Implementation of Infineon TriCore 1.3 model in ArchC

Diogo José Costa Alves

25th August 2005

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of technological processes maintains its exponential growth; 810 Mtrans/chip in 2003 will become 2041 Mtrans/chip in 2007. This obliges an increase in the designer productivity, from 2.6 Mtrans/py in 2004 to 5.9 Mtrans/py in 2007, that is, a productivity increase of 236% in three years[2]. Most of these new products will be embedded System-on-Chip (SoC) and include embedded software. In fact, embedded software new routinely accounts for 80% of embedded system development costs[5].

Today, most embedded systems are deigned from RT level description for HW part and the embedded software code separately. Using a classical top-down methodology (synthesis and compilation) the implementation is obtained. The 2001 International Technology Roadmap for Semiconductors (ITRS) predicts the substitution (during upcoming years) of waterfall methodology by an integrated framework where co-design, logical, physical and analysis tools operate together. The design step being where the designer envisages the whole set of intended characteristics of the system to be implemented, system-level specification acquires a key importance in this new design process since it is taken as the starting point of all the integrated tools and procedures that lead to an optimal implementation[5, 2].

The lack of an unifying system specification language has been identified as one of the main obstacles bedeviling SoC designers[7]. Among the different possibilities proposed, languages based on C/C++ are gaining a wider consensus among the designer community[14], SystemC[3] being one of the most promising proposal.

In order to accelerate the SW design a model of development platform is necessary. ArchC is an open-source SystemC-based architecture description language ADL that is specialized for processor architecture description, its primary goal is to provide enough information, at the right level of abstraction, in order to allow users to explore a new architecture and automatically generate software tools like assemblers, simulators or even compiler back-ends.

This work has the objective to develop a functional model of processor Infineon(R) TriCore in ArchC. The Infineon(R) TriCore is a 32-bit microcontroller-

DSP architecture optimized for real-time embedded systems. The TriCore Instruction Set Architecture (ISA) combines the real-time capability of a microcontroller, the computational power of a DSP, and the high performance/price features of a RISC load/store architecture. The architecture supports both 16-bit and 32-bit instruction formats. The 16-bit instructions are a subset of the 32-bit instructions, chosen because of their frequency of use.

The contents of the work are as follows. In this chapter, motivation and objectives have been presented. In chapter 2, the ArchC ADL resources are briefly explained in order to show how to write an executable processor model. In chapter 3, an architecture overview of the embedded processor Infineon TriCore 1.3 will be presented. In chapter 3, the development details of Infineon TriCore 1.3 ArchC model will be presented. In chapter 4, some performance comparison of processor model written in ArhcC, an instructions set simulator (ISS) and a TC1MP-S core integrated on the FPGA Spyder System will be provided. Finally, the conclusions will be drawn in chapter 5.

Chapter 2

The Architecture Description Language ArchC

In this chapter we introduce the architecture description language (ADL) called ArchC. The information on this chapter was collected of ArchC manuals and articles, which are cited in elapsing of the text.

ArchC is an open-source SystemC-based ADL that is specialized for processor architecture description. Its primary goal is to provide enough information, at the right level of abstraction, in order to allow users to explore a new architecture and automatically generate software tools like assemblers, simulators or even compiler back-ends. It is a simple language that follows SystemC syntax style, which makes clear the connection between these two languages.

ArchC key features are a storage-based co-verification mechanism that automatically checks the consistency of a refined ArchC model against an ArchC reference description, memory hierarchy modeling capability and the possibility of integration with other SystemC IPs.

An architecture description in ArchC is divided in two parts: the Instruction Set Architecture (AC_ISA) description and the Architecture elements (AC_ARCH) description. Into AC_ISA description, the designer provides to ArchC details about instruction formats, size and names combined with all information necessary to decoding and the behavior of each instruction. The AC_ARCHC description informs ArchC about storage devices, pipeline structure, etc. Based on these two descriptions, ArchC will generate a behavior simulator for the architecture in SystemC. The ArchC tools run on a GNU/Linux environment and on MS-Windows with CYGWIN installed.

Up to this point[15], functional and cycle-based simulators have been synthesized starting from ArchC descriptions for the MIPS, Intel 8051, PIC 16F84 and SPARC processors. Functional models for the Texas TMS320C62x, Intel XScale, Motorola ColdFire, PowerPC, Altera NIOS, OpenCores OR1k, and Hitachi SH-4 where also developed in ArchC, some of them are still in the verification phase.

Figure 2.1: Architecture Resources of TriCore in `tricore13.ac` file

```
AC_ARCH(tricore13) {
  ac_wordsize 32;
  ac_fetchsize 16;

  ac_regbank DB:32;
  ac_regbank AB:32;
  ac_reg PCXI;
  ac_reg PSW;
  ac_reg SYSCON;
  ac_reg BIV;
  ac_reg BTV;
  ac_reg ISP;
  ac_reg ICR;
  ac_reg FCX;
  ac_reg LCX;
  ac_cache MEM:35M;

  ARCH_CTOR(tricore13) {
    ac_isa("tricore13_isa.ac");
    set_endian("little");
  };
};
```

2.1 Describing Architecture Resources

ArchC uses some structural information about the resources available in the architecture in order to automatically generate a simulator. The designer must provide an information in the `AC_ARCH` description, which is basically composed of storage elements and pipeline declarations.

The level of details used for this description will depend on the level of abstraction that designer wants for his model. An architecture resources description starts with the keyword `AC_ARCH`. The designer is supposed to inform a name for the project, like is usually done for modules in SystemC. The follows the storage device declarations. In the TriCore Model architecture resources description, showed in Figure 2.1, we have used the keyword `ac_cache` to declare instruction and data caches. The number after the colon represents the size in bytes, of the device. Following with the description, a register file is declared through the keyword `ac_regbank`. We need 32 data registers, 32 address registers and 9 specials registers to model the TriCore. The declaration of `ARCH_CTOR` constructor finishes the `AC_ARCH` description and uses the `ac_isa` keyword to inform in which file the ArchC pre-processor will find `AC_ISA` description.

In order to get a more detailed model, like a cycle-accurate model the designer must provide more information about the architecture. The designer is allowed to declare the word-size for the architecture using the `ac_wordsize` keyword. Memories and caches are byte-addressed but their return type is always a word. If not declared, ArchC assumes that the word-size is 32 bits by default.

Figure 2.2: Instructions and format declaration for TriCore in `tricore_isa.ac` file.

```
AC_ISA(tricore13) {
  //16bit opcodes
  ac_format SB   = "%op1:8 %disp8:8";
  ac_format SBC  = "%op1:8 %disp:4 %const4:4";
  ...

  //32bit opcodes
  ac_format ABS  = "%op1:8 %s1:4 %off18_17_14:4 %off18_5_0:6
                    %off18_13_10:4 %op2_2b:2 %off18_9_6:4";
  ac_format RR   = "%op1:8 %s1:4 %s2:4 %n:2 %RESERVED:2 %op2_8b:8 %d:4";
  ...

  //instruction instantiation
  ac_instr<RR> add__dc_da_db, add_a__ac_aa_ab, ...;
  ...

  ISA_CTOR(tricore13) {
    add__dc_da_db.set_asm("add d%d, d%s1, d%s2");
    add__dc_da_db.set_decoder(op1=0x0B, op2_8b=0x00);
    ...
  }
}
```

2.2 Describing the Instruction Set Architecture

The `AC_ISA` description provides ArchC with all information it needs to automatically synthesize a decoder, along with the behavior of each instruction the architecture. This description is dived in two files, one containing the instructions and format declarations and another containing the instruction behaviors.

2.2.1 Providing instructions and format declarations

Figure 2.2 shows the `AC_ISA` description extracted from our TriCore model. The beginning is similar to the `AC_ARCH` description, starting with the keyword `AC_ISA`, along with the name of the project. The designer continued by declaring the instruction formats. This is done by using the `ac_format` keyword. Next step is to declare instructions. Every instruction must have a previously declared format associated to it.The designer declares an instruction through the keyword `ac_instr` and he can assign it a format using a syntax similar to C++ templates. In the TriCore model, format RR is associated to instruction add__dc_da_db. This allows the designer to access each instruction field individually when describing instruction behaviors.

The description follows with th `AC_ISA` constructor declaration (`ISA_CTOR`). This is here the designer will have to initialize some key values for each instruction. An assembly syntax is assigned to an instruction using the set_asm

Figure 2.3: Instruction Behavior of `add__dc_da_da` in `tricore13-isa.cpp` file.

```
void ac_behavior( add__dc_da_db ){
  dprintf("add d%d, d%d, d%d",d, s1, s2);
  sc_int<33> result = ((sc_int<32> )DB.read(s1)) + ((sc_int<32> )DB.read(s2));
  DB.write(d, result );
  set_PSW( result );
}
```

method. The decodification sequence is a key element to the automatic generation of an instruction decoder. It is provided to ArchC though the `set_decoder` method, and is composed by sequence of pairs `<field_name = value>`. In Figure 2, examine the `add__dc_da_db` instruction. The call to add__dc_da_db.set_decoder method states that a bit stream coming from memory actually is an `add__dc_da_db` instruction if, and only if, fields `op1` and `op2_8b` contain the values `0x0B` and `0x00`, respectively. The decoder will know where to look for the field values in the bit stream, because this information is available in the format previously associated to the instruction.

2.2.2 Providing Instruction Behavior

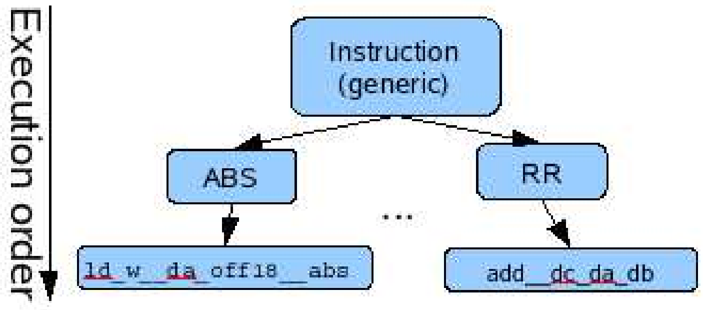
The behavior description file is where the designer provides a description of which operations are executed by each instruction in the architecture. By issuing both description files introduced so far, `AC_ISA` and `AC_ARCH`, to ArchC preprocessor (`acpp`), the designer gets a template of the behavior description file. This template is a skeleton of a `.cpp` (SystemC source) file where the designer is going to fill out the behavior method of each instruction in the architecture. The behavior description file is named as `project_name-isa.cpp.tmpl`, by default.

This behavior information can be expressed in several levels of abstraction. For example, at the very early stages of the design, cycle-accuracy may not be important. Normally, the first model of a new architecture does not have timing information. So, for this preliminary model, the behavior of a given instruction is just a sequence of C++ statements representing the operations that this instruction would execute in the hardware. Figure 2.3 illustrates how we could model the behavior of `add__dc_da_db` instruction in the TriCore architecture. At this abstraction level, the simulator will execute one instruction per cycle, so it is not necessary to worry about data hazards and forwarding, because every time an instruction begins its execution, the previous will have been completed for sure. A so high-level modeling style will provide the designer with an executable specification of the architecture very rapidly, but this specification will be suitable only for experimenting with the instruction set, not for performance measurements, due to its lack of timing information.

Figure 2.4: Generic behavior for ABS format instructions.

```
/* ABS Generic Instructions
     * Result: Compute effective absolute address for ABS type instructions
     */
     void ac_behavior( ABS ){
       EA_abs.range(31, 28) = off18_17_14;
       EA_abs.range(27,14) = 0;
       EA_abs.range(13,10) = off18_13_10;
       EA_abs.range(9,6) = off18_9_6;
       EA_abs.range(5,0) = off18_5_0;
     }
```

Figure 2.5: Execution hierarchy



2.2.3 Providing Format and Generic Instruction Behavior

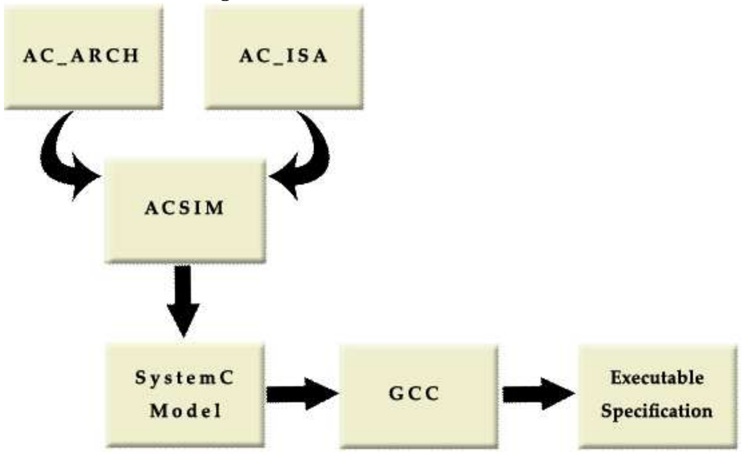
Often , there are many instructions in a particular architecture that execute exactly the same task as part of their behavior. As mentioned above, in the TriCore microcontroller, an instruction of ABS type need always calculate a effective address to access the memory. As a consequence, a piece of code to check that would have to be inserted into the behavior method of every instruction in this class, showed in Figure 2.4.

In ArchC, instead of repeating the code for every instruction, the designer is able to use formats in order to factor out this behavior, Figure 2.5 show a example, writing it once and using it for whole class of instructions. This is possible because ArchC provides the designer with the possibility of overloading the ac_behavior method so that if can take an instruction format as argument.

There is still another situation, where the designer wants that all instruction in the architecture execute a piece of code before running its own behavior method. This is also possible in ArchC, by describing a generic instruction behavior, i.e., a behavior method that belongs to all instructions. Following the same style used above, the designer has to pass the keyword instruction as the argument of the behavior method:

The hierarchy of behavior methods in ArchC states that the simulator will, at a given simulation time, start the execution by the generic instruction be-

Figure 2.6: ArchC Simulator Generation Flow



havior method, followed by the behavior of the format corresponding to the current instruction and, finally, the current instructions behavior itself. The same sequence is performed by each pipeline stage in the case of a pipelined architecture. Through this ArchC behavior hierarchy, we were able to factorize a lot of operations that would have to be repeated into several instruction behaviors, what ended up saving a great amount of redundant code in our models.

2.3 The ArchC Simulator Generator

We called as ArchC Simulator Generator (`acsim`) the tool that takes an ArchC description, composed by an instruction set architecture description (`AC_ISA`) and an architecture resource description (`AC_ARCH`), and generates a behavioral model of the architecture. `Acsim` uses two other tools that are not visible to the user: the ArchC Pre-Processor (`acpp`), which is composed by a lexical analyzer and a parser for the language, and a decoder generator. The parser extracts information from the description files and stores it into data structures that are used by `acsim` to create all C++ classes and/or SystemC modules necessary to build the architecture simulator. This generation flow is showed in Figure 2.6. The decoder generated by ArchC is capable of handling ISAs from simple RISC machines till multi-word of variable length instructions, like in many DSPs.

ArchC can generate simulators using the interpreted technique. Interpreted simulators execute instruction decoding, schedule and behavior dynamically. Since the decoding process is too costly in terms of simulation performance, these interpreted simulators may use a cache for decoded instructions in order to speed-up simulation. This technique can be disabled by command-line options passed to acsim, in order to enable the execution of self-modifying code.

2.3.1 Fast Static Compiled Simulation

The traditional approach to simulate an architecture is to mimic the hardware: fetch an instruction from memory, decode, and execute it. This is the interpreted simulation method. Unfortunately, the growing system complexity makes the traditional simulation approach inefficient for todays architectures and time-to-market pressure, makes performance one of the most important features of an instruction-set simulator.

The new approach is based on creating a simulator optimized for the application that will be executed, using an additional pre-processing step before the simulation. ASIPs and many DSPs applications that can be fully analyzed at design-time are the focus. Also, programs need to be run-time static, as all the programs in typical benchmarks suites like MediaBench[13] and MiBench[8]. During the pre-processing phase, the application is fully decoded and the instructions are instantiated statically.

The Fast Static Compiled Simulation (FSCS) need more information about the instruction set that is available from architecture description manual. Use information about jump instructions to produce a simulator that tests for control flow changes only after those instructions, meaning less work to be done at run-time. Optimizations techniques detailed could be found in [6].

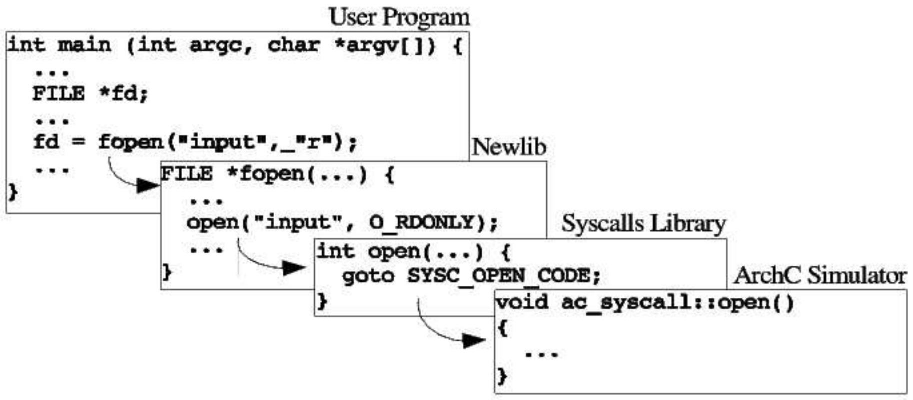
2.4 Emulating Operating System Calls

Programming in a high level language like C/C++ is mandatory for the the current embedded software development flow. The majority of applications are designed using C/C++ together with some hand-tuned assembly code to improve performance of critical inner-loops. Any non-trivial C/C++ program uses the standard C library (stdlib), which provides the most essential routines for I/O, dynamic memory allocation and other utilities.

ArchC generate simulators capable of being instrumented with an OS call emulation mechanism, which enables ArchC models to simulate applications containing I/O operations. The ArchC team hat grouped the system call routines in a retargetable library written in the C language, so only a recompilation is needed to port it to any other target. The designer of a new architecture is able to tell from which storage elements (memory, register bank, etc) the arguments to the system call will come from. It is done by writing interface functions that provide the required information to the ADL compiler. Typical examples of information that is required by most of the operating system calls are: how to get the first three arguments provided by a function call and how to save the return value as an integer or pointer.

Functions in this group are normally very small, with less then five lines. The information required to implement them is taken from the processor Application Binary Interface manual. The Figure 2.7 summarize how to control flow is transfered from user program running inside ArchC simulator to the native host OS routine.

Figure 2.7: Summary for calling the host OS



```
int main (int argc, char *argv[]) {
  ...
  FILE *fd;
  ...
  fd = fopen("input",_"r");
  ...
}
```

```
FILE *fopen(...) {
  ...
  open("input", O_RDONLY);
  ...
}
```

```
int open(...) {
  goto SYSC_OPEN_CODE;
}
```

```
void ac_syscall::open()
{
  ...
}
```

This feature allows ArchC to simulate programs extracted from real-world benchmarks, like MediaBench and MiBench, running them with realistic data samples.

Chapter 3

Infineon TriCore 1.3

The ArchC ADL has been used to describe diverse architectures but none with the characteristics of the Infineon TriCore Architecture. These chapter present these architecture, a 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. The TriCore Instruction Set Architecture (ISA) combines the real-time capability of a microcontroller, the computational power of a DSP, and the high performance/price features of a RISC load/store architecture. The architecture supports both 16-bit and 32-bit instruction formats. The 16-bit instructions are a subset of the 32-bit instructions, chosen because of their frequency of use.

In section 3.1 present a overview of architecture and congregates the information about the register and memory organization, addressing modes, task and context concepts. The section 3.2 describe briefly the instruction set and how the 16-bit instructions are implemented. These 2 sections contains reproductions of originals, copied of TriCore Datasheets[9, 10].

The 3.3 section is a survey of the existing tools development, that is important for the future use of the developed model. Finally the 3.4 section is overview of a IP-core based on TriCore Architecture, that can in future base the development of a TriCore cycle-accurate model.

3.1 Architecture Overview

The key features of the TriCore Instruction Set Architecture (ISA) are:

- 32-bit architecture. 4 GBytes of address space.
- 16-bit and 32-bit instructions for reduced code size.
- Most instructions executed in one cycle.
- Dedicated interface to application-specific co-processors to allow the addition of customised instructions.

- Zero overhead loop capabilities.
- Dual single-clock-cycle 16×16-bit multiply-accumulate unit (with optional saturation).
- Optional Floating-Point Unit (FPU) and Memory Management Unit (MMU).
- Extensive bit handling capabilities.
- Single Instruction Multiple Data (SIMD) packed data operations (2×16-bit or 4×8-bit operands).
- Byte and bit addressing.
- Little-endian byte ordering for data memory and CPU registers.
- Memory protection.

3.1.1 Architectural Registers

The architectural registers, showed in Figure 3.1 consist of:

- 32 General Purpose Registers (GPRs).
- Program Counter (PC).
- Two 32-bit registers containing status flags, previous execution information and protection information (PCXI - Previous Context Information register, and PSW - Program Status Word).

The PCXI, PSW and PC registers are crucial to the procedure for storing and restoring a tasks context.

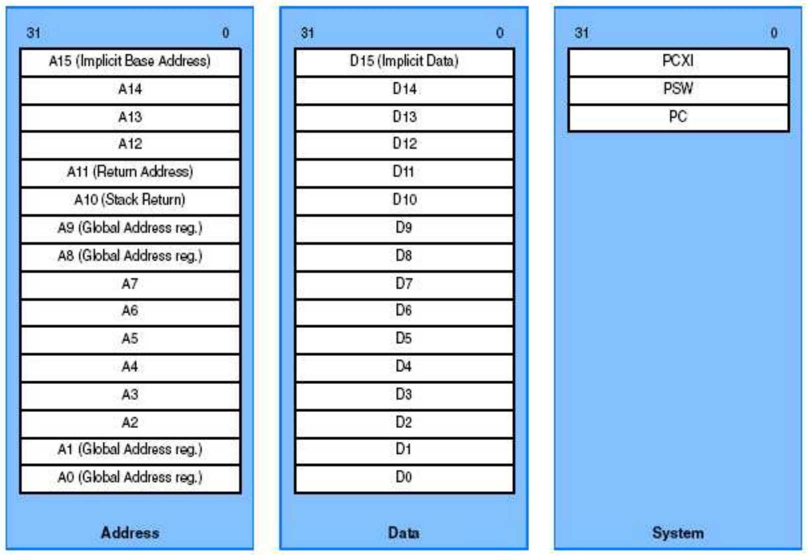
The 32 General Purpose Registers (GPRs) are divided into sixteen 32-bit data registers (D[0] through D[15]) and sixteen 32-bit address registers (A[0] through A[15]).

Four of the General Purpose Registers (GPRs) also have special functions:

- D[15] is used as an Implicit Data register.
- A[10] is the Stack Pointer (SP).
- A[11] is the Return Address (RA) register.
- A[15] is the Implicit Address register.

Support of 64-bit data values is provided with the use of odd/even register pairs. In the assembler syntax these register pairs are either referred to as a pair of 32-bit registers (for example, D[9]/D[8]) or as an extended 64-bit register. For example, E[8] is the concatenation of D[9] and D[8], where D[8] is the least significant word of E[8].

Figure 3.1: Architectural Registers



In order to support extended addressing modes, an even/odd address register pair holds the extended address reference as a pair of 32-bit address registers (A[8]/A[9] for example).

There are no separate floating-point registers. The data registers are used to perform floating-point operations. The floating-point data is saved and restored automatically using the fast context switch support.

Registers [0 - 7] are referred to as the lower registers and registers [8H - FH] are called the upper registers.

Registers A[0], A[1], A[8], and A[9] are defined as system global registers. These are not included in either the upper or lower context and are not saved and restored across calls or interrupts. They are normally used by the operating system to reduce system overhead.

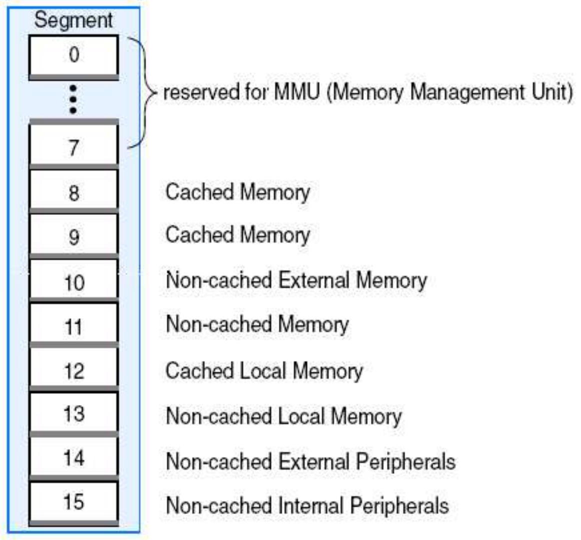
In addition to the General Purpose Registers (GPRs), the core registers are composed of a certain number of Core Special Function Registers (CSFRs).

3.1.2 Memory Model

The architecture has an address width of 32 bits and can access up to 4 GBytes of memory. The address space is divided into 16 regions,[0 - FH], showed in Figure 3.2, . Each segment is 256 MBytes. The upper 4 bits of an address select the specific segment. The first 16 KBytes of each segment can be accessed using either absolute addressing or absolute bit addressing.

Many data accesses use addresses computed by adding a displacement to

Figure 3.2: Address map and memory model


Segment
0
7
8
9
10
11
12
13
14
15
reserved for MMU (Memory Management Unit)
Cached Memory
Cached Memory
Non-cached External Memory
Non-cached Memory
Cached Local Memory
Non-cached Local Memory
Non-cached External Peripherals
Non-cached Internal Peripherals

the value of a base address register. Using a displacement to cross one of the segment boundaries is not allowed and if attempted causes a MEM trap. This restriction allows direct determination of the accessed segment from the base address.

3.1.3 Addressing modes

Addressing modes allow load and store instructions to efficiently access simple data elements within data structures such as records, randomly and sequentially accessed arrays, stacks and circular buffers. Simple data elements are 8-bits, 16-bits, 32-bits, or 64-bits.The TriCore 1 architecture supports seven addressing modes, they are:

- Absolute Addressing
- Base + Short-Offset Addressing
- Base + Long-Offset Addressing
- Pre-Increment and Pre-Decrement Addressing
- Post-Increment and Post-Decrement Addressing
- Circular Addressing
- Bit-Reverse Addressing

These addressing modes support efficient compilation of C/C++ programs, easy access to peripheral registers and efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for Fast Fourier Transformations). Addressing modes which are not directly supported in the hardware can be synthesized through short instruction sequences.

3.1.4 Tasks and Contexts

A task is an independent thread of control. There are two types: Software Managed Tasks (SMTs) and Interrupt Service Routines (ISRs). SMTs are created through the services of a real-time kernel or Operating System, and are dispatched under the control of scheduling software. ISRs are dispatched by hardware in response to an interrupt. An ISR is the code that is invoked directly by the processor on receipt of an interrupt. SMTs are sometimes referred to as user tasks, assuming that they execute in User Mode.

Each task is allocated its own mode, depending on the tasks function:

- User-0 Mode: Used for tasks that do not access peripheral devices. This mode cannot enable or disable interrupts.
- User-1 Mode: Used for tasks that access common, unprotected peripherals. Typically this would be a read or write access to serial port, a read access to timer, and most I/O status registers. Tasks in this mode may disable interrupts for a short period.
- Supervisor Mode: Permits read/write access to system registers and all peripheral devices. Tasks in this mode may disable interrupts.

Individual modes are enabled or disabled primarily through the I/O mode bits in the Processor Status Word (PSW).

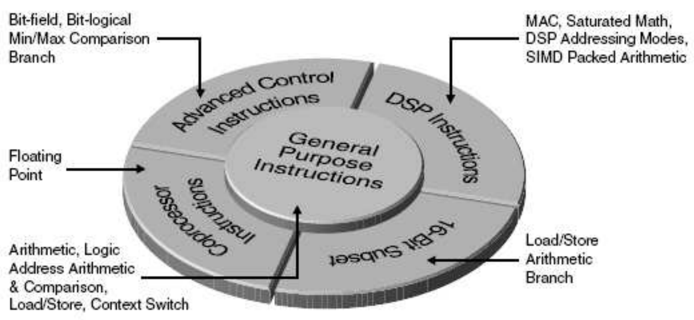
A set of state elements are associated with any task, and these are known collectively as the tasks context. The context is everything the processor needs to define the state of the associated task and enable its continued execution. This includes the CPU General Registers that the task uses, the tasks Program Counter (PC), and its Program Status Information (PCXI and PSW). The architecture efficiently manages and maintains the context of the task through hardware. The context is subdivided into the upper context and the lower context.

Context Save Areas

The architecture uses linked lists of fixed-size Context Save Areas (CSAs). A CSA consists of 16 words of memory storage, aligned on a 16-word boundary. Each CSA can hold exactly one upper or one lower context. CSAs are linked together through a Link Word.

The architecture saves and restores context more quickly than conventional microprocessors and microcontrollers. The unique memory subsystem design

Figure 3.3: TriCore Instructions Overview



with a wide data path allows the architecture to perform rapid data transfers between processor registers and on-chip memory.

Context switching occurs when an event or instruction causes a break in program execution. The CPU then needs to resolve this event before continuing with the program.

The events and instructions which cause a break in program execution are:

- Interrupt or service requests.
- Traps.
- Function calls.

3.2 Instruction Set

The instruction set of the TriCore is formed basically by 339 instruction mnemonics, divided in 5 types as showed in Figure 3.3.

The General Purpose Instructions is formed by arithmetic, logic, address arithmetic, comparison, load/store and context switch instructions. The Advanced Control Instructions is formed by bit-field, bit-logical, min/max comparison and branch instructions. The Coprocessor Instructions is formed by floating point instructions. The DSP Instructions is formed by MAC, saturated math, DSP addressing modes and SIMD packed arithmetic instructions. The 16-Bit Subset is formed by load/store, arithmetic and branch instructions.

The 16-Bit instructions are subset of the 32-bit instruction set, chosen because of their frequency of static use. The 16-bit instructions significantly reduce static code size and therefore provide reduction of the cost of the code memory and a higher effective instruction bandwidth. Because the 16-bit and 32-bit instructions all differ in primary opcode, the instruction sizes can be freely intermixed. The registers D[15] and A[15] are used as implicit registers in many 16-bit instructions.

Had to the 7 different types of addressing the number of instruction behaviors increases for 758. This number represents the total number of methods to be implemented in the ArchC processor model. Document [10] presents details about the implementation of each instruction.

The majority of the instructions executes in an machine cycle. The document [11] describes the timing of version v1.3 of the TriCore architecture, these information is necessary for the refinement of the functional model of the processor to a cycle-accurate model.

3.3 Compilers and Development Support

The $TricoreTricore^{\mathrm{TM}}$ architecture is well supported by a robust and comprehensive suite of development tools and services such as compiler-assembler tool chains, real-time operating systems, emulators, simulators, evaluation boards, training and consulting. In addition, SPACE Program[x], a partnership program of Infineon Technologies and third party vendors, the , places all development tool information for Infineon's embedded architectures at the developers fingertips.

For performance and trade-off analysis, and for developing and debugging a customized design, Infineon offer the TSIM, a configurable, instruction-accurate model of TriCore-1.3 core architecture that is integrated into all supported source-level debuggers. TSIM provides a simulation environment that models the TriCore core, memory configuration and interrupt mechanism.

To help designers to reduce DSP application development time and to understand the general purpose signal processing routines and their implementation on TriCore, is made available a DSP Library for TriCore (TriLib) containing more than 60 commonly used DSP routines.

The market pressure for productivity compels the development in languages as C/C++ and only little parts, performance sensitive, in assembly. Therefore the availability of compilers of these languages if becomes essential. Below 3 options for this architecture are related:

- **TASKING TriCore VX family C/C++ compiler package:** It is a proprietary solution of Altium Limited®, for MS-Windows® and Solaris® platforms. It brings C++/EC++/C compilers, Macro Assembler, Linker/Locator tools, a Debugger and a Development Environment.

- **Unicals (R) C/C++ Preprocessor:** That is a proprietary of Unicals(r), only for MS-Windows® platform. It brings a Hi-end meta-programming solution.

- **HighTec GNU TriCore C/C++ Compiler:** It is a commercial solution, for MS-Windows®, Solaris® and GNU/Linux platforms. It is based on a not public port of Gnu Compiler Collections[x]. It brings C Compiler, Assembler, Integrated Linker/Locator tools, a Debugger and various others tools to inspect and manipulate objects, libraries and executables.

Table 3.1: OS available for TriCore 1.3

OS	SUPPLIER	CODE-SIZE	DISTRIBUTION
MicroC/OS-II	Infineon	5-16kB	freely available source;use in product requires one-time licensing cost; no royalties
CMX-RTX	CMX Company	~5kB	source code: licensed per seat. no royalties
Nucleus Plus		25-40kB	source code: licensed per product. no royalties
OSE	OSE	application dependent~60kB	binary or source code: licensed per project; royalties or buy-out
PXROS	Hightec	application dependent	binary-only and source code available, licensed per-user, per-product or buy-out basis
VxWorks	WindRiver	60-250Kb	source code not provided; licensed per product; royalties
Linux	Infineon	application dependent?	Freely available source code, no royalties, no license

The fact, that these development tool-set run on GNU/Linux, as ArchC tools, and the availability of this tool in our environment of development, had guided the choice of this tool in the generation of the code of the applications in this work.

3.3.1 Compatibles Operating Systems

The existence of an operational system, that appears as an intermediate layer above of the hardware, offers to the designer some easiness when develop more complex applications, as for example, the creation and manipulation of threads.The table 3.1 shows some operating systems available for these architecture, the supplier, the code-size of OS in bytes and the distribution model of each one.

3.4 TriCore IP-core

The TriCore was ideally suited to SoC applications that require both microcontroller and DSP functionality together with high performance, low cost and minimal power consumption, Second Infineon Website, TriCore meets the needs of automotive, industrial, mass storage and communications applications where TriCore based ASSP silicon devices from Infineon are already successful. TC1MP-S is the synthesizable implementation of the TriCore architecture and is now available as a coreKit in the highly popular Synopsys DesignWare® library, an industry standard delivery package. The coreKit is a complete configurable subsystem (available in either VHDL or Verilog) with an industry standard

AMBA AHB interface, enabling simple integration into the wider platform, saving engineering time and effort to market.

- Latest implementation - 0.13 micron technology generation
- Clock Frequency - 300 MHz
- Performance - 450 sustainable MIPS
- Performance - 600 sustainable MMACs
- Core Area - 2.2 sq mm (including CPU, Code and Data Memory interfaces)
- Core Power Dissipation - 0.65 mW/MHz (running Dhrystone 2.1)

Design Views of the TriCore 1 coreKit are provided on request with the standard DesignWare® license, set to a default configuration. On completion of a License Agreement with Infineon, Synopsys can deliver the Implementation Views which include the synthesizable RTL in encrypted form. Under some circumstances unencrypted RTL may also be made available for licensing. Business terms consist of an up-front license fee and piece part royalties. Support and maintenance are included as part of the DesignWare® license between the customer and Synopsys.

Chapter 4

Model Development

In this chapter the main stages of TriCore processor model development are presented. The adopted solutions to solve the difficulties found in the way to generate a correct model are described.

Before initiating the creation of the processor model it was necessary to configure the development environment, which is composed in its majority for open-source tools, details on the environment configuration in section 4.1. As it was said in the previous chapter, the TriCore processor has many of instructions behaviors, due to the time limit we decides to initially implement a subgroup of these instructions. The criterion of choice and other implementation details are described in section 4.2. It is essential to assure that the model is correct to make possible the use of this model in future projects. With this objective two different approaches have been used that are described in section 4.3. Section 4.4 describes the creation of programs for the simulator from C code.

Finishing the chapter, two expansions in the model are presented in section 4.4.

4.1 Setting Project Environment

ArchC is an open-source SystemC-based ADL that is specialized for processor architecture description. SystemC is developed on the GNU/Linux environment, therefore we opted to use the distribution SUSE GNU/Linux 9.3, that has a rich tool-sets, broad standards support and is packed with tools that make it easy to develop applications.

During the development of the model we needed to compile the code and to execute simulation frequently, these activities were made remotely in an Intel(R) Pentium(R) 4 CPU 2.60GHz machine with 1GB of memory RAM.

It was used the versioning control tool CVS jointly with some verification Scripts, these tools automated the management of the code produced. Below we list the versions of the tools that were used.

- GCC 3.3.5

- SystemC 2.0.1
- ArchC 1.3, 1.5, 1.5.1, and initially an unreleased version that corrected a problem in the decoding of 16bit instructions.
- CVS 1.12.12

This environment kept stable during all development period. The fact that all the software used is open-source facilitates the redistribution of the developed model.

4.2 Describing Processor Model

The initial proposal describes the implementation of all the instructions of the TriCore processor, our initial analysis pointed a great number of instructions, but since they were simple instructions we believed the time would be enough. The plan was to implement about 60% of the instructions in the first month and the rest in the second month. At the end of the first month only 13% of the instructions were implemented and verified, due to the complexity in the addressing modes and also to the discovery of a BUG in the ArchC tool that did not correctly decode a code with 16bit.

Thus, from this moment on, it was determined that the instructions necessary to execute a set of 6 applications (described in chapter 5) were to be implemented. We initiate with the analysis of the Viterbi application assembly code that required the implementation of 80 new instructions behaviors. The necessary instructions for the other applications have been implemented gradually, and at the end 29% of the TriCore processor instruction-set were implemented.

The developed model is composed of the files:

- `tricore13.ac`: here we describe the information about the architecture, register banks, word size, endianess, etc.
- `tricore13_isa.ac:` here we provide information about the instructions, instruction formats, instruction opcodes and mnemonics.
- `tricore13-isa.cpp:` here we implement the instructions behaviors. Because of the great number of instructions, similar instructions have been grouped and divided in different files to facilitate the navigation in the code during the development.

Table 4.1 shows each one of these files, associating them with the instructions implemented. The field #TI means total instruction number of each file. The field II% indicates the percentage of implemented instructions of each file. The field #lines shows the number of lines of each file. At the end of the table the total values will be showed.

Table 4.1: Implemented files

File	**Description**	**#TI**	**II%**	**#lines**
tricore13-isa-abs.cpp	eq, eq.a, eq. b, eq.h, eq.w, eqany.b, eqany.h, eqz.a, ge, ge.u, ge.a, lt, lt.u, lt.a, lt.b, lt.bu, lt.h, lt.hu, lt.w, lt.wu, ne.a, ne, nez.a	12	0%	20
tricore13-isa-add.cpp	add, add.a, add.b, add.h, addc, addi, addih, addih.a, adds, adds.h, adds.hu, adds.u, addsc.a, addsc.at, addx	30	100%	238
tricore13-isa-and_or_xor.cpp	and, and.and.t, and.andn.t, and.nor.t, and.or.t, and.eq, and.ge, and.ge.u, and.lt, and.lt.u, and.ne, and.t, andn, andn.t, or, or.and.t, or.andn.t, or.nor.t, or.eq, or.ge, or.ge.u, or.lt, or.lt.u, or.ne, or.t, ornm orn.t, xor, xor.eq, xor.ge, xor.ge.u, xor.lt, xor.lt.u, xor.ne, xor.t, nand, nand.t, nor, nor.t, xnor, xnor.t	74	20,3%	204
tricore13-isa-cadd_csub_sel_cmov.cpp	cadd, caddn, csub, csubn, sel, seln, cmov, cmovn	16	37,5%	84
tricore13-isa-clo_cls_clz.cpp	clo, clo.h, cls, cls.h, clz, clz.h	6	0%	7
tricore13-isa-context.cpp	ldlcx, lducx, stlcx, stucx, svlcx, rslcx, bisr	12	8,3%	80
tricore13-isa-dv.cpp	dvadj, dvinit, dvinit.u, dvinit.b, dvinit.bu, dvinit.h, dvinit.hu, dvstep, dvstep.h	9	44,4%	118
tricore13-isa-eq_ge_lt_ne.cpp	eq, eq.a, eq.b, eq.h, eq.w, eqany.b, eqany.h, eqz.a, ge, ge.u, ge.a, lt, lt.u, lt.a, lt.b, lt.bu, lt.h, lt.hu, lt.w, lt.wu, ne.a, ne, nez.a	35	25,7%	141
tricore13-isa-extr_dextr_insert.cpp	extr, extr.u, dextr, insert, ins.t, insn.t	16	25%	55
tricore13-isa-j.cpp	j, ja, jeq, jeq.a, jge, jge.u, jgez, jgez, jgtz, ji, jl, jla, jlez, jli, jlt, jlt.u, jltz, jne, jne.a, jned, jnei, jnz, jnz.a, jnz.t, jz, jz.a, jz.t, call, calla, calli, ret, rfe, loop, loopu	57	68,4%	651
tricore13-isa-ld.cpp	ld.a, ld.b, ld.bu, ld.d, ld.da, ld.h, ld.hu, ld.q, ld.w, ldmst, lea, imask, swap.w	93	29%	308
tricore13-isa-madd.cpp	madd, madds, madd.h, madds.h, madd.q, madds.q, madd.u, madds.u, maddm.h, maddms.h, maddr.h, maddrs.h, maddr.q, maddrs.q, maddsu.h, maddsus.h, maddsum.h, maddsums.h, maddsur.h, maddsurs.h	88	1,1%	129
tricore13-isa-min_max_sat.cpp	max, max.u, max.b, max.bu, max.h, max.hu, min, min.u, min.b, min.bu, min.h, min.hu, sat.b, sat.bu, sat.h, sat.hu, ixmax, ixmax.u, ixmin, ixmin.u	28	10,7%	73
tricore13-isa-mov.cpp	mov, mov.a, mov.aa, mov.d, mov.u, movh, movh.a	15	100%	85
tricore13-isa-msub.cpp	msub, msubs, msub.h, msubs.h, msub.q, msubs.q, msub.u, msubs.u, msubad.h, msubads.h, msubadm.h, msubadms.h, msubadr.h, msubadrs.h, msubm.h, msubms.h, msubr.h, msubrs.h, msubr.q, msubrs.q	88	0%	120
tricore13-isa-mul.cpp	mul, muls, mul.h, mul.q, mul.u, muls.u, muls.u, mulm.h mulr.h, mulr.q	33	12,1%	86
tricore13-isa-mxcr_trapxv_rstv.cpp	mfcr, mtcr, trapv, trasv, rstv	5	40%	89
tricore13-isa-others.cpp	bmerge, bsplit, cachea.i, cachea.w, cachea.wi, debug, rfm, pack, unpack	23	15%	60
tricore13-isa-sh_sha_shas.cpp	sh, sh.eq, sh.ge, sh.ge.u, sh.h, sh.lt, sh.lt.u, sh.ne, sh.and.t, sh.andn.t, sh.nand.t, sh.nor.t, sh.or.t, sh.orn.t, sh.xnor.t, sh.xor.t, sha, sha.h, shas	32	15,6%	145
tricore13-isa-st.cpp	st.a, st.b, st.d, st.da, st.h, st.q, st.t, st.w	62	100%	411
tricore13-isa-sub.cpp	sub, sub.a, sub.b, sub.h, subc, subs, subs.u, subs.h, subs.hu, subx, rsub, rubs, rusbs.u	19	42,1%	82
tricore13-isa-system.cpp	syscall, dsync, isync, enable, disable, nop	6	50%	18
tricore13-isa-utils.cpp	auxiliary functions	-	-	450
tricore13.ac	architecture description	-	-	36
tricore13_isa.ac	instructions description	-	-	2844
tricore13-isa.cpp	instruction behavior implementation	-	-	287
	TOTAL	756	29,2%	6801

4.2.1 Compiling and Simulating the Model

As it was already presented in Figure 2.6, the generation process of the executable simulator model consists basically of:

1. Describe the model of the processor in ArhcC;
2. Execute the `acsim` tool that will generate the component's code in SystemC;
3. Compile the SystemC code that at the end will generate an executable model of the processor.

To execute an application, we execute the simulator model file and pass as command line parameter the path for the archive that contains the application to be executed. It is enough to compile it only one time to execute the various applications. The compilation process in a Intel Pentium 4 2.6GHz machine with 1GB of memory lasts about 1 minute.

At the simulation end 3 files with information on the simulator execution are generated, these files are:

- `tricore13.trace`: This file shows the change in the program counter (PC).
- `tricore13.dasm:` This file shows to the decoded instructions and its operators.
- `tricore13.stats:` This archive presents the amount of times that each instruction was executed, the amount of times that each memory device was used and the total number of executed instructions.

Information on simulation performance are presented in chapter 5.

4.3 Verifying

We planned the process of verification of implemented instructions on the following form. We describe the instructions in ArchC, after that would be created the test-vector, a program that tests the main functionalities of the implemented instruction. The lack of an assembler made that much time in the creation of each vector of test was expense in the creation of the test program. Because of the limited time we decide to change this verification method. It was continued the implementation of the instructions, without verification.

When we find a compiler for TriCore, we use then the following technique to verify the model. We executed an application in parallel in the simulator of instructions TSIM and the model that we develop, at the execution end the number of instructions executed in each model and the content of specific variable must have equal values.

Figure 4.1: Optimization information example

```
j__disp24.is_jump();
j__disp24.delay(1);
```

4.4 Running C Programs

In order to run more complex applications in the model, e.g. C code, we need a cross-compiler. It was used the Hightec GCC port to generate the code of applications. The output of this cross-compiler is a file in the ELF format. Initially, to execute this file with the ArchC model we had to change the code of the ArchC tool, currently this alteration is no longer necessary. A link-script was developed, in which file we inform the position of the program in the memory to the ELF file. The link-script file is called `tricore13_archc.ld` and has 208 lines of code.

4.5 Model Expansion

The model was expanded beyond the initial proposal in two ways. The first one with the objective of increasing the simulation performance using the Fast Static Compiled Simulation approach described previously in chapter 2. The second with the objective of increasing the observation capacity during simulation, making the model usable jointly with `gdb` debugger.

4.5.1 Compiled Simulation

In order to archive better performance for the simulator, we must identify the instructions that can change the execution flow. Using this information, the ArchC tool creates that saves execution time by testing for testing for control flow changes only after those instructions, 179 lines in the `tricore13_isa.ac` file had been added with this information. Figure 4.1 shows an example of the information that must be written.

The generation of the optimized simulator is a little different, now we need to use the tool `accsim` and to pass to it for command line already the application. With this the code of the specific simulator for this application will be generated. Using the Viterbi (bigger application) as example, the compilation has duration of about 43 seconds. The simulator executes 4774298 instructions in 2.27 seconds what it gives to a average of 2103.21 K intr/s.

4.5.2 GDB Support

Simulators built with ArchC can use the GDB protocol. We must only implement a few processor dependent methods for the interface and the simulator will be able to respond to GDB, allowing users to debug software inside the simulator. We implemented methods to interface the architecture and GDB, mapping registers and memory to the desired GDB format. These methods are:

- `int nRegs(void)`, `ac_word reg_read(int)` and `void reg_write(int, ac_word)`: so the simulator can send the read and write register packets to GDB.

- `ac_word mem_read(unsigned int)` and `void mem_write(unsigned int, byte)`: to inform how to read and write memory regions.

These methods are implemented in the `tricore13_gdb_funcs.cpp` file that has 180 lines.

Chapter 5

Results and Conclusions

This chapter provides the comparison results between the TriCore model developed in ArchC, an instruction-set simulator (ISS) TSIM[12] and the TC1MP-S core integrated on the FPGA system.

5.1 Benchmarks

Schmitt compares the execution time of an instruction-set simulator (ISS) TSIM[12] with the TC1MP-S core integrated on the FPGA system. The TSIM ISS can be found in software development tools from GNU or Tasking for TriCore in [4]. We used this data to compare the execution times of same the programs in the model developed in ArchC.

For performance analysis of the different platforms, benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC)[1] are used. The consortium was formed in 1997 to develop meaningful performance benchmarks for the hardware and software used in embedded systems. Through the combined efforts of its members, EEMBC® benchmarks have become an industry standard for evaluating the capabilities of embedded processors, compilers, and Java implementations according to objective, clearly defined, application-based criteria.

Since releasing its first certified benchmark scores in April 2000, EEMBC scores have effectively replaced the obsolete Dhrystone mips, especially in situations where real engineering value is important. EEMBC benchmarks reflect real-world applications and the demands that embedded systems encounter in these environments. Since the benchmark suite is not freely available Stephen Schmitt reimplemented six algorithms of EEMBC benchmarks, we use the same application code in TriCore ArchC model. Below are described the six used applications:

- **FFT:** Fast Fourier Transformation is a fundamental method from the signal processing domain. It transforms signals from the time domain into the frequency domain. There it works on discrete values. To get reasonable values for TSIM the FFT function works on 8 input values and transforms them 1000 times.

- **DCT:** Discrete Cosine Transformation is used as a compression technique in image processing. There, it can be found e.g. in JPEG and MPEG compression. We used a 8x8 matrix form transformation.

- **JPEG:** JPEG algorithm (Joint Pictures Experts Group) compresses pictures by removing frequency parts of an image not visible for a human eye. Quality of the pictures is configurable. For the benchmark a picture with eight pixels is used. The pixels are described with an 8x8 matrix.

- **LU:** LU is a triangular decomposition of a matrix A into the product of a left lower (L) matrix and a right upper (U) matrix such that $P*A = L*U$ holds. Matrix P is the permutation matrix with a one in each column and row. The measurement was done with a 3x3 matrix.

- **Packet flow:** Packet flow algorithm is a subset of the packet forwarding procedure of network routers. Potential performance of a microprocessor for an IP routing system can with this benchmark. For the measurement the link layer header war already removed from the packet.

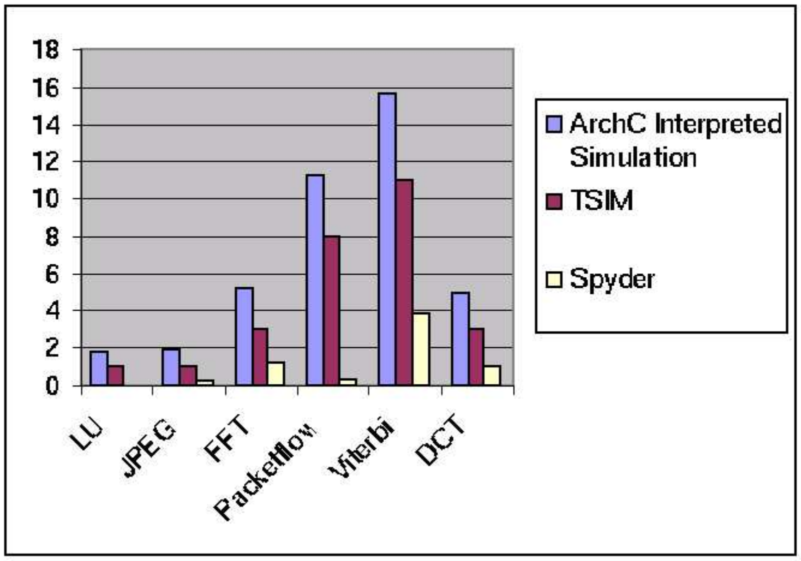
- **Viterbi:** Viterbi decoding is used in data transmission systems over channels with noise e.g. GSM or WLAN networks. Before transmission starts the data is encoded with a Reed Salomon Encoder from which the Viterbi algorithm can reconstruct the data with a maximum likelihood decoding. The benchmark uses 32 input values, simulates a transmission channel with noise and decodes data at the end of transmission. This procedure is done 100 times.

Schmitt made the runtime measurements for TSIM ISS on a Windows workstation with a Pentium 4 processor with 2.8 GHz and 1 GB memory. Figure 5.1 shows statistical values for benchmarks mentioned above.

5.2 Conclusion and future work

In this work a functional model of the processor Infineon TriCore 1.3 was developed using the architecture description language ArchC. Although just 29% of the instruction set was implemented, this number is enough to execute a specific applications subset. The model was validated comparing the outputs of the simulation with the instruction-set simulator TSIM. The link-script developed allows the model to execute ELF files, these are generated by a cross-compiler. The model was expanded beyond the initial proposal to increase the simulation performance using the Fast Static Compiled Simulation and increase the observation capacity during simulation making the model usable along with `gdb` debugger.

Figure 5.1: Simulation time



In order to improve the verification of processor model we can use the public benchmarks MediaBench and MiBench, which supply all test vectors. Prior to that, it is necessary to implement the Application Binary Interface (ABI) in the TriCore ArhcC model. This interface will allow the model to read the test vector and save the results in another one. The output can then be compared with the output vector of the public benchmark.

A natural evolution of the model, would be to implement the totality of its instructions, a relatively simple task. Another alternative would be to implement the cycle-accurate version with pipeline, based on the TC1MP-S IP core.

Finishing, the developed simulator showed to have satisfactory performance in comparison with ISS TSIM. Because the model is based on the SystemC library, it can be easily integrated in a SoC simulation platform.

Bibliography

[1] Eembc: Embedded microprocessor benchmark consortium, http://www.eembc.org.

[2] International technology roadmap for semiconductors. Tech. rep., 2001.

[3] *SystemC 2.0 Functional specification*, 2001.

[4] Speac architecture accurate prototyping on different levels of hw/sw abstraction. Tech. Rep. IT-WP5.2-Q2/05, August 2005.

[5] Allan, A., Edenfeld, D., Jr., W. H. J., Kahng, A. B., Rodgers, M., and Zorian, Y. 2001 technology roadmap for semiconductors. *IEEE Computer* (2002).

[6] Bartholomeu, M., Azevedo, R., Rigo, S., and Araujo, G. Optimizations for compiled simulation using instruction type information. *16th Symposium on Computer Architecture and High Performance Computing (SBAC'04)* (10 2004).

[7] Geppert, L. Eletronic design automation. *IEEE Spectrum 37*, 1 (January 2000).

[8] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization* (Dec. 2001).

[9] Infineon Technologies. *TriCore 1 Volume1: v1.3 Core Architecture*, July 2005.

[10] Infineon Technologies. *TriCore 1 Volume2: v1.3 Instruction Set*, July 2005.

[11] Infineon Technologies. *TriCore V1.3 Instruction Timing*, July 2005.

[12] Infineon Technologies AG. *Infineon Technologies AG: TriCore Instruction set simulator (TSIM)*, 2002.

[13] LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)* (Los Alamitos, Dec. 1–3 1997), IEEE Computer Society, pp. 330–337.

[14] PROPHET, G. System level design languages: to c or not to c? *EDN Europe* (October 1999).

[15] SANDRO RIGO, GUIDO ARAUJO, M. B., AND AZEVEDO, R. Archc: A systemc-based architecture description language. In *to appear in the 16th Symposium on Computer Architecture and High Performance Computing - Foz do Iguacu, Brazil* (October 2004).