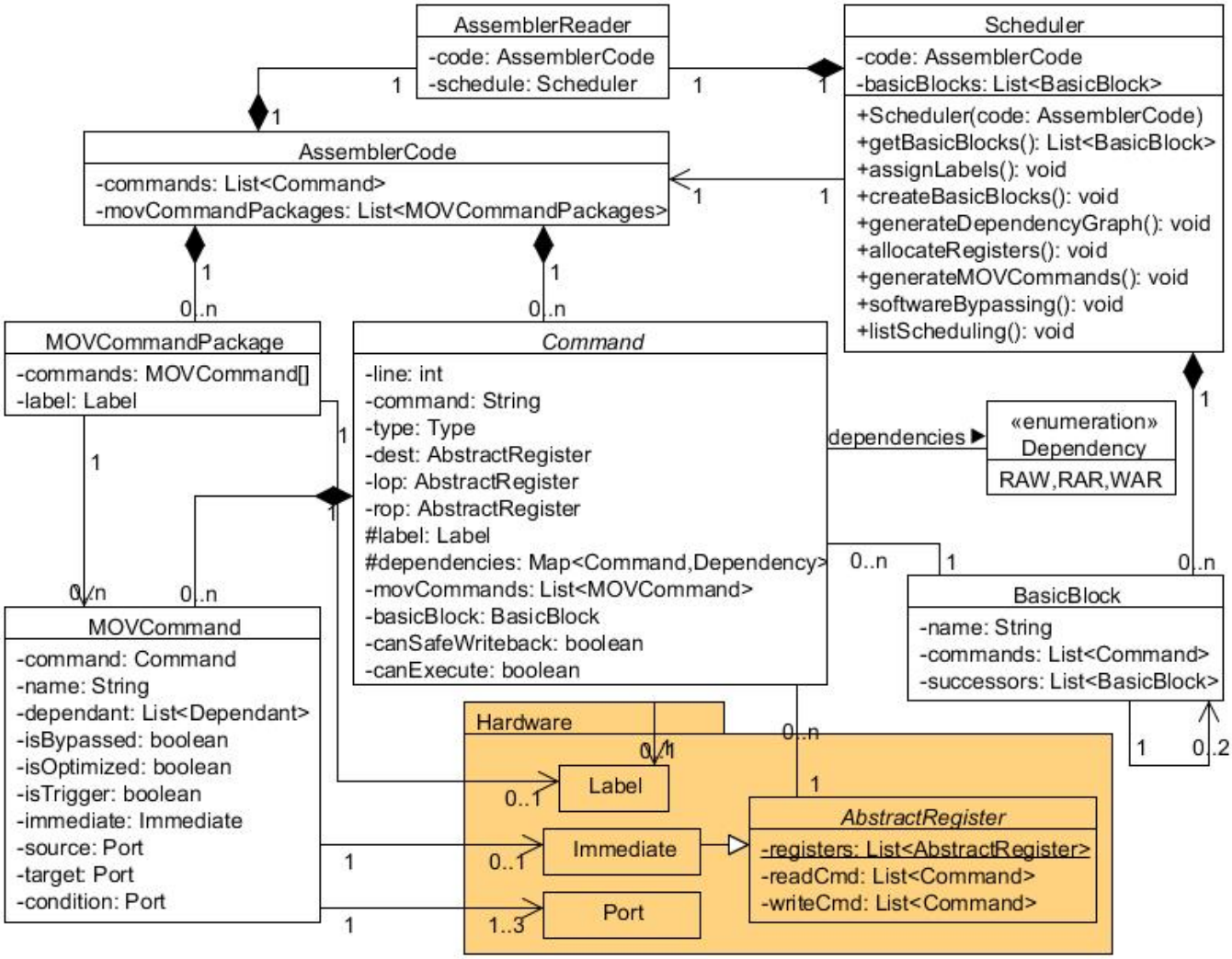

Figure 5.3: UML of Code Generation Components

5.4.1 Identify Basic Block

Basic blocks are stored within the Scheduler object. Beginning of basic blocks (bobb) have to be identified. There are either unconditional or conditional branches. Depending on that, the next basic block in program order is either a successor or not. Bobbs are easily identified by this code snippet:

```
1 for (Command command : code.getCommands()) {
2   if (createNext || command.hasLabel()) {
3     createNext = false;
4     // add basic block to list
5   }
6   if (command instanceof FlowCommand) {
7     createNext = true;
```

```
  }
}
```

5.4.2 Generate Dependency Graph

At creation of a Command object the involved registers store it eventually as a reading or writing command. Then the scheduler can loop through all registers and compare reading and writing commands to find dependency conflicts.

```
for (AbstractRegister register : AbstractRegister.getRegisters()) {
  // write after read
  for (Command cmd : register.getReadingCommands()) {
    for (Command c : register.getWritingCommands()) {
      if (c.getLine() > cmd.getLine() && !cmd.hasDependency(c)
          && c.getBasicBlock().equals(cmd.getBasicBlock())) {
        cmd.addDependency(c, Dependency.WAR);
      }
    }
  }
  // write after write
  ...
  // read after write
  ...
}
```

5.4.3 Generate Move Commands

A Command object has a list of its corresponding TTA 'MOV' commands. These MOV-Command objects are created in this step. A command's type can be used to make a case distinction. The cases are listed in section 4.3.3 and can be used in exactly the same way.

5.4.4 Apply Software Bypassing

Software bypassing is applied to every path of the program. A basic block knows its successors, thus every path can be evaluated. Three different HashMaps are used to store all relevant values:

```
HashMap<FunctionUnit, Command> fuMap;
HashMap<AbstractRegister, Command> registerMap
HashMap<Immediate, Port> portMap;
```

FuMap takes a FU as key and returns the instruction, that last used the FU. If a FU is reused, registerMap takes the destination register of the instruction being overwritten as key and returns the instruction being overwritten as a value, therefore it uses fuMap. The registerMap's content is important to know if a writeback instruction cannot be skipped. PortMap uses a registers id as key and returns the FU port, which contains the content of the register. To replace a TTA 'MOV' instruction the value of the portMap can then be used.

5.4.5 List Scheduling

For list scheduling MOVCommandPackage objects are created. They can contain as many instructions as busses exist, since a fully interconnected hardware is assumed. A FU has a busy flag and an instruction has a canExecute flag. Every time an instruction triggers these flags will be set. If an instruction cannot execute, it will be added into a FIFO (first-in first-out) buffer. As long as this buffer is not empty, it will try to execute as many instructions as possible.

6 Conclusion and Future Work

A code generator for TTAs was implemented, which converts an Abacus assembly program to parallel TTA 'MOV' data transports for execution on a TTA with the configuration specified in a hardware description file. This report describes the architecture and design of the code generator after introducing Abacus and TTA hardware architectures. Later implementation details are also explained. However the code generator has a few limitations. The current implementation is only capable of using a fully interconnected TTA hardware. Furthermore register allocation is not implemented. It is assumed that TTA hardware contains a register file with at least as many registers available as available in Abacus hardware. To this end, software bypassing is the only TTA-specific optimization implemented in our code generator, although there are other optimizations such as operand sharing, operand swapping etc. In the future work, we intend to remove these limitations.

Currently, the code generator does instruction scheduling only within the basic block scope, using the list scheduling technique. Future work must address scheduling at extended basic block level and software pipelining (optimizing loops), which are important to obtain optimized parallel TTA code. Before producing the final parallel TTA 'MOV' code, all the results at intermediate steps (like list of basic blocks, control flow graph, data dependence graph, 'MOV' data transports corresponding to operations) are written to files, which makes it easier to understand the individual steps and to debug the code generator.

Bibliography

[BeRo91] David Bernstein and Michael Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 241–255, New York, NY, USA, 1991. ACM.

[BhSS14] N. Bhardwaj, M. Senftleben, and K. Schneider. Abacus - the Processor Architecture, July 2014. unpublished.

[CiSJ07] A. Cilio, H.J.M. Schot, and J.A.A.J. Janssen. Architecture Definition File - Processor Architecture Definiton File Format for a new TTA design framework, October 2007. Revision 1.7.

[CoHA99] H. Corporaal, J. Hansson, and M. Arnold. Computation in the Context of Transport Triggered Architectures, August 1999.

[Hogg96] J. Hoggerbrugge. Code Generation for Transport Triggered Architectures, October 1996.

[JAV] Java Programming Language. https://www.oracle.com/java/index.html.

[TCE] TCE - TTA-based Co-design Environment v1.9. http://tce.cs.tut.fi.