UC Irvine
ICS Technical Reports

Title
Architecture and compiler for an ANSI C-targeting reduced instruction set core for embedded systems (ANTARES)

Permalink
https://escholarship.org/uc/item/824974nd

Author
Jansen, Dirk

Publication Date
2000

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

ICS

TECHNICAL REPORT

Architecture and Compiler for an ANSI C – targeting Reduced Instruction Set Core for Embedded Systems (ANTARES)

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

Dirk Jansen

Technical Report ICS-00-26

Department of Information and Computer Science
University of California, Irvine
Irvine, Ca 92697-3425, USA

July 2000

Information and Computer Science

University of California, Irvine

Architecture and Compiler For an ANSI C – targeting Reduced Instruction Set Core For Embedded Systems

Dirk Jansen

Technical Report ICS-00-26

Department of Information and Computer Science

University of California, Irvine

Irvine, Ca 92697-3425, USA

001-949-824-8919

d.jansen@ics.uci.edu

Abstract:

In this document the architecture and instruction set of a RISC-Core is designed in a top down fashion so ANSI-C-programs are easily processed on the core giving optimal performance with a minimum of architectural features. Severe constraints have to be obeyed concerning register size, instruction-coding and addressing, limitation resulting from the goal of a small area of the core and easy memory interfacing. Compiler design and architecture engineering is done concurrently, influencing each other and showing strong interdependence of instruction set, architecture and compiler. The LCC- retargetable compiler is used as a basis and the machine description file is elaborated. A retargetable assembler is taken from an existing design with a new instruction description file. The core is simulated with an existing, retargetable C- based simulator.

Contents

CONTENTS 3

1 INTRODUCTION 5

2 ARCHITECTURE CONCEPT 6

2.1 ADDRESS-SPACE 6
2.2 REGISTER NUMBER AND SIZE 6
2.3 INSTRUCTION FORMAT 7
2.4 LOAD – STORE ARCHITECTURE 8
2.5 INTERMEDIATE STORAGE, STACK-CONCEPT 8
2.6 ADDRESSING MODES 9
2.7 ALU DESIGN 10
2.8 CONTROL STATE MACHINE DESIGN 12

3 MEMORY LAY-OUT 13

3.1 HARD CODED SOC 13
3.2 SOC WITH DOWNLOAD (SOFT CODED SOC) 13
3.3 CORE WITH EXTERNAL RAM AND EXTERNAL ROM 13
3.4 MAIN APPLICATION GOAL OF ANTARES 14
3.4 POSITION OF BIOS ROM IN THE MEMORY MAP 14
3.4 MAPPING OF MEMORY SECTORS TO THE RAM 14

4 INSTRUCTION SET DESIGN 16

4.1 INSTRUCTION CODING 16
FORMAT A: INSTRUCTIONS WITHOUT EXTENSIONS 16
FORMAT B: INSTRUCTIONS WITH 8-BIT VECTOR ON BUS 16
FORMAT C: INSTRUCTIONS WORKING ON ONE REGISTER 16
FORMAT D: INSTRUCTIONS WITH TWO REGISTERS ADDRESSED 16
FORMAT F: INSTRUCTIONS WITH LONG OFFSET 17
4.2 INTERDEPENDENCE BETWEEN COMPILER AND INSTRUCTION SET 17

5 ADAPTING THE LCC-COMPILER 20

5.1 RULES DESCRIPTION 22
5.2 C-CODE SECTION 27
5.3 ANTARES INTERFACE BINDING 28

6 VERIFICATION AND PERFORMANCE 29

6.1 WORKING WITH GLOBAL VARIABLES 29
6.2 USING LOCAL VARIABLES 31
6.3 USING ARRAYS AND LOOPS 32
6.4 COMPILING FUNCTION CALLS 32
6.5 PERFORMANCE OF LONG AND FLOAT PROCESSING 32
6.6 COMBINED TYPES AND OPERATIONS 33
6.7 OVERALL PERFORMANCE ASSESSMENT 33

7 ASSEMBLER 33

8 SIMULATOR 35

9 CONCLUSION 36

10 REFERENCES 37

1 Introduction

Application specific integrated circuits (ASIC) are getting more and more complex. The requirements of customers to higher intelligence of the chips and at the same time further pressure on the costs leads to Asics with all electronic functions integrated on only one chip, so called „systems on chip“ (SOC). These ICs now contain a programmable microprocessor core together with memory, interface and communication devices as well as analog interfaces to sensors and actuators. They are embedded systems in a similar way as they were integrated before on an electronic circuit board. High integration, flexibility of programming and small silicon area makes these SOC attractive for mass applications like chip cards, distributed sensors, identification devices etc., allowing new products and services with a potential of multimillion dollar markets.

In the design of systems on chip (SOC), the processing core is of key importance. There are several existing cores on the market, compatible with their discrete counterparts like the 8051, the 6805 or ARM 7TDM. These cores can be integrated as a hard-macro or a soft- macro, together with special designed interface and application electronics.

Cores can be general categorized related to their word size as 8-bit, 16-bit or 32-bit processors and to their architecture as RISC or CISC – style. A big bulk of software support exists for these cores with assemblers, debuggers, and compilers. This support is taken from the discrete component developments and adopted to the integrated environment.

So *why should we do the effort to design a new processor core* in competition to the existing ones and set up all the software and hardware tools for effective development with this core?

First of all and already mentioned:

- The **processor core is the key component** and we should know all of it and its behavior.
- The **cores from the market are expensive** and require license fees in the order of several hundred thousand dollars for high performance cores like the ARM 7. Only big companies can afford that.
- **One core is not enough**. For different applications different cores with more or less performance are needed, 8-bit for small applications, 32-bit may be for high performance tasks. We need a family of scalable cores with similar interfaces and bus structures.
- **Times have changed**. The core should be described today as a soft-macro in a high level language like VHDL and be synthesizable to different CMOS technologies. Routing should be possible in a flat manner and with predefined shape to optimize for small chip area.

There are lots more arguments for an own design, if you have the capability in doing so.

There are several academic designs in the literature [FHOP], but the circuit design is only a small part of the general task. The effort for setting up the development tools is much bigger and with more and more software integrated in the chip it is no longer reasonable to stay on the assembler level. The step to high level language programming, which is C today, has to be taken. If you decide to use a C- compiler, it has to be checked, if your *core fits to high level language programming*, which is mostly not the case. This is one of the reasons why more than 50% of all embedded programming is still done in assembler [LEU99]. This is also true for many of the commercial legacy cores.

The classic way in designing a processor is

- to set up the architecture first,
- then define the instruction set
- and much later consider the C-compiler,

which now needs complicated constructs to fit the architecture and instruction set.

Reason for this bottom up design stile has been the big effort for the circuit design, which has been considered as critical and changes were difficult to implement. Nowadays with VHDL synthesis the effort for architecture definition is less and the possibility to include changes and modify structures is largely improved. It is possible now, to design all parts of the architecture and the related development environment concurrently. Performance is optimized in all parts at the same time.

With this in mind the task of designing a family of integrated cores, optimized for C- programming has to be reconsidered. This reports shows in detail and in a tutorial manner, how design decisions are done. The

ANTARES – core is one result of this effort, it may not be the "optimal C-core", if that exists at all, but it is well suited to the goal: to **get an efficient, easy to program and small core for SOC-applications**.

2 Architecture Concept

Discussing the architecture, we first have to fix, that there is no Pentium© or Power-PC© to be invented again. The application area of the intended core is the SOC, typical with on chip RAM of several Kbytes and ROM below 8 Kbytes, maybe with some download capability as later described.

The task of the core is mostly control and communication with some rare exceptions of mathematical calculation. The SOC is very cost sensitive, meaning there is a hard limit in chip size to some 20 ...30 mm² which implicates a core size for the processor of maximum 2...3 mm² and on chip RAM of maximum 8 Kbytes, dependent on the process (0.5u assumed).

The architecture shall be designed for easy adaptability of programming constructs, used in the C-programming language, preventing that there are constraints to the compiler, lowering compilation efficiency. The core should use available memory as efficient as possible. There should be provision for interrupt handling and a concept for an operating system.

With ANTARES, the time needed to setup the software for a typical SOC application should be significantly reduced by using the C-language without paying the penalty of large memory usage and slow performance.

2.1 Address-Space

Memory is the most expensive part, so the code must be as compact as possible. With this in mind, we choose a 16-bit address space, allowing addressing memory of 64 Kbytes.

For bigger applications with additional dynamic RAM outside the chip there would be a larger address-space of 32 bit advised. This is the big brother in the family, you can do it, but it needs resources.

An address-space below 64 Kbytes makes no sense nowadays, so 16-bit address is the minimum.

2.2 Register Number and Size

Registers store information and handle calculations. How many do we need? Registers have to be addressed or selected. With 2 bits select code you can address 4 registers, with 3 bit 8 registers, with 4 bit 16 registers etc. A typical register to register instruction combines 2 registers with an operator and writes the result in a third register:

R3 <= R1 (op) R2

With an n-bit select address we need 3 x n bits for register selection in the instruction code word. In a compact code of 16 bit only, it is not possible to spend 9 bits on the selection addresses.

There are ways to simplify the situation:

Using one select address twice:

R2 <= R1 (op) R2

Constraints implied are not very severe. Two registers are selected in the instruction, resulting in a select address of 2*n. The content of register R2 is destroyed with this concept. Many expressions can be arranged in that way, that one of the two variables is only used once, so the overwriting has no effect. This way of register selection can be found by many existing architectures, fi the x86 series [INT86].

Registers can be arranged in

- the on chip memory, which is very space efficient (see 8051, [INT51]),
- or in a register file made out of flip-flops.

Acquisition time to flip-flops is below 1 nsec today and a factor of minimum 5 faster than access to memory. So the memory version is much slower. It requires a two port memory, better a three port memory, which is not available in standard libraries. But flip-flops are large in size and there are additional tristates for selection and gating, so the number is most critical for the size of the core. Indeed the register file takes about half the size of the overall core in a comparable design [FHOP].

A further advantage of the flip-flop version is the lower power consumption in comparison to the RAM version. For synthesis, flip-flops are easy to include, building a homogenous architecture. Memory based register files are hampered by the problems with the memory-macros, which are different from technology

to technology. So decision is made to take the flip-flop version.

How many registers do we need? There should be a stack register S, an index register X and some registers for intermediate storage and operation. 16 register would be nice but much to large, so 8 registers is chosen using a selection address of 3 bit each.

Register size is next to decide. One register should take the content of one variable. In C, there are variables of type *char* (8 bit), *int* (16 bit), *long* (32 bit), *float* (32 bit), *double* (64-bit) etc. so it depends on the application, which is the optimal size. Control applications with only marginal calculations, handling measurement values of below 16-bit accuracy, are well served with 16-bit word size. Data types "*long*" and "*float*" are rarely used so using more complicated processing for them does not impede the performance much.

Is there is a lot of mathematical calculation required with high accuracy involved, 32 bit word size is unavoidable, this is the big-brother-design mentioned before.

So there are several possible register organizations, three are shown in Fig 2-1 to Fig. 2-3.

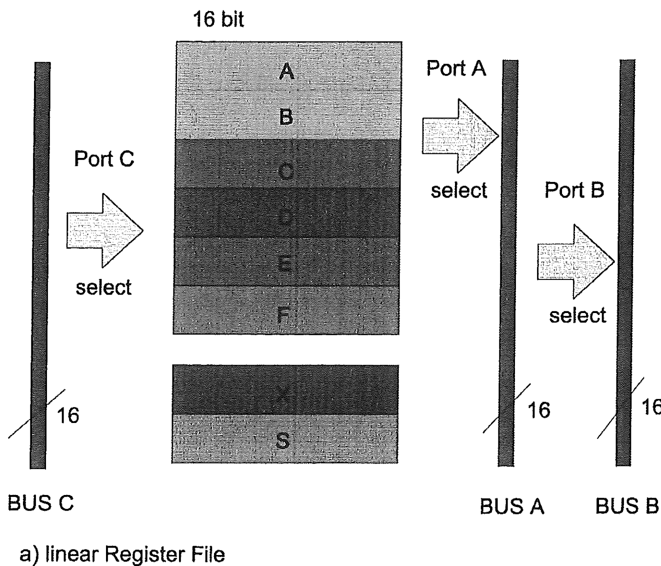

Figure 2-0-1: Linear register set, one input, two outputs.

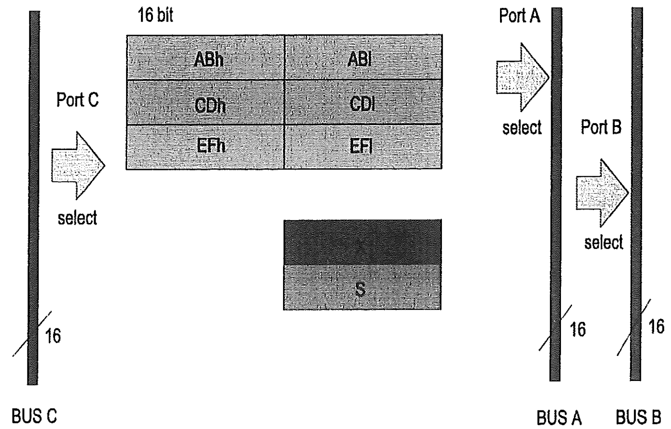

Figure 2-0-2: Register File with three separable double registers of 32 bit, hi-word and low- word separate accessible (selected version for ANTARES)

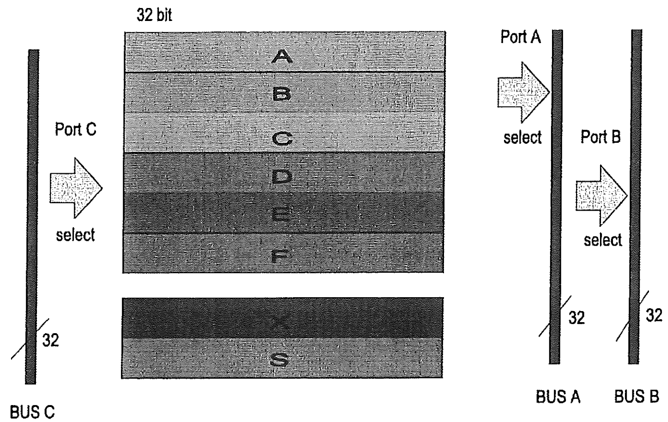

Figure 2-0-3: Register file of 32-bit registers each.

For the intended application area we decide to take a word size of 16-bit, which is compatible with the addressing word size (pointer size) of 16-bit. The version of Figure 2-2 is selected, in preparation of later 32-bit processing of long- and float- variables. With this the register file contains 8 x 16 flip-flops or 128 bits.

2.3 Instruction format

With register size 16-bit all the busses are 16-bit in width and it is reasonable to choice 16-bit too for the size of the instruction word. This is a decision with many implications, which shall be outlined further.

An uniformed format for addresses, variables and instructions allows an efficient use of memory in the classical John von Neumann architecture concept. A 16- bit instruction format together with a 16-bit data bus allows accessing one instruction with every bus cycle. Pipeline control is easy to implement with an intended one-instruction-per-clock-cycle performance.

But there is a severe limitation to this concept: the 16-bit word size cannot transport instruction code and immediate constants of 16-bit. Or more severe, it cannot include address pointers of 16-bit at the same time.

There are two general solutions to this problem:

1. **Extension of the Instruction with a second 16-bit data –format**. The instruction format does not contain any constant or address information. This information is packed into the extension, so that these instructions in fact are 32-bit instructions.
2. **Packing only a part of the constant (8-bit, 12 bit) into the instruction itself** and using a prefix-instruction mechanism, which transports the additional information. In fact, these instructions are combined of two separate instructions. The information of the prefix instruction is stored in an internal prefix-register (8-bit) and used together with the content of the immediate field in the instruction format. The prefix instruction can be combined with all related instructions. It is included by the compiler/assembler if needed.

Looking to typical assembler codes and trying to optimize in that way, you can avoid in many cases to use large address pointers. Immediate constants are mostly small and below 8-bit too. With this in mind it is not effective and blows up the code, to use version 1 with code extension, so we decide to use the prefix mechanism (version 2) here, although it complicates the compiler/assembler. How much prefix is needed, depends largely on the stile and memory layout of the programmer. It may be below 10% or even less. The instruction processing is kept to one instruction/cycle. The prefix mechanism is more described in detail in the next chapter.

2.4 Load – Store Architecture

There are several ways to process data in the data path. Here we decide to use the RISC – typical load – store architecture, meaning Figure 2-0-4:.

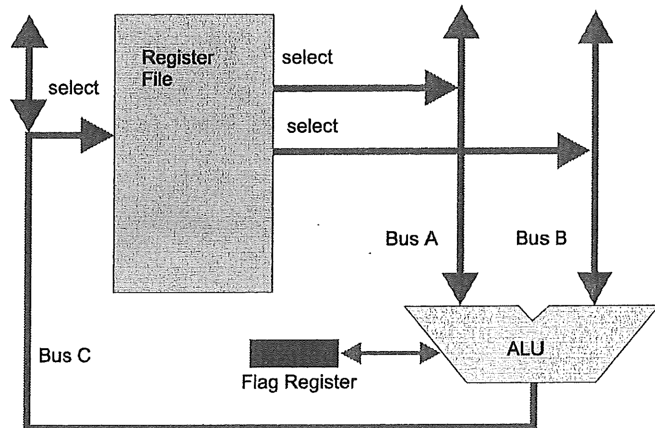

Figure 2-4: Data path general concept. Two registers are combined to one result, which is stored again in a register

- there are dedicated instructions for loading data to registers and storing data to memory.
- Combining registers via an ALU and storing results again in registers does processing of data, see figure 2-4.

2.5 Intermediate Storage, Stack-Concept

For intermediate storage of data and addresses the well-proven stack-concept is chosen. This harmonizes with the C-compiler later described, which supports stack architecture. For stack management, a dedicated register S as a pointer register is used. S is member of the register set and can be used, read, written or manipulated like the other data registers. There are hardware programmed instructions like PSH and POP to move data on the stack or get it from, decrementing or incrementing the stack register S by 2 (all addresses calculated in bytes). Subroutine CAL – instruction puts actual address on stack, RET from subroutine uses this address automatic.

Stack is further used for arguments of functions, which are pushed on the stack before CAL and for allocation of local memory, which is done by decrementing the stack pointer so many times as required cells for storage are needed.

Access to local memory is done by stack pointer relative addressing, with the offset contained in the immediate field of the instruction. With good programming stile, most of the locals can be accessed without using prefix mechanism, leading to very efficient use of memory without any managing overhead.

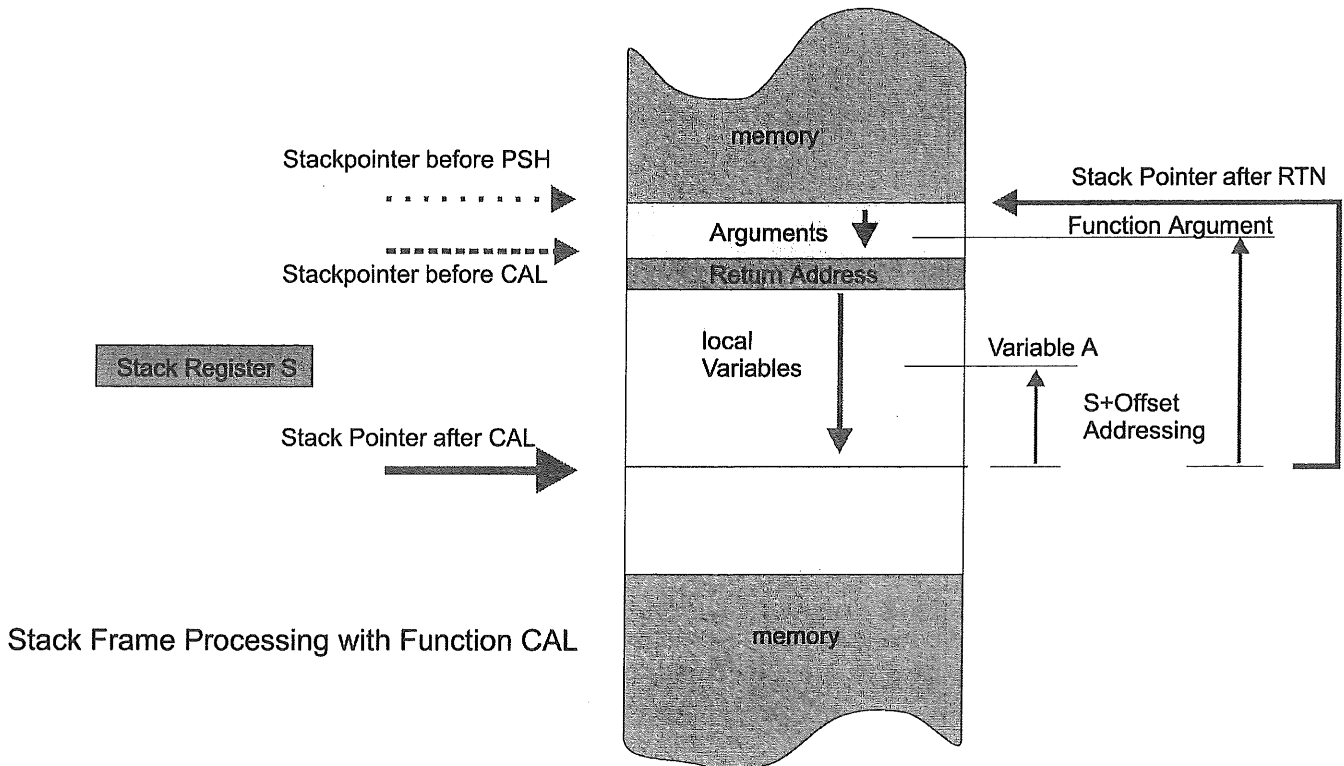

Figure 2-5: Stack usage for arguments, local memory and return address.

2.6 Addressing Modes

There are several addressing modes implemented.

Absolute addressing (immediate)

There are load and store instructions, which address memory absolute. The immediate field contains the direct position in memory. With 8 bit, the first 256 words (512 bytes) of the boot sector can be accessed directly, with prefix mechanism the whole 16 bit memory space. The compiler uses absolute addressing for global variables only. The immediate field is interpreted as word-address, getting the byte-address by shifting left 1, see figure 2-6.

Indexed addressing

There is a dedicated index register X in the register file, which is used for pointing to addresses. Loading this register with an address, every other register can be loaded indexed. X has 16-bit width, so each cell in the 16-bit memory space can be accessed, see figure 2-7.

There is no offset possible and needed in the index instructions. All pointer manipulation is done with normal calculations. The Index register is widely used from the compiler.

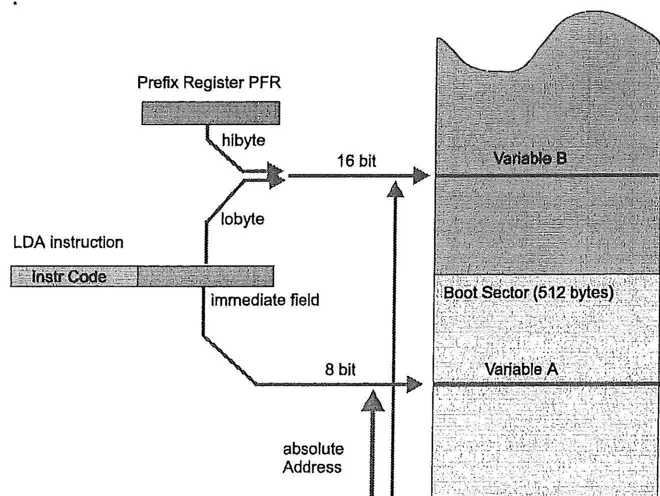

Figure 2-6: Absolute addressing mode.

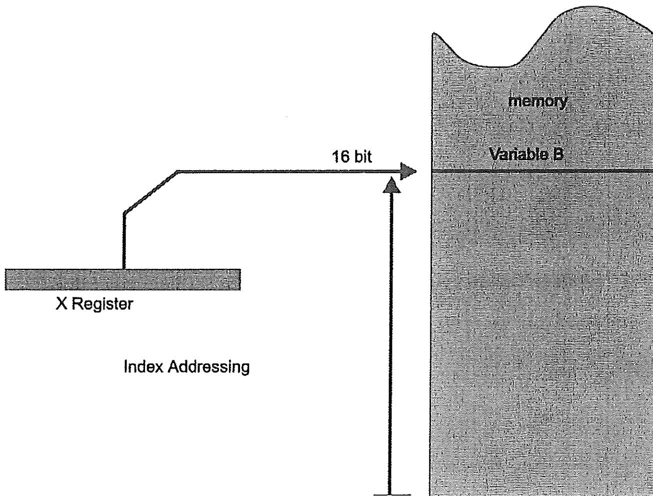

Figure 2-0-5: Index addressing mode

Stack-relative addressing

Memory can be accessed using the stack register S as an index register. The instructions contain an immediate field allowing adding an offset to S without modifying S. Stack-relative addressing instructions are broadly used by the compiler to access local variables, allocated on the stack. The offset is limited to 8-bit, requiring a prefix mechanism if larger offsets are needed, which may be the case for large arrays or many locals. In most cases the 8-bit address space will be sufficient. See figure 2-8 for details.

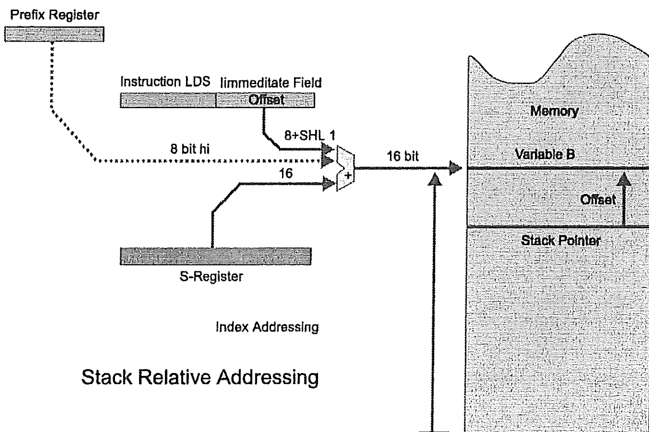

Figure 2-0-6: Stack relative addressing mode

Program counter relative addressing

This addressing mode is implemented although the compiler does not use it for variable access directly. Addresses are calculated using the program counter and an 8-bit offset in the immediate field of the instruction. PC relative addressing is only used for subroutine CAL and relative jump JPR instructions, taking advantage of the locality of the code. There may be also some usage for local constants, located to the functions, but this is not supported by the compiler and not implemented yet. See figure 2-9.

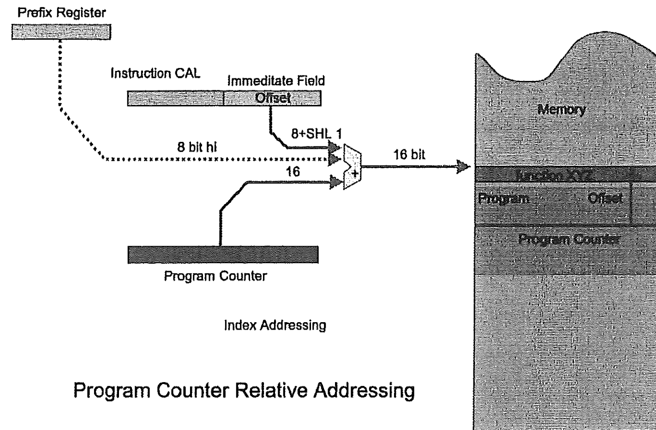

Figure 2-0-7: Program counter relative addressing mode.

Vector Addressing

Every start address in memory can be reached via the software interrupt mechanism SWI, described later in detail. The interrupt vector is placed in a table, residing in low memory.

2.7 ALU design

The arithmetic logic unit combines two busses of 16-bit each to a result of 16-bit plus flag output, which is stored in the flag register. So integer variables can be processed as well as characters. Long and float variables need 2 registers and a minimum of two instructions to process. This fits to the overall 16-bit architecture design. Of course this is not optimal for floating point processing and mathematical calculations, but this has been discussed before. If you need that, a consequent 32-bit design with 32-bit busses and 32-bit memory access has to be elaborated. Such a core would be, roughly estimated, 4 times larger the one we are designing here. The implication on the instruction set is not very large, we will give later a short idea on the design of such a core, but we will not elaborate the design in detail.

The ALU should be able to perform the functions from table 2-1.

Table 2-1: Main functions of the ALU

arithmetic	*logic*	*unary*
$C = A + B$	C= A & B	C = 0
$C = A - B$	$C = A \mid B$	C = A
$C = A * B$	C= A (+) B	C = B
$C = A + 1$	C = A<< 1	C = -A
$C = A - 1$	C = A >>1	C = !A

There are several details and additional functionality that are discussed with the instruction codes.

There is no division and no barrel shifter included because of the effort for these devices. Both functions are provided by software, which of course slow down performance. Division is rarely used in data processing and can be substituted by multiplication in many cases. Shifting is needed in many ways, but mostly with a constant, so the general shifting function is again done by a short subroutine.

Multiplication is directly included in the ALU. The C language defines the result of the multiplication of integer numbers as integer. Taking this, we do not need a full 16 x 16 bit multiplier with 32 bit result. Only the lower 16 bits are used. So there is roughly half the effort in hardware. The already existing adder can be used for adding the partial products. Propagation time of such a device is still twice the time or more than for an adder alone, so there should be a pipeline register stage between multiplier and adder. The critical path delay from register to ALU and back to register again may be in the range of 3…5 nsec. This should allow a clock frequency of more than 200 MHz for the core, if needed in the application.

Multiplication as well as some other instructions like CAL, RET and PSH will need more than 1 clock cycle, which complicates the control state machine, but this has to be accepted.

The ALU design is straightforward. For low power consumption only that part switched to active, which is needed in the actual instruction. The type of the adder may be carry-look-ahead or faster, depending on the balance of the delay times. With 16–bit and up to date

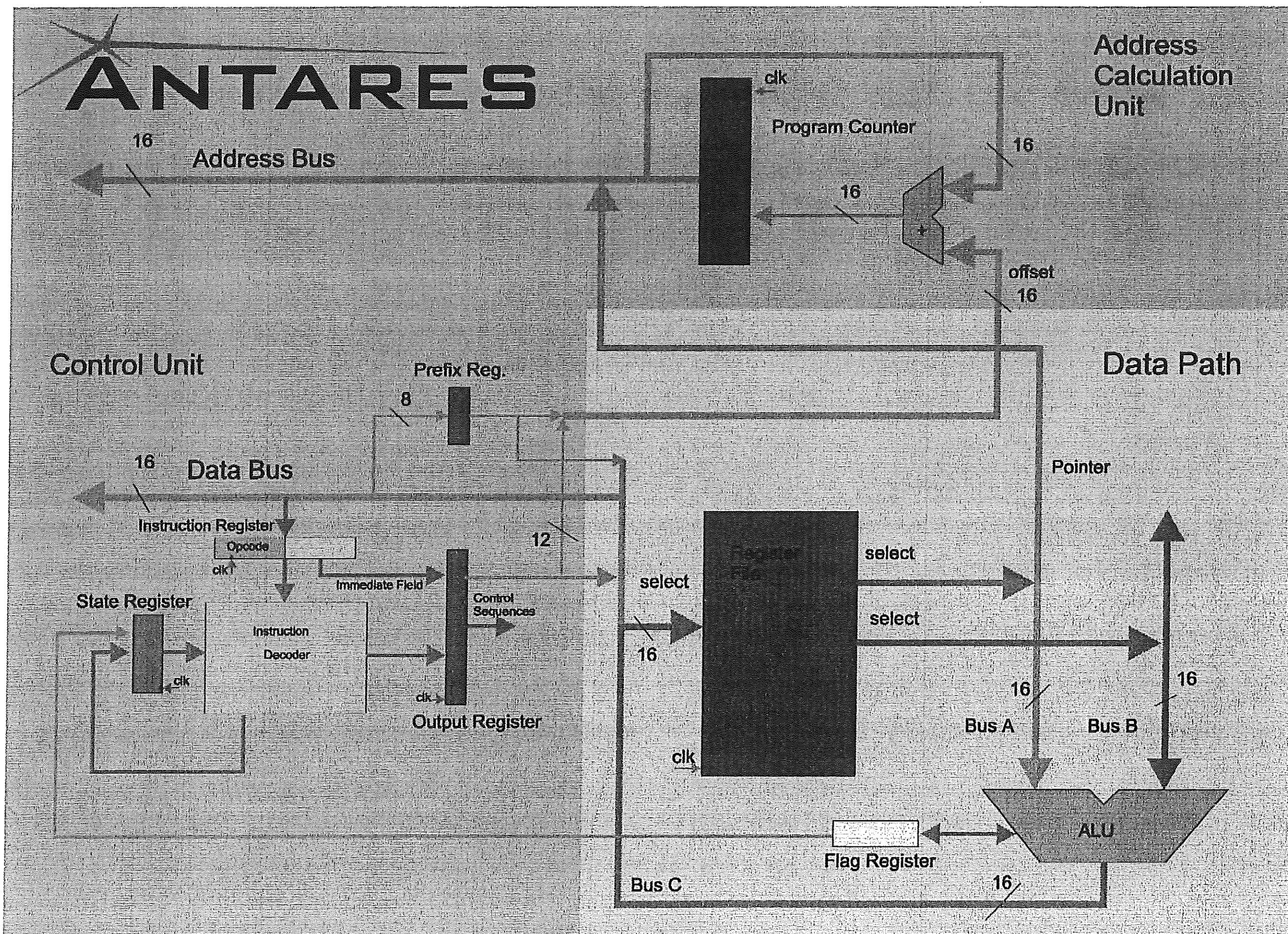

Figure 2-11: Antares architecture design, control-unit, data path and address unit as main building blocks.

CMOS technologies, this is standard design work. Some work has to be done to formulate the VHDL-codes in that way, that there is an efficient synthesis for these components, using mostly a structural description style.

2.8 Control state machine design

With a simple clock scheme of 4 states, there will be all (but multiplication) register-to-register-instructions evaluated:

1 Instruction fetch

2 Instruction decode

3 Execute A

4 Execute B

There are two states for instruction execution, allowing additional address calculations and access to the program counter. These 4 states can be easily pipelined with an average of 1 clock cycle/inst. For load /store instructions, a minimum of 6 states are required. This results from the chosen "von Neumann architecture" with only one data-bus interface. On average, these kinds of instructions need 2 clock cycles with pipelining, details have to be elaborated later in the design.

We have considered a cache memory with separate data and instruction space too, allowing a parallel access to data and instructions. This complicates memory design enormously and is not effective for small SOCs. Another solution may be a two port memory. Again this puts further constraints to the universality of the core, which is not intended. So we have to live with the fact, that memory access to variables needs double the time of a register to register instruction.

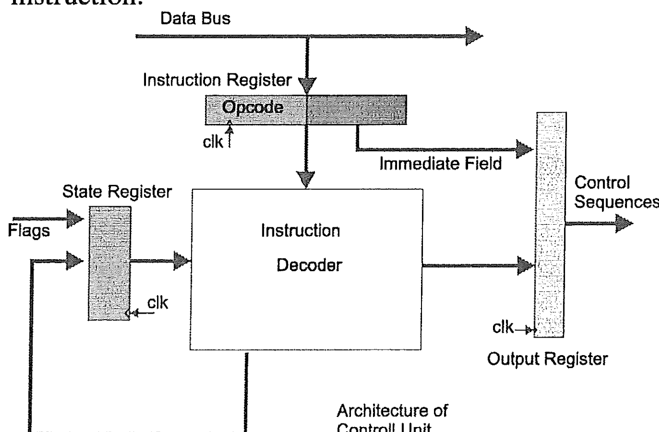

Figure 2-0-8: Control unit design concept

Instruction decoding will be organized as effective as possible, using the systematic structure of the instruction code. A pipeline register at the output of the decoding will synchronize to the clock, shorting the critical path and providing clear and hazard free control signals.

Interrupt is implemented in the control structure in two ways:

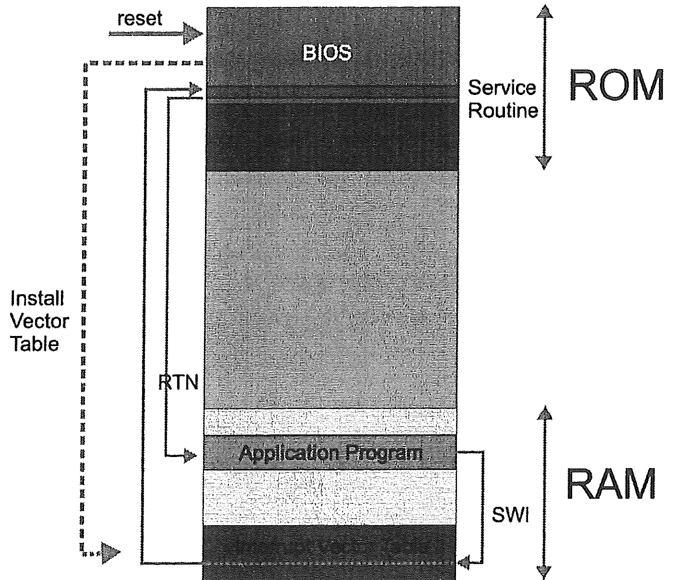

Figure 2-12: Interrupt mechanism. The interrupt table is installed with initialization by the BIOS. If interrupt (software or hardware) occurs the address of the related service routine is found in the vector table. The service routine may be typical a BIOS routine.

Hardware interrupt via 1 sensitive interrupt line, which is latched by a flip-flop, member of the flag register, controlled by an interrupt enable flip-flop. Interrupt priority and managing is left to an interrupt controller, which may be added as a peripheral unit. Interrupt is organized similar to a CAL instruction, but with the interrupt vector-index, an 8-bit number, on the bus. These index points to an address in the boot sector, the lowest part of the RAM. At this address the interrupt vector is found, which is the address of the service routine. The vector table has to be loaded during initialization of the program. The index has to be provided by the interrupt controller via the bus. A maximum of 256 vectors is possible with this scheme, but rarely needed.

The interrupt mechanism is also used for so-called **software interrupt** SWI, with the index provided in the immediate field of the instruction. Software and hardware interrupts share the same vector table. The compiler linking the BIOS routines to the program uses software interrupts too. So vectors to the BIOS routines occupy the first 32 vectors. See figure 2-12 for a memory map. More on this is detailed in the next chapter.

There is further a HLT instruction, putting the controller in a low power sleep state with no memory access and all busses high impedance. Wake up is only possible with reset or a hardware interrupt. This mechanism is also included in hardware.

Figure 2-11 shows all decisions and a first scope of the architecture.

3 Memory Lay-Out

Memory is very costly in a SOC so there has to be a careful layout of memory used. Generally there are two main different kinds of memory: read only memory (ROM) and random access memory (RAM). Memory is not part of the processor core, which consists only out of gates and flip-flops, but has to be considered as an important part of the system, influencing the core design.

We will first list the most important SOC configurations:

3.1 Hard coded SOC

There is only one chip, containing all software in the ROM with only a small RAM for variables and intermediate storage. This is not a flexible but often used configuration for mass production. Some more flexibility is there if the ROM is a FLASH or EEPROM device, but this needs special CMOS processes and is a relatively expensive approach, only used for development or small production numbers. See here the experience with the 8051 [INT51] family or the PIC-family of products [PIC].

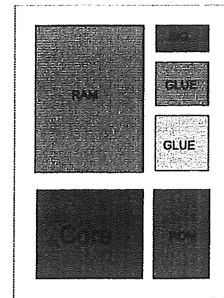

Figure 3-0-1: Basic hard coded System on Chip (SOC)

3.2 SOC with download (soft coded SOC)

Part of the software is placed in the ROM, part of the software is downloaded in an initialization or configuration process. The ROM contains general routines, mostly called BIOS, the application program is downloaded via a serial interface. Downloading can be done actively or by a routine situated in the BIOS during initialization. Program storage is external in an EEPROM or a FLASH - memory. Execution of a program is done mostly in RAM, which is now larger in size. This is a very flexible solution with nearly no limitations in program size.

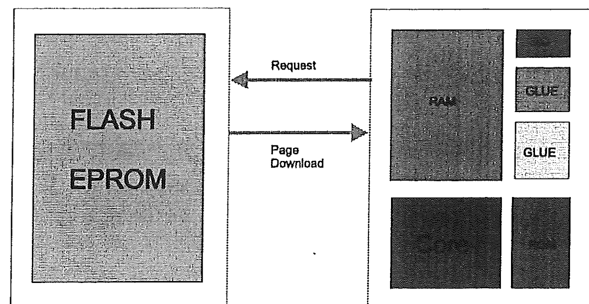

Figure 3-0-2: Soft coded System on Chip (SOC), with download capability.

3.3 Core with external RAM and external ROM

This is the well-known microprocessor system, there are already many solutions available with existing processors. These systems belong to a different category and can be found in palm-sized computers, digital cameras etc. This is the domain of upcoming 32-bit processors. They should be not discussed here any further.

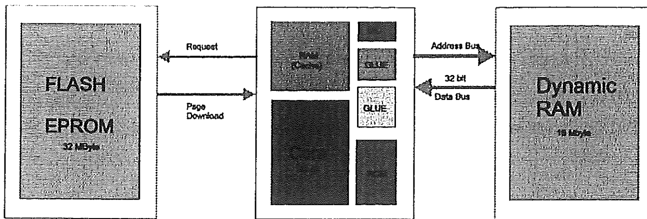

Figure 3-0-3: Core with external RAM and ROM, complete microprocessor system.

3.4 Main Application goal of ANTARES

Hard-coded SOC contain only few Kbytes of ROM and very limited RAM, so programming will stay on the assembler level at most. It makes no sense to write a 2 Kbytes program in a high level language, only few features of the C-language will be used.

So this development here targets on the **soft-coded SOCs** with some kind of download, where the main program storage is outside in a mass storage FLASH-EPROM or an EEPROM. These devices with up to 32 Mbytes are very small in size and easy to interface, generally with a byte serial interface similar to a magnetic disc system. They are not designed to run programs directly from the FLASH, programs have to be serial downloaded into the SOC-RAM before execution. So we have here some realm of the old overlay programming technique from the early days of computer science. With the ability to integrate larger dynamic memory on SOCs, this will be more and more efficient.

External program storage in a mass storage device eliminates nearly all constraints on program size and makes high-level language programming effective. Although the internal RAM has to be large enough to keep the actual pages, for many applications 4 …6 Kbytes are sufficient. For applications with big storage requirements for variables like image processing, this is of course not sufficient. Here we need a big external RAM, but this is another category of systems.

To use these features of paging and downloading, a build in operation system (BIOS) is required. The BIOS must also manage the interrupt levels and system tasks and must be resident, or most of it, in the on-chip ROM.

3.4 Position of BIOS ROM in the memory map

The ROM must be placed there in the address space, where the program counter addresses the first instruction after reset. This may be at address 0000h or somewhere in the upper memory, we define here the address FFF6h. The lower memory 0000h is not a good selection for reset start-up-address. This low memory may be better used for variables, which can be addressed with only 8 bytes. If we put ROM to the lower memory, all 8-byte addresses will be in the ROM and can't be used for variables or tables. Fig 3-4 shows the general memory map for ANTARES.

The BIOS ROM must contain the following service routines as a minimum:

- Initialization of system hardware,
- Setup of the interrupt table
- Routines for feature extension, division, block-copy and floating point processing (if used),
- Routines for error processing,
- Routine for communication via serial devices,
- Routines for program download and paging.
- Routines for memory allocation.

All routines are linked with the application program via the software interrupt mechanism, allowing isolation of BIOS from the application program. So absolute addresses of BIOS routines must not be known to the programmer during software development, only the reference identifiers (names) of the related vectors, which are tabulated. The same is with input/output routines and communication via a serial link. These routines are accessed via the SWI mechanism and all details are hidden to the user.

3.4 Mapping of memory sectors to the RAM

We have decided, mapping the ROM to the upper edge and the RAM to the lower edge of the memory space, with the interrupt vector table at the low end. The rest of RAM – memory must cover

- the Stack,
- the global variables section,
- the constants section,

the code segment.

As mentioned before the used C-compiler supports the stack architecture, using stack for local memory, for return address and for arguments of functions. Access to this dynamical administrated memory space is done via stack relative addressing. So the stack pointer is decremented when putting data on stack and incremented when data is removed. Deepness of stack depends on the nesting level of the routines, which is non-deterministic in an interrupt environment and the amount of local memory used. In each case, the stack should have a certain volume, so we decide to put the stack just below the top of the RAM.

Variables, which are declared before the main( ) – program, are called "global variables" and are accessible from everywhere in the program. But global variables are expensive because they are translated to absolute addressed memory cells by the compiler, residing fix and forever in memory.

The compiler further differentiates between initialized variables and uninitialized variables. Initialized variables have to be initialized before program start, which only can be done by the loader, which copies constants to those variables. Both type of variables have to be mapped to a space in RAM, where they can easily bee accessed, so we decide to put these global variable sector as low as possible into the RAM. If there are only few, we can put them on top of the interrupt table. We can access these global variables directly with the 8-bit address field without using the prefix mechanism. May be the BIOS uses some own buffers and global variables, these space requirements have to be put on the interrupt table first. The first address, which is free, is then available for the application code.

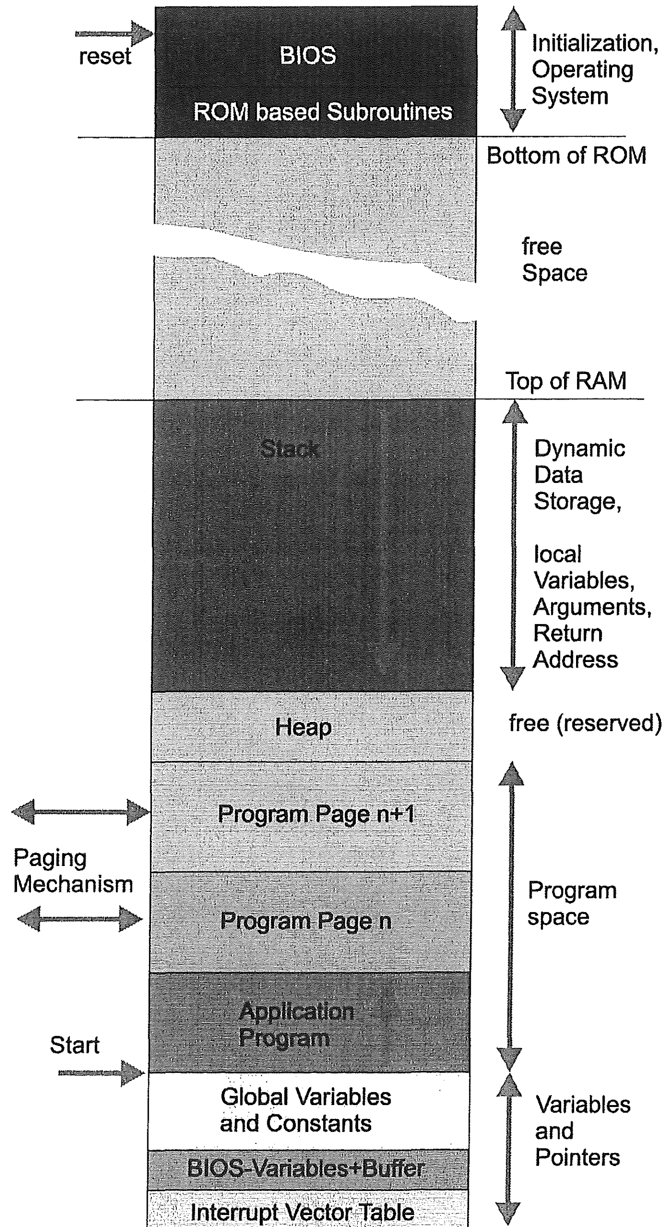

Figure 0-3 Memory map of an ANTARES based design

The number of global variables is fixed, so the first instruction of the main program can follow directly on top. The space above the main will be broken down into pages of 512 bytes each, used by the paging mechanism, with as many pages loaded as actually needed. These pages will contain most of the functions, called by the main program or by other functions in a nested manner. There should be some space left between the top of the application program and the bottom of the stack. This space, the heap, can be used, if there is some intelligence build into the BIOS concerning memory allocation, but normally not.

What's about the constants section? Constants used in the program are collected in a special memory segment, the constant section. All constants, which doesn't fit into the instructions, fi all float and long constants are found here as well as text strings defined for display. There are two choices to handle these constants:

1. Handle constants like initialized variables and put them into the global variables sector,
2. Handle constants like code and keep them together with the related function.

The second would be better but is not supported by the compiler, which collects all constants from the overall

program in one sector, which may be near for some functions, but far for others (needs prefix which is not effective). So we have to take the first variant which is much easier to implement too. So we decide, to handle constants like initialized variables and put them into the same sector. Their values are loaded during program initialization by the loader. With this decision, constants are similar expensive as global variables and should be used only rare.

There may be some problems with this concept in cases where there are many text outputs, eating up all the global variable space. So we leave the details of constant handling to the programmer, who can decide how to handle it. The above-described way is the default one, which can be overwritten by other definitions (all defined in the BIOS).

See figure 3-4 for a memory map of an ANTARES based SOC-design.

4 Instruction Set Design

After defining the general architecture lay out of ANTARES, we now have to come to the details of instruction set design. There is a close interdependence between architecture, instruction set and compiler design. Some linkage is discussed in the chapters before. ANTARES is special designed for effective C-compilation, so we have now to explain this special relation. We will do this in this and the next chapter, both have to be read in close connection to each other. Many decisions in this chapter can only be fully understand by using the information of the compiler chapter.

4.1 Instruction Coding

The early decision on using a consequent 16-bit instruction set format lays hard constraints on the coding of the instructions. We decide to use 6 different instruction formats, listed in table 4-1.

See the instruction reference table in the annex for a complete listing and explanation of all instructions.

Format A: Instructions without extensions

This type of instruction doesn't carry any immediate or register address fields. There are 8 bits (marked XXX... in the table), that allow differentiating between different instructions. We have coding space for 256 instructions of type A. Examples are DIS (disable interrupt), NOP (no action) and SCF (set condition flag).

Table 4-1 : Instruction formats

	N	15 12	11 8	7 6 5 4	3 2 1 0
A	256	0 0 0 0	0 0 0 0	X X X X	X X X X
B	8	0 0 X X	X 0 0 1	N N N N	N N N N
C	256	0 0 R R	R 0 1 0	X X X X	X X X X
D	32	0 0 R R	R 0 1 1	X X X X	X R R R
E	8	0 1 R R	R X X X	N N N N	N N N N
	8	1 0 R R	R X X X	N N N N	N N N N
F	4	1 1 X X	N N N N	N N N N	N N N N

Format B: Instructions with 8-bit vector on bus

These type of instructions contain an 8-bit immediate field (NN...), but no register address field. Examples are the input and output instructions PIN, POT, the instruction to load the prefix register LPR and RET. From the remaining 8-bit of the coding space three bits are used to differentiate, so there are maximum of 8 different instruction codes of that type, 6 are used in this design.

Format C: Instructions working on one register

These type of instructions code instructions for unary operations like CLR a register, INV a register etc., but also the indexed load register instruction LDX can be found here, because register X is implied indirectly and no addressing is needed for X. Because there is no immediate field and only three bits are used for register addressing, there are 256 possible codes in the coding space, only 30 are used here.

Format D: Instructions with two registers addressed

These type of instructions contain two register address fields of 3 bit each, the coding space is quite small with a maximum of 32 possible codes, with 16 used. The remaining 16 codes are reserved for further expansion in a 32 bit-architecture (double word instructions). Examples for instructions of type D are add register to register ADD, compare register with register CEQ, or multiply register with register MPY. In all these cases the second register will be overwritten by the result as outlined before. A true three-register instruction type is

not needed and not possible to implement with an instruction format limited to 16 bit.

Format E: Instructions with one register addressed and an immediate constant field

These instruction type codes instructions like LDI, loading a constant in a designated register, ADI, adding a constant to a designated register etc. With 3 bits used for the register address and 8 bit used for the constant immediate field, there is only very small coding space left for this type of instructions. So there are only 16 instructions possible, which are all used. The constant field is also used to carry addresses as in the absolute register load instruction LDA, or to define offsets in the relative load instruction LDW and the stack relative load instruction LDS.

Format F: Instructions with long offset

These instructions use 12-bit from 16-bit to carry a long 12-bit offset, used for pointers (LEA instruction), absolute jumps JMP and relative jumps JPR and the subroutine call CAL, which is a program counter relative call instruction. These 4 instructions are the maximum that can be coded. The relative large offset of 12-bit is further improved, taking addresses as word addresses, meaning there is a 13-bit local access space without any need for a prefix mechanism. With 13-bit, we reach nearly all cells of the typical RAM space, implemented in most SOCs of this genre.

Coding of instructions is done in a Huffman-code style, with the most expensive instructions in format F and the second expensive in format E. There are no further possibilities to add instructions of that type. There are already some compromises been made related to the conditional jump instructions, which are decomposed into two instructions, a compare instruction like CEQ and the jump instruction JPR. CEQ sets a flag Cd, if a jump is needed, so there is no unconditional jump. To perform an unconditional jump, the flag Cd has to be set with a special instruction SCF, which is a format-A-instruction. So the unconditional jump instruction can be avoided and coding space is saved.

4.2 Interdependence between Compiler and Instruction Set

The used LCC-compiler [FraHa], which is configured to ANTARES with a machine description file, uses, like nearly all other existing C-Compilers, an. Fraser and Hanson call elements of this language intermediate language or **meta-language**. The parsed C-expressions are first translated into the meta-language, which is machine independent "**directed acyclic graphs**" or DAGs. DAG operators are forming one tree, or more then one tree, a "forest" for each expression in the C-language. There are only few DAG operators needed to describe all kind of operations, see table 4-1.For more details see [FraHa].

The type suffixes in table 4-1 describe the types of the arguments, see table 4-2. There are additional suffixes for the number of bytes used in the operation, so a complete description for an integer ADD with integer defined as 2 bytes is

```
targ = ADDI2(arg1,arg2),
```

or for negation of an unsigned long value

```
targ = NEGU4(arg1).
```

The arguments *arg1, arg2* may be registers as well as the target of the operation *targ*.

Table 4-1: Type suffixes for DAG operators

Type Suffix	*Variable Type*
F	Floating point variable
I	Integer variable
U	Unsigned integer variable
P	Pointer variable
V	Void
B	Block, structure type

To get the arguments into the registers there are DAG operators for fetching values from global or local memory, and for storing (assigning) variables to memory cells. The C-expression:

```
int a,b,c;
    c = a + b;
```

with a, b, c all global integer variables, will be described in DAG – notation with:

ASGNI2(ADRGP(c),ADDI2(INDIRI2(ADRGP2(a)), INDIRI2(ADRGP2(b))

The graphical representation of this easy acyclic graph or tree is in figure 4-1.

Table 4-2: Directed acyclic graphs (DAG) operators, defined by Fraser and Hanson, as a meta-language.

Operator	Type Suffixes	Operation
ADDRG	P	Global address
ADDRL	P	Local address
ADDRF	P	Parameter address
CNST	FIUP	Constant
ARG	FIUPB	Function argument
LABEL	V	Label definition
INDIR	FIUPB	Fetch variable
ASGN	FIUPB	Store variable
DAGs with one argument		
BCOM	IU	Bitwise complement
NEG	FI	Negate
LSH	IU	Left shift
RSH	IU	Right shift
CVF	FI	Convert from float
CVI	FIU	Convert from integer
CVP	U	Convert from pointer
CVU	IUP	Convert from unsigned
DAGs with two arguments		
ADD	FIUP	Addition
SUB	FIUP	Subtraction
MUL	FIU	Multiplication
DIV	FIU	Division
MOD	FIU	Modulus
BAND	IU	Bitwise AND
BOR	IU	Bitwise OR
BXOR	IU	Bitwise EXOR
Control flow DAGs		
JUMP	V	Jump unconditional
EQ	FIU	Jump if equal
GE	FIU	Jump if greater or equal
GT	FIU	Jump if greater
LE	FIU	Jump if less or equal
LT	FIU	Jump if less
NE	FIU	Jump if not equal
CALL	FIUPVB	Function call
RET	FIUPV	Return from function

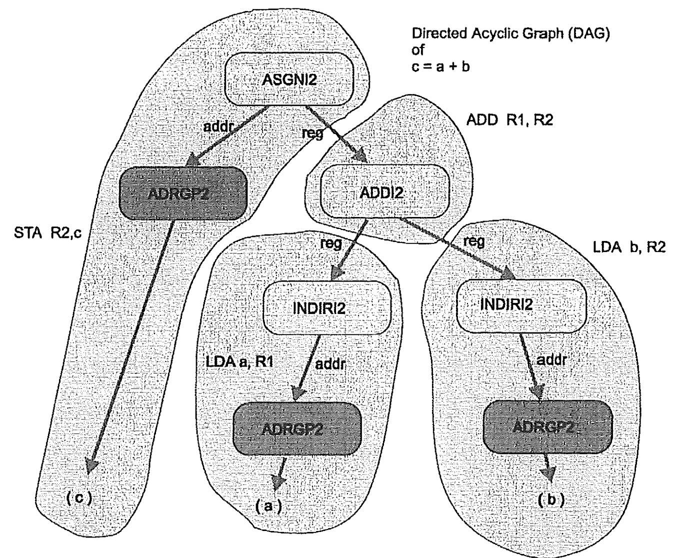

Fig 4-1: Graph of the tree (c = a + b)

The meta-language is machine independent. To map the language on a dedicated architecture, we need two main tasks:

- Linking assembler codes to the branches and trunks of the tree, which is called "**tree covering**",
- Managing the transfer and intermediate storage of variables during the operations, which is called "**register allocation**".

Both together form the heart of the **code generator,** which rules are described in the machine description file.

Tree covering means that we have to define an architectural representation or better machine instruction of the used DAG, fi for the integer adding operation

```
reg = ADDI2(reg , reg) => "ADD R1,R2"
```

For the absolute address fetch operation:

```
reg = INDIRI2( ADRGP2(a))=>"LDA a,R1"
```

And for the absolute address store operation:

```
ASGN(ADRGP2( c),reg) => "STA c,R1"
```

The tree is covered (see figure 4-1) and then translated to the assembler program of

```
LDA  a, R1
LDA  b, R2
ADD R1, R2
STA R2, c
```

Register allocation is done automatic in a way that there is a continuo flow of data without any overwriting, which may be difficult in complicated expressions. The register allocator is part of the code generator and performs also the in lining of the instruction stream in the right order. With unlimited number of registers and no constraints in using them, this will give optimal compilation results.

But there are constraints:

- The number of registers are limited
- Some of the registers are used for certain operations (X, S) and can't be used fully free.

To avoid register clobbering, transfer instructions like `MOV R1, R2` have to be inserted, or in severe cases, the registers have to be spilled to memory. Register-register transfer and spilling is very ineffective and should be avoided if possible, lowering the overall performance.

So the first important thing to get an effective architecture is to provide as much registers as possible. With 8 registers, 6 of them without any constraints, the ANTARES architecture is good for most of type integer-, unsigned- and char– processing. For type long- and float- processing, where two registers have to be used to hold a value, there are only three double registers left. This is the absolute minimum.

Complicated expressions of type float or type long will surely trigger register spilling with big penalty of additional memory accesses. This can only be improved with spreading the architecture to 32 bit. Implications have been discussed before. So we have to accept limited performance in type long and float operations.

The second important thing is to design the instructions in that way, that there are no constrains in register use. The compiler should be able to choose argument registers free. Register allocation is eased. There should be only few register targeting, sometimes not avoidable as with the index-register X, where pointers have to be positioned. All other targeting was removed, as it showed up to be a big blockage for expression evaluation, lowering code performance significantly.

To **optimize tree covering**, we choose our instructions close to the DAGs, so we have in many cases a one to one representation of the DAG operator in the instruction code. This is very obvious for the DAGs with two arguments, which are processed by instructions, working on two registers, and with the DAGs with one argument, processed by instructions working on one register. See the instruction table in the annex for more detailed information.

The compare instructions are defined in the same way as the DAGs, but with the jump instruction separated as explained before. So conditional jumps are mapped on two consecutive assembler instructions. There is again some saving by using the equality

GT R1, R2 == LE R2, R1.

DAG operators with constants are directly mapped on instructions with immediate field.

Addressing modes are used with mapping global variables to absolute addressing, local variables to stack related addressing and pointer addressing to indexed addressing mode, using the index register X. Arguments to functions are pushed on the stack with the PSH instruction, returns are in registers.

The different size of the variable types needs some special care. Bytes are loaded and stored with dedicated instructions (LDB, SBA etc.), which is no problem with loading but with storing in a word oriented architecture. There has to be a certain mechanism to prevent the overwriting of the above byte when writing to memory. This has to be done in hardware. Further extensions of the instruction set are needed with sign extension and with calculation in bytes.

There are no instructions for long- or floating point-processing directly, because there are no 32-bit registers in the actual design. But for easing double word processing, some extensions to the instruction code are made. Fi there are now 6 different shift instructions to ease implementation of shifting to connected registers. The instruction IVC, inverting with carry added, allows the implementation of negation of a long number in only two instructions.

There are a few other instructions added to the code, which are intended to be used in a later to define operating system, which don't have representation in the C-language. These are fi the instructions for manipulating the flag register GFL, SFL, the input, output instructions PIN and POT and the interrupt control instructions DIS and ENI. There is also a hardware sense line included (CSE) with copying the sense input to the condition flag Cd. A sense line allows installing a hardware dependent waiting loop.

For improvement of some communication routines, there is also a calculation of register parity included. This can be easily done in hardware. These codes are used in assembler language only and can't be reached via the compiler.

The instruction table in the annex shows in detail all instructions, their coding and what they are doing.

5 Adapting the LCC-Compiler

The LCC "Little C Compiler" is a retargetable compiler, written by Fraser and Hanson [FRAHAN] and free available on the web. This compiler is much smaller than the GNU-C-compiler and there is only one machine description file, which has to be adopted to the target architecture. There are several other retargetable compiler available, see [LEURET], [MARW] fi the LANCE-Environment, developed at the university of Dortmund, Germany. LCC was selected at least because of the very good documentation in [FRAHAN] and the limited effort to get a reasonable result.

A compiler transforms C-code to assembler code. `rcc.exe` is the main compiler executable. There is also a more standard preprocessor `cpp.exe` for further preprocessing. The lexical analysis is done in the LCC with a scanner, which brakes the input into tokens. The next compiler phase parses the token stream according to the syntax rules of the C-language. There is also proof of semantic correctness. Outcomes of these processing are abstract syntax trees, transformed to acyclic graphs and formulated in an intermediate language. Elements or operators of this meta-language are DAG operators, as mentioned before. Common sub expressions are eliminated during this process and temporaries automatically inserted if needed. Optimization is locally, concentrated on each expression. Global optimization may be added later.

The compiler does code generation by covering the trees with assembler instructions, using the rules in the machine description file. There are only few changes done to the LCC source code itself as adding the ANTARES interface description to the bindings (in `bind.c`) and predefining ANTARES as the default target. All other targets are still included and can be called using the command line option. See original LCC documentation for all command line options.

LCC is delivered as a bundle of source files, which have to be compiled on the PC to build an executable.

The compiler needs rcc.exe and cpp.exe in the same directory. The executables are compiled for

```
C:\ANTARES
```

With the environment variable BUILDIR set to this directory. The executables should be installed there. This directory should also be added to the PATH. An environment variable

LCCDIR = c:\ANTARES

Has to be added.

Compilation with LCC is done by opening a DOS – window and typing the command line:

```
>>Lcc_ant abc.c -S -Wf-g2,; -o abc.asm
```

This compiles the file `abc.c` into the assembler file `abc.asm` with all C-code as comment after the ";" for each expression. The option -Wf... is needed.

During the compiler – building process the machine description file `ANTARES.md` is compiled into a file called `ANTARES.c`, which is a C-file. This compilation is done with **`iburg.exe`**, a program used widely [IBURG] for this purpose of code generator generation. `Lburg.exe` transforms a system of rules, described in the iburg-syntax, in standard C-code formulations, which can be included in the compilation of the compiler source code. So most of the machine description file is written in this syntax, the rest in C-language, because these parts are directly included into `ANTARES.c`.

The grammar of the iburg machine description file is shown in figure 5-1.

Grammar	→ {*dcl*} %% {*rule*}
dcl	→ %start *nonterm*
	| % term *{identifier = integer}*
rule	→ *nonterm: tree = integer [cost];*
cost	→ *(integer)*
tree	→ *term (tree, tree)*
	| *term (tree)*
	| *term*
	| *nonterm*

Figure 5-0-1: EBNF Grammar for iburg Specifications

The format of the machine description file `ANTARES.md`, obeying this grammar, is shown in figure 5-2. The C-code part of the declaration sections and the functions-sections are directly copied to the target file `ANTARES.c` and used without further modification. The section after the `%start` statement is a huge list of all terminals `%term`, which are further used by the rules sections. These terminals are the DAG-operators, which have been described before with suffixes for type and size. All needed operators have to be listed in this section. There are integer numbers assigned to each operator, which code the certain operator, so they are not accidental. To find this coding and the right list of numbers, Fraser and Hanson provide a short program `opsgen.exe`, which generates the required operators and their codes for a certain interface description.

```
%{
c-code declarations
%}
%start stmt
%term ARGB=41
.....
%term ADDI2 = 2357
%term SUBI4 = 4422
%%
.....
reg: ADDI2( reg, reg)  "?MOV  %0,  %c\nADD  %1,%c\n"
1
......
stmt:ASGNI2(addr, reg)   "STA %1,%0\n"   3
%%
C-code functions
Interface antaresIR
{
interface declaration
};
```

Figure 5-2: Format of the machine description file ANTARES.md

The rules-section contains the rules for tree covering and the associated assembler translation (code emitter section) in a string, linked to each rule. A rule must be placed in one line. All rules are combinations of nonterminals with stmt as the only terminal, defined in

`%start stmt.`

A tree must start with a statement, with nonterminals at the branches, which are called "kids" here. There are many rules with the same nonterminal like fi "`reg:`.." which are valid in parallel and have to be selected by the cost mechanism.

Last in the line is a number or a function, providing a number. This number defines the costs of this rule. If there is no number at all, cost is assumed as zero.

The cost mechanism is used to select the best rule coverage from equivalent coverage and may also be used to switch rules on and off by inserting a very high cost. As an example, the rules concerning the value of a constant can be taken:

```
con7: CNSTI2   "%a" range(a,-128,127)
```

This rule is only used, if the range( ) –function delivers a small cost value, which is true if "a" is between –128 and 127 (8 bit signed integer), else there would be a cost of IBURGMAX, which is defined of the maximum integer number. These functions are defined in the C-functions sections or elsewhere in the compiler.

For further discussion of the machine description file this file is provided in the annex.

The code emitter section of the rules uses certain syntax with the most important symbols described in table 5-1.

An emitter string starting with # calls for a special treatment routine `emit2( )`, which can be configured as needed. This is not needed for ANTARES, the `emit2( )` routine is empty but already provided.

Table 5-1: Used symbols in the emitter section

Symbol	*Meaning*
%0, %1 ...% 9	Arguments (kids), starting numbering with %0, left to right
%a	Alpha, name or identifier
%c	Current register or identifier used
%F	Framesize
%s	String
Letter	Every letter as printed
#	Call for emit2( ) for special treatment
?	First instruction is removed, if same register used
\n	Carriage return, end of line, end of statement

An assembler statement is marked in the emitter string by a `\n` end of line termination, if there is no `\n` in the string, the string is not assumed as an assembler code. In this case there is further processing of the requested arguments (kids).

Arguments, which are the results of branches of the tree in a nested manner, are called by %0, %1 etc. This mechanism is used to link tree branches together, which combined represent one assembler statement. As an example, the access to a global memory variable "*wert*" is:

```
addr: ADRGP2          "%a"              2
reg: INDIRI2(addr)    "LDA   %0, %c\n"  3
```

These two rules combined deliver the emitted assembler instruction

```
LDA wert, AB1
```

With *wert* as a kid of the second rule and AB1 as the actual register used. The first rule does not emit any code itself, it is not a statement (there is no \n), but provides information for the above rule. The linking can cover more than two rules, with the kids calling for kids again.

The question mark "?" tells the emitter to skip the first statement (before the \n end of line character), if the actual and the target register are the same. This mechanism is widely used with the register – register instructions. The rule mechanism itself supports a three-address instruction, combining two registers and placing the result in a third register. As mentioned before, we can only provide two select-addresses in the instruction code, so we have to use one of the kid-registers as the target register, overwriting the content. To build in a flexible element, a MOV – instruction is included for this register, so the original variable is first copied to a second register and than this register is overwritten by the result. With this penalty of one MOV-instruction, the original three-address performance is restored. But the MOV is not needed in each case, only if both variables in the expression have to be reused later. This seldom happens. For this, the question mark removes the MOV-instruction in most expressions and there is no penalty anymore. The mechanism is the same as for the X86-code [FraHan].

The above mechanism doesn't fully works with double word processing (long, float), where we need two succeeding MOV-instructions for the same purpose. So with the "?" only the first MOV is eliminated, the second MOV R1, R1, which is a move to itself, will be emitted. This can be further eliminated by some optimization done after code emission, but is not implemented yet.

The frame size %F is used with stack-relative addressing. The number of used local address cells on the stack (frame) is needed to calculate the address offsets to reach this variables from the actual S content. The frame is finally defined after code compilation, so these offsets can only be calculated by the assembler.

In the following we describe the machine description file in detail with regard to ANTARES, first the rule section, then register steering and at least the C-functions, added to the file.

5.1 Rules description

The rules section starts with rules, which are machine independent, allowing to read from registers and to write to registers. The first machine dependent section are the constants-rules:

```
con0:   CNSTI2      "0"  range(a,0,0)
con0:   CNSTI1      "0"  range(a,0,0)
con0:   CNSTU1      "0"  range(a,0,0)
con0:   CNSTU2      "0"  range(a,0,0)
con1:   CNSTI1      "1"  range(a,1,1)
con1:   CNSTU1      "1"  range(a,1,1)
con1:   CNSTI2      "1"  range(a,1,1)
con1:   CNSTU2      "1"  range(a,1,1)
con2:   CNSTI2      "2"  range(a,2,2)
con2:   CNSTU2      "2"  range(a,2,2)
conM1:  CNSTI2      "-1" range(a,-1,-1)
con7:   CNSTI1      "%a" range(a,-128,127)
con7:   CNSTI2      "%a" range(a,-128,127)
con8:   CNSTU1      "%a" range(a,0,255)
con8:   CNSTU2      "%a" range(a,0,255)
con12:  CNSTU2      "%a" range(a,0,4095)
con7p:  CNSTP2      "%a" range(a,-128,127)
con12p: CNSTP2      "%a" range(a,0,4095)
con16:  CNSTI2      "%a" range(a,-32768,32767)
con16u: CNSTU2      "%a" range(a, 0 , 65535)

aconp:  ADDRGP2     "%a"   2
aconp:  con7p       "%0"
aconp:  con16u      "%0"   2

aconu:  con16u      "LPR %a\256\n%0%%256"  2
aconu:  con12       "%0"
aconu:  con8        "%0"

acons:  con7        "%0"
acons:  con16       "LPR %a\256\n%0%%256"  2
acons:  con1        "%0"
acons:  conM1       "%0"
acons:  con0        "%0"
```

Figure 5-3 Rules for constants, ANTARES.md

There are several rules defined for constants, related to the different size of the immediate field or for certain fixed constants like 0, 1 or two, which require special treatment. Further differentiation is needed for signed

or unsigned representation as well as for pointer constants. In each case, the usage of the rule is controlled by a `range( )` function, switching the related rule on, if the constant is in range, or setting the cost to maximum, so that this rule is not used anymore.The following rules combine some constants-rules for easier handling. Here we can find the first assembler code emission for 16 bit constants, which require the prefix mechanism. The rule `aconu:` and `acons:` emit an assembler code:

```
"LPR %a\256\n%0%%256"          2
```

With `%a` as a number fi 2356 and `%0` the next code, using the number, this leads to the prefix instruction:

```
LPR  2356/256
LDI  2356%256
```

The constant is divided into a high byte by the division operation and the low byte by modulo operation. The High byte is included into the prefix operation LPR, the low byte is included in the instruction, here LDI as an example, as an immediate 8 bit constant.

Next is the rule

```
stmt: reg " "
```

This rule means that there are also statements possible which starts as a register. The empty string references to the kid rules. This rule is needed in any case and is machine independent.

```
reg: INDIRI1(addr)      "LBA %0,%c\n"                 3
reg: INDIRI2(addr)      "LDA %0,%c\n"                 3
reg: INDIRU1(addr)      "LBA %0,%c\n"                 3
reg: INDIRU2(addr)      "LDA %0,%c\n"                 3
reg: INDIRP2(addr)      "LDA %0,%c\n"                 3
reg: INDIRI4(addr) "LDA %0,%cl\nLDA %0+2,%ch\n"   5
reg: INDIRU4(addr) "LDA %0,%cl\nLDA %0+2,%ch\n"   5
reg: INDIRF4(addr) "LDA %0,%cl\nLDA %0+2,%ch\n"   5
```

Figure 5-4: Rules for memory access

Figure 5-4 shows the rules for memory access, which is done by the DAG-operator INDIR. There is a rule for each kind of type. This operator places the result in a register and needs an address as an argument (kid). The shown rules are for global memory accesses only, because the rule `addr:` delivers an absolute address. So the related codes can be emitted for absolute memory access like

```
LDA %0,%c\n     => LDA wert, ABl
```

With `%0` referencing to the kid which is ADDRGP2(*wert*) and `%c` referencing the current (target) register `ABl`. See the rule for absolute addresses later.

Fetch of 32-bit variables like *long* or *float* type needs two consecutive instructions. We need placing of the low word of the long variable in the current register AB, ABl (low), and the high word of the long variable, which is accessed by `%0+2`, or "longwert+2" in the high word of the current register AB, ABh (high).

This is the right place to explain the strange register naming. For the register allocation mechanism, we need 2 sets of registers, one set of 16-bit registers `intreg`, declared as

```
static Symbol intreg[32],dblereg[32]
```

in the declaration part of ANTARES.md, and 32-bit registers `dblereg`. There are 32 registers each declared, but of course only 8, (or 3) used. The mask setting does this. These registers are using the same hardware, which are 8 16-bit registers as described before. Long operations are considered as using the double registers `dblereg`, integer operations using the smaller `intreg`. Six of these `intreg` are combined logically to three `dblereg`, which are named:

```
AB   = Abh_ABl
CD   = CDh_CDl
EF   = EFh_EFl
```

The high word of this double register is ABh, the low word ABl. We can reference to this register halves by adding the suffixes "h" or "l" to the register name in the emitter, as it is done in the rules for long and float processing (see `%cl\n` ...`%ch\n`). If we use the same registers for integer operations, we call them directly ABl or ABh without any suffix. The assembler only knows ABl, ABh, etc and doesn't care on register size. With this trick, the register allocation mechanism works as well for long- as for int- type variables.

Figure 5-5 shows the address operation rules. There are three type of addresses

```
addr:        absolute address
addrx:       index address
addrs:       stack relative address
```

as described before. There is more than one rule for each address type. Absolute addresses can be a global variable in memory (ADDRGP2) with a name required

or a 16-bit constant as defined before. There are several rules for index addressing, some of them difficult to understand. One of these rules is loading of a pointer with LEA-instruction. The other rules are needed together with address calculation (ADDP2) or if an already loaded value is treated as an address. The terminal VREGP represents such a value.

```
addr: ADDRGP2                    "%a"
addr: con16u                     "%0"

addrx: ADDI2(reg,acons)          "ADI %1,%c\n"   1
addrx: ADDU2(reg,aconu)          "ADI %1,%c\n"   1
addrx: ADDP2(reg,aconp)          "ADI %1,%c\n"   1
addrx: LOADP2(memx)              "MOV %0,%c\n"   1
addrx: memx                      "MOV %0,%c\n"   1
addrx: INDIRP2(addr)             "LEA %0\n"      1
addrx: INDIRP2(VREGP)                 ""
addrx: ADDP2(addr,regx)          "ADD %0,%1\n"   1
addrx: ADDP2(reg,regx)           "ADD %0,%1\n"   1

addrs: ADDRFP2                         "%a"
addrs: ADDRLP2                         "%a"
```

Figure5-5: Addresss operation rules

Mapping the ADDRLP and ADDRFP, standing for access to local variables and local parameters, to `addrs:` does stack related addressing. This is a design decision made for ANTARES. Other mapping is possible, but with effective stack instructions this is the best way to manage intermediate, dynamic storage.

This sector of the machine description file is the most critical and the most sensitive and there may be still some errors or dead rules (rules, which are never used). Extended testing is needed to clarify all the details and implications.

The following rules are all placing results in a register, so they are all called *reg:* figure 5-6. We start with the rules to fetch a variable.

These are the rules for fetching a variable in the different addressing modes. The first two rules `regs:` and `regx:` are special cases, getting the stack address or the address from the index- register for further treatment. The `stmt:` rule is needed to load the X register with a pointer address in a separate tree for further using it in index addressing (LEA-instruction). Fetching a memory cell with indexed addressing is in the following rules, with the index-register referenced as VREGP. There has to be a special register steering in the C-code section to use the X-register for that, as it will be explained later. The next rules using the kid rules `addrx:` for index addressing, a second mechanism activated.

```
regs:   addrs    "MOV S,%c\nADI %0,%c\n"        2
regx:   addr     "LEA %0\n"                     1

stmt:   ASGNP2(VREGP,LOADP2(addr)) "LEA %0\n"  1

reg:    INDIRU2(LOADP2(INDIRP2(VREGP))) "LDX %c\n 1
reg:    INDIRI2(LOADP2(INDIRP2(VREGP))) "LDX %c\n"1
reg:    INDIRI1(LOADP2(INDIRP2(VREGP))) "LBX %c\n"1
reg:    INDIRU1(LOADP2(INDIRP2(VREGP))) "LBX %c\n"1
reg:          INDIRU4(LOADP2(INDIRP2(VREGP)))    "LDX
%cl\nADI 2,X\nLDX %ch\n"    3
reg:        INDIRI4(LOADP2(INDIRP2(VREGP)))      "LDX
%cl\nADI 2,X\nLDX %ch\n"    3
reg:        INDIRF4(LOADP2(INDIRP2(VREGP)))      "LDX
%cl\nADI 2,X\nLDX %ch\n"    3
reg:    INDIRU2(INDIRP2(VREGP))         "LDX %c\n" 1
reg:    INDIRI2(INDIRP2(VREGP))         "LDX %c\n" 1
reg:    INDIRI1(INDIRP2(VREGP))         "LBX %c\n" 1
reg:    INDIRU1(INDIRP2(VREGP))         "LBX %c\n" 1
reg:    INDIRU4(INDIRP2(VREGP))         "LDX  %cl\nADI
2,X\nLDX %ch\n"    3
reg:    INDIRI4(INDIRP2(VREGP))         "LDX  %cl\nADI
2,X\nLDX %ch\n"    3
reg:    INDIRF4(INDIRP2(VREGP))         "LDX  %cl\nADI
2,X\nLDX %ch\n"    3
reg:    INDIRU2(addrx)                  "LDX %c\n" 1
reg:    INDIRI2(addrx)                  "LDX %c\n" 1
reg:    INDIRI1(addrx)                  "LBX %c\n" 1
reg:    INDIRU1(addrx)                  "LBX %c\n" 1
reg:    INDIRI4(addrx)                  "LDX
%0,%cl\nADI 2,X\nLDX %0h\n" 3
reg:    INDIRI4(addrx)                  "LDX
%0,%cl\nADI 2,X\nLDX %0h\n" 3
reg:    INDIRF4(addrx)                  "LDX
%0,%cl\nADI 2,X\nLDX %0h\n" 3
reg:    INDIRI1(addrs)             "LDS %0,%c\n"      1
reg:    INDIRU1(addrs)             "LDS %0,%c\n"      1
reg:    INDIRU2(addrs)             "LDS %0,%c\n"      1
reg:    INDIRI2(addrs)             "LDS %0,%c\n"      1
reg:          INDIRI4(addrs)          "LDS   %0,%cl\nLDS
%0+2,%ch\n"     2
reg:          INDIRU4(addrs)          "LDS   %0,%cl\nLDS
%0+2,%ch\n"     2
reg:        INDIRF4(addrs)       "LDS        %0,%cl\nLDS
%0+2,%ch\n"
reg: LOADI1(reg)   "?MOV %0,%c\n"      move(a)
reg: LOADI2(reg)   "?MOV %0,%c\n"      onlysize2(a)
reg: LOADI2(reg)   "?MOV %01,%c\n"     onlysize4(a)
reg: LOADU1(reg)   "?MOV %0,%c\n"      move(a)
reg: LOADU2(reg)   "?MOV %0,%c\n"      move(a)
reg: LOADP2(reg)   "?MOV %0,%c\n"      move(a)
reg: LOADI4(reg)
        "MOV %01,%cl\nMOV %0h,%ch\n"  move(a)
reg: LOADI4(reg)
        "MOV %01,%cl\nMOV %0h,%ch\n"  move(a)
reg: LOADF4(reg)
        "MOV %01,%cl\nMOV %0h,%ch\n"  move(a)
```

Figure 5-6: Rules for variable access

The next rules in this sector references `addrs:`, as we know this is stack related addressing, the last section

refers to fetching information from an other register with a LOADI2(reg), which translates to a MOV R1,R2 instruction.

The instruction for long and float emit two assembler instructions as before described to consecutive addresses. There are some special cases handled with a cost function `onlysize(a)`, which is defined in the C-code section and differentiates the size of the variable. The rule is only switched on, when the size of the variable is of that size required for this rule. The cost function `move(a)` does some post code optimization as described in [FraHan].

```
reg: con0      "CLR %c\n"                         1
reg: con7      "LDI %0,%c\n"                      1
reg: con8      "LDI %0,%c\n"                      1
reg: con12p    "LEA %0\nMOV X,%c\n"               2
reg: con16     "LPR %0/256\nLDI %0%%256,%c\n"     2
reg: con16u    "LPR %0/256\nLDI %0%%256,%c\n"     2
```

Figure5-7: Rules for loading constants to registers

The next rules are quite easy to understand figure 5-7. Setting the actual register to zero (CLR-instruction) is equivalent with loading the constant `con0`:. Loading an 8-bit constant to a register we can use the dedicated instruction for this purpose LDI. The 8 bit constant may be signed `con7:` or unsigned `con8:`. A 12-bit pointer constant may be loaded with LEA, a 16-bit constant needs the prefix instruction LPR before LDI to load the upper half of the word. Again this rule is written for signed and unsigned values.

The next rules describe the instructions, working on one register, but with the result may be placed in another register, see figure 5-8.

```
reg: BCOMI2(reg)   "?MOV %0,%c\nINV %c\n"       1
reg: BCOMU2(reg)   "?MOV %0,%c\nINV %c\n"       1
reg: BCOMI4(reg)   "?MOV %01,%cl\nMOV %0h,%ch\n
  INV %cl\nINV %ch\n"    move(a)
reg: BCOMU4(reg)   "?MOV %01,%cl\nMOV %0h,%ch\n
  INV %cl\nINV %ch\n"    move(a)
reg: NEGI2(reg)    "?MOV %0,%c\nNEG %c\n"       2
reg: NEGI4(reg)      "?MOV  %01,%cl\nMOV%0h,%ch\nNEG
%cl\nIVC %ch\n"       move(a)
reg:  NEGF4(reg)                  "?MOV   %01,%cl\nMOV
%0h,%ch\nPSH %cl\nSWI FLTNEGVECTOR\n"  15
reg: ADDI2(reg,con1)   "?MOV %0,%c,INC %c\n"   2
reg: ADDU2(reg,con1)   "?MOV %0,%c,INC %c\n"   2
reg: ADDP2(reg,con1)   "?MOV %0,%c,INC %c\n"   2
reg: SUBI2(reg,con1)   "?MOV %0,%c,DEC %c\n"   2
reg: SUBU2(reg,con1)   "?MOV %0,%c,DEC %c\n"   2
reg: SUBP2(reg,con1)   "?MOV %0,%c,DEC %c\n"   2
reg: LSHI2(reg,con1)   "?MOV %0,%c\nSAL %1,%c\n"  2
reg: LSHU2(reg,con1)   "?MOV %0,%c\nSLL %1,%c\n"  2
reg:  LSHI2(reg,con2)    "?MOV  %0,%c\nSAL  %1,%c\nSAL
%1,%c\n"       3
reg:  LSHU2(reg,con2)    "?MOV  %0,%c\nSLL  %1,%c\nSLL
%1,%c\n"       3
reg: LSHI2(reg,reg)    "PSH %0\n
       SWI SIGINTLEFTSHIFTVECTOR\n"          12
reg: LSHU2(reg,reg)    "PSH %0\n
       SWI UNSIGINTLEFTSHIFTVECTOR\n"        12
reg:     LSHU4(reg,con1)         "?MOV     %01,%cl\nMOV
%0h,%ch\nSLL %11,%c\nSLC %1h,%c\n"    move(a)
reg:     LSHI4(reg,con1)         "?MOV     %01,%cl\nMOV
%0h,%ch\nSLL %11,%c\nSAL %1h,%c\n"    move(a)
reg: LSHI4(reg,reg)    "PSH %01\nPSH %0h\n
       SWI SIGNLONGLEFTSHIFTVECTOR\n"         12
reg: LSHU4(reg,reg)    "PSH %01\nPSH %0h\n
       SWI UNSIGLONGLEFTSHIFTVECTOR\n"       12
reg: RSHI2(reg,con1)  "?MOV %0,%c\nSAR %1,%c\n"  2
reg: RSHU2(reg,con1)  "?MOV %0,%c\nSLR %1,%c\n"  2
reg: RSHI2(reg,reg)    "PSH %0\n
       SWI SIGINTRIGHTSHIFTVECTOR\n"         12
reg: RSHU2(reg,reg)    "PSH %0\n
       SWI UNSIGINTRIGHTSHIFTVECTOR\n"       12
reg:     RSHU4(reg,con1)         "?MOV     %01,%cl\nMOV
%0h,%ch\nSLR %ch\nSRC %cl\n"            move(a)
reg:     RSHI4(reg,con1)         "?MOV     %01,%cl\nMOV
%0h,%ch\nSAR %ch\nSRC %cl\n"            move(a)
reg: RSHI4(reg,reg)    "PSH %01\nPSH %0h\n
       SWI SIGNLONGRIGHTSHIFTVECTOR\n"        12
reg: RSHU4(reg,reg)    "PSH %01\nPSH %0h\n
       SWI UNSIGLONGRIGHTSHIFTVECTOR\n"      12
reg: ADDI2(reg,con7)   "?MOV %0,%c\nADI %1,%c\n"  1
reg: ADDI2(reg,con16)  "?MOV %0,%c\nLPR %1/256\nADI
%1%%256,%c\n"       2
reg: ADDU2(reg,con8)   "?MOV %0,%c\nADI %1,%c\n"  1
reg:        ADDU2(reg,con16u)        "?MOV        %0,%c\nLPR
%1/256\nADI %1%%256,%c\n"     2
reg: SUBI2(reg,con7)   "?MOV %0,%c\nSBI %1,%c\n" 1
reg: SUBI2(reg,con16)  "?MOV %0,%c\nLPR %1/256\nSBI
%1%%256,%c\n"       2
reg: SUBU2(reg,con8)   "?MOV %0,%c\nSBI %1,%c\n" 1
reg:        SUBU2(reg,con16u)        "?MOV        %0,%c\nLPR
%1/256\nSBI %1%%256,%c\n"     2
reg: BORI2(reg,con7)   "?MOV %0,%c\nORI %1,%c\n" 1
reg: BORI2(reg,con16)  "?MOV %0,%c\nLPR %1/256\nORI
%1%%256,%c\n"       2
reg: BORU2(reg,con8)   "?MOV %0,%c\nORI %1,%c\n"  1
reg:        BORU2(reg,con16u)        "?MOV        %0,%c\nLPR
%1/256\nORI %1%%256,%c\n"                  2
reg: BANDI2(reg,con8) "?MOV %0,%c\nANI %1,%c\n"  1
reg:       BANDI2(reg,con16)       "?MOV        %0,%c\nLPR
%1/256\nANI %1%%256,%c\n"                  2
reg: BANDU2(reg,con8) "?MOV %0,%c\nANI %1,%c\n"  1
reg:       BANDU2(reg,con16u)       "?MOV        %0,%c\nLPR
%1/256\nANI %1%%256,%c\n"                  2
reg: BXORI2(reg,con8) "?MOV %0,%c\nXRI %1,%c\n"  1
reg:       BXORI2(reg,con16)       "?MOV        %0,%c\nLPR
%1/256\nXRI %1%%256,%c\n"                  2
reg: BXORU2(reg,con8) "?MOV %0,%c\nXRI %1,%c\n"  1
reg:       BXORU2(reg,con16u)       "?MOV        %0,%c\nLPR
%1/256\nXRI %1%%256,%c\n"                  2
```

Figure 5-8: Rules working on one register, placing the result in an other register and rules for combining constants with registers.

The first rules are straight forward with the already explained question mark mechanism, a one to one representation of DAG-operator and assembler instruction (BCOMI2 => INV). This is more complicated for long operators, but can be done with two or more instructions. The floating point subtraction

routine is called via the SWI-mechanism, with the address of the vector coded as the macro FLTSUBVECTOR, which will be defined at assembler level.

The next rules combine constants with register content. The simple looking INC-instruction is mapped to ADDI2 (reg, con1) with `con1:` is "1". The same is with the shift instructions, which are simple for integer but complicated for long. The general shift instruction, which allows a parameter as a second kid, is implemented with a software call, because there is no barrel shifter in the hardware. The instruction is seldom used, so this can be done without jeopardizing efficiency. The shift instructions for long variables need some additional effort, but much is covered by the layout of the shift instructions, using the carry bit for intermediate storage between the consecutive instructions. These rules have to be tested in detail with a simulator, it is difficult to evaluate each case on the paper alone.

The last rules cover the operations with constants, with the prefix mechanism (LPR instruction) for long constants used.

For the next rules section we refer to the complete listing in the annex, because these rules are using all the line and are difficult to read here.

The section contains the rules for register-register operation with two kids. Rules for integer are simple, rules for long and float need additional codes. Calling subroutine functions via the SWI mechanism does all floating point processing. The arguments are pushed on the stack, the result is steered to the second kid-register by register targeting (in the C-code section), so there are very few constrains in using the three double registers for float handling.

Multiplication of integers is done by a dedicated instruction, division is done by a software call as mentioned before. For long and float, software calls are implemented.

The following section of the machine description files contains rules for type conversion. Some are quite straightforward, some are more complicated. The floating-point types are again converted by subroutines.

The following section describes the rules for variable storage, which is mostly the same as for fetching data from memory. The same addressing modes are used.

Arguments to functions use a special DAG-operator ARG... that is mapped here to the PSH instruction, pushing the argument on stack. So arguments are passed to the function by using the stack as an intermediate storage. The stack pointer has to be adjusted with leaving the function, which is done in the related function routine in the C-section.

The jump and conditional jump instructions are described in the next rules. There is first an own definition of the address operators, which are now

```
addrja:    absolute jump
addrjr:    relative jump
addrjx:    indexed jump
```

All three types of jumping are implemented. Conditional jumps are formed out of a compare operation, preceding the jump instruction. The information from one instruction to the next is transported by the Cd-flag, included in the flag-register.

For conditional jumps of long arguments there are small macros included, to avoid SWI calls. Floating-point compares require again SWI calls to related routines.

The next rules allow block processing (structures) as defined in the C-language. Blocks are handled with pointers, but can be physically moved with the copyblock-subroutine.

Last section describes the rules of function calls and returns, which are implemented straightforward. Function calls are normally executed as relative calls, assuming some locality of the code. With 13-bit address space for the relative displacement, most of the RAM space is reached without prefix. Indexed call is implemented for pointer to function.

There is no code translated with function return. The RET instruction is implemented in the function frame, see C-code section. The assignment of values is done with assignment statements, so there is no need for a return translation [FraHan].

There may be still some bugs and errors in the machine description-file, which have to be eliminated in the next time with detailed testing of examples.

5.2 C-Code Section

After the rules-section there are all the C-function-bodies, which are referenced in the part before. Most of these functions can be taken without modification, some have to be adapted to the architecture.

The function **`progbeg( )`** is called with program start and contains the initialization of the compiler. This initialization is

- the set up of the registers, setting the masks for register allocation (tmask[IREG]) and defining the names of the registers intreg[BR]= mkreg( ), and
- emitting the header of the assembler file. The header of the assembler file contains additional macro definitions and some documentation notices. These outputs can be easily adapted to the requirements.

The function **`rmap( )`** defines, which register set, `intreg` or `dblreg`, is used for which size of variable.

The function **`segment( )`** defines the segments, used in memory to store

- code,
- initialized variables,
- uninitialized variables and
- constants

and how these memory-segments are named and emitted to the assembler code.

`Progend ( )` is called at the end of the compilation and is used to emit some final message.

`target( )` is a very important function, allowing register targeting for the instructions. There is a switch-case-loop, looking for the used DAG-operator in combination with the variable type. The function

```
rtarget(p, 0, intreg[XR]),
```

called from the case INDIR+P steers the register allocator to use a pointer (type P) in the X – register, when an indexed fetch operation is executed. This function relates to the kids (arguments) and can be also used for steering the kids of the kids.

The function

setreg(p, intreg[XR])

defines the X- register as the result register of that operation, which is here the case for ASGN+P, so pointers have to be loaded to the X – register.

Register steering is critical for compiler performance and with two many steering, or you can say to many constraints, there may be no solution and the compilation fails. So there are only a few guidance used, where it has to be done:

- For using the X-register as an index register,
- For defining the result –register of some subroutines like left-shift, right-shift, multiplication, division etc., which is needed because the registers are used to transport the results and the subroutine must know where to put them.

For most of all operations, the build in register allocator works very well.

The function **clobber( )** is used to spill registers, if the allocator is running out of registers. There are only few spilling needed under normal conditions and this function is critical to test. So spilling is done automatically in most of the cases and this function defines only additional spilling for certain situations.

The functions **`chstack_a( )`**, **`memop_a( )`**, and **`sametree( )`** were not changed and taken as is. They are not called as far as we could check that. We left them in to memorize them for other purposes.

The functions **onlysize4( )** and **onlysize2( )** are defined for ANTARES and used for differentiating between variable type sizes, delivering high costs for the wrong size.

The function **`emit2( )`** allows special treatment for codes on the code emission side as mentioned before (called with the # sign), but is not used here, so this function is in, but empty.

`Doarg( )` is left as is and only called from the compiler. The functions **`blkfetch( )`**, **`blkstore( )`** and **`blkloop( )`** can be used to emit code for copying blocks with structure processing, but they are not used here and block copying is better done in the subroutine **blockcopy**, called from the related codes instead.

The function **`local( )`** is not touched.

The next very important function is called **`function( )`** and defines the function frame and

body for function calls. This function has to be adapted in many ways. It is called with each function call in the C-program, with the call for `main( )` in the beginning.

There is no register saving convention yet with ANTARES, so we out commented the PSH instructions and the related POP instructions at entering the function and leaving the function. Register saving is kept to the responsibility of the programmer for the assembler subroutines for floating point multiplication etc., for the normal function call there is register managing done by the compiler.

The routine calculates the frame offset and the space needed for storing the locals and for accessing the function arguments. Stack pointer is adjusted after leaving the function with adding the frame size to the stack register S (ADI…), so that S points on the return address. With the return statement RET…, there is again a correction of the stack pointer register S, after returning to the calling program, but before the next instruction is executed. This mechanism assures that the stack pointer gets the same value after leaving the subroutine than it has before the function call, freeing the memory used for the arguments.

The next functions in the C-code section define, how memory space is named for further processing of the compiler. These are

- **`address( )`**, defining how symbol names are emitted,
- **`defconst( )`**, defining how constants have to be emitted
- **`defaddress( )`**, defining the way pointers are emitted,
- **`defstring( )`**, defining how character strings are emitted,
- **`export( )`**, defining how symbols are made public for further use in other programs,
- **`import( )`**, defining symbols which have to be imported from other programs,
- **`global( )`**, defining global variables,
- **`space( )`**, defining free reserved space in memory.

All these functions are modified for ANTARES assembler.

5.3 ANTARES Interface Binding

The last section of the machine description file describes the interface binding, which is done in one huge structure **`interfaceIR`** declared here. This structure defines the size of the used variables, fi how many bytes are integers and how many bytes are floats, and defines the alignment of these variables in memory. Alignment is of importance for memory-access and address calculation. We choose size and alignment for the used variables as defined in table 5-2.

The table shows that “short” is equivalent to “int” and long to “long long”, for the floating point variables there is only one format “float”, which is equivalent to “double” and “long double”. This has to be known with compiling C-programs, using these extended types, getting only the lowest precision results. High floating point resolution may be added later here, if really needed.

Blocks are aligned to word addresses to ease handling, even when they are composed of an even number of cells. The only compromise regarded alignment is done for char, which are aligned each. Char are very important for control and small messages and the waste of one byte, resulting from alignment, is not acceptable. So there is some hardware support in the instruction set to access and store bytes in an overall word oriented architecture. Details have to be defined with elaboration of the architecture.

Table 5-2: Defined Type Sizes and Alignments in Memory

Type	*Size*	*alignment*
Char	1	1
Short	2	2
Int	2	2
Long	4	4
Long long	4	4
Float	4	4
Double	4	4
Long double	4	4
Pointer T	2	2
Structure	0	2

In the interface there are further flags set, which should not discussed here any further. One flag, which is

linked to the type size definition as shown in table 5-2, defines, if constants are used in the instructions like "`ADI const, R1`" or handled like symbol-addresses, placed in the constant section. The flags are set for 32-bit constants like long and float, which are now placed in the constant section.

The rest of the interface definition is the listing of the target specific functions, which are used by the compiler and are defined in the C-code section. This part is not changed.

6 Verification and Performance

Setting up the compiler and doing all these complicated decisions on architecture needs of course verification and performance checking. This is done here with C-examples, compiled with `LCC_ant` and evaluated in comparison to a good hand coded program. The examples have to be chosen in that way that most of the features of the compiler is used. Further verification for more complicated examples have to be done later by compiling benchmark programs. Full checking and debugging is expected to take some month of time in addition. It will be done together with a fully tuned assembler and simulator, when these parts are qualified too. So the following examples are no more than a representative demonstration of performance than a proof.

Table 6-1: List of examples used for verification.

	Lines	*Ass. Stats.*	*Verified functionality*
Glbl_int.c	14	42	Global, constants, assignments, indexing
Locl_int.c	8	25	Access to locals, mixed usage
Arrays.c	12	30	Array usage, addressing,
Loops.c	7	38	Looping, control
Calcul.c	7	41	Calculation with int
Charact.c	16	23	Using characters
Struct.c	16	26	Using structures
Points.c	8	10	Using pointers
Longtp.c	11	106	Using long variables
Floats.c	9	92	Using floats
Functs.c	15	37	Using function calls
A_basic.c	161	1340	Using all features

Table 6-1 lists the examples and the verified functionality. We will only show here the first examples in detail and leave most of them to the annex. The example `a_basic.c` contains all C-operators and most of the C-constructs, this example can be used to test the compiler, but is has to mentioned that it is not complete and that there may be some C-constructs left which have to be proven.

6.1 Working with global variables

Global variable are declared before main( ). They are mapped to absolute addresses. Figure 6-1 shows the test program glbl_int.c and the result after compilation.

```
int iglb,jglb,kglb;
int i_val=1;
const con1=357;
main(void)
{
kglb=23;
kglb=2370;
iglb=i_val;
jglb=i_val+123;
iglb=i_val+con1;
iglb=i_val+kglb;
iglb++;
iglb-=jglb;
i_val=(iglb+i_val*kglb)/(kglb*i_val-jglb);
return 0;
}
```

```
;compiled with LCC (c) Christoph Fraser and David
Hanson
;ported to ANTARES architectur by DIRK JANSEN (c)
7/2000
;++++++++++++          ANTARES_16_Version   1.0
++++++++++
;              Version 4.6a

#ifndef MAINPROGRAM
#define MAINPROGRAM
#include <BIOS.asm>
#endif
CSEG    ORG    INIDATASEGMENT    ;   initialized  data
segment

i_val   DW 1
con1    DW 357
INIDATA_END:    ;       INIDATA ends

CSEG    ORG PROGRSTART ;  code segment

main:
;{
;kglb=23;
LDI 23,AB1
```

```
STA ABl,kglb
;kglb=2370;
LPR 2370/256
LDI 2370%256,ABl
STA ABl,kglb
;iglb=i_val;
LDA i_val,ABl
STA ABl,iglb
;jglb=i_val+123;
LDA i_val,ABl
ADI 123,ABl
STA ABl,jglb
;iglb=i_val+con1;
LDA i_val,ABl
LDA con1,ABh
ADD ABh,ABl
STA ABl,iglb
;iglb=i_val+kglb;
LDA i_val,ABl
LDA kglb,ABh
ADD ABh,ABl
STA ABl,iglb
;iglb++;
LEA iglb
LDX ABl
ADI 1,ABl
STX ABl
;iglb-=jglb;
LEA iglb
LDX ABl
LDA jglb,ABh
SUB ABh,ABl
STX ABl
;i_val=(iglb+i_val*kglb)/(kglb*i_val-jglb);
LEA i_val
LDX ABl
LDA kglb,ABh
LDA iglb,CDl
MOV ABl,CDh
MSY ABh,ABl
ADD CDh,CDl
MSY ABl,ABh
LDA jglb,ABl
SUB ABl,ABh
PSH CDl
PSH ABh
SWI SIGINTDIVVECTOR
STX EFh
;return 0;
CLR ABl
L1:
RET 2
CODE_END:       ;       CODE ends

DUNSEG ORG UNIDATASEGMENT ;    uninitialized data
segment

kglb    BYTE  DUP 2
jglb    BYTE  DUP 2
iglb    BYTE  DUP 2
UNIDATA_END:   ;       UNIDATA ends

; Thank you for compiling with LCC_ANTARES!
```

Figure 6-0-1: Results of the compilation of test program glb_int.c (above) into glb_int.asm.

The assembler code starts with a standard header, which is noted as comment and not further processed.

Next is the prolog with the `#defines` which allows to include a `BIOS.asm` precompiled operating system, only needed once. This file resolves the external macro definitions and contains the service routines for long and float processing as mentioned before.

Next is the memory reservation for the initialized variables, starting with the assembler directive

CSEG ORG INIDATASEGMENT

This is a directive to the assembler and loader, to place the following data declarations in this memory segment. The macro constant INIDATASEGMENT may be defined in the BIOS. CSEG is the name of a counter in the assembler, counting the addresses of all data put in this segment. The directive ORG sets the counter start to the defined value.

The program starts with

CSEG ORG PROGRAMSTART

Which defines the start address of the program, again defined in `BIOS.asm` and the label declaration of `main:`, which is the main program.

The LCC generates the assembler code with the related lines of the source code included, but out commented. This allows an easy control of the translation.

The first statement is storing a constant 23 to the global variable `kglb`. This is translated in

```
LDI 23, ABl
STA ABl, kglb
```

Using the load immediate instruction LDI and storing the value to the absolute address of `kglbl` with STA.

If the constant doesn't fit into the 8 bit of the LDI instruction, which is the case with the next statement, the prefix mechanism is used with

```
;kglb=2370;
LPR 2370/256
LDI 2370%256,ABl
STA ABl,kglb
```

showing the use of the prefix register.

The following statements are self-explaining. We have to mention the translation of

```
;iglb++;
LEA iglb
LDX ABl
ADI 1,ABl
STX ABl
```

The counter variable `iglbl` is addressed using the index register X. Loading first a pointer to `iglbl` to X, then using the pointer in X to load `iglbl`. Next step is incrementing the register and storing the value back to memory, again using the pointer in X. This index addressing is used for all statements, which show the same value on the left as on the right of the equality sign.

The test programs ends with a more complicated expression, showing the register allocation mechanism. There are still two registers (EFl, EFh) not used, so there is no need for register spilling to memory. The data flow shows very good performance of the register allocator. There is no improvement possible with hand coding in this example.

The integer division operation, which is not available in hardware, is coded as a SWI – call to a routine where the macro constant SIGINTVECTOR points. This vector is initialized by the BIOS and can be flexible handled as needed. The routine expects the arguments on the stack and delivers the result in register EFh.

The epilog contains the declaration of the uninitialized variables and the assembler file ends with a comment line.

6.2 Using local variables

Local variables are variables with a scope, constrained to the function block. They are declared in the function body. So variables declared in the function main( ) are local variables. The compiler maps local variables to stack resident variables. So they are addressed relative to the stack pointer. Because the stack pointer is manipulated in several ways, it is difficult to keep track with the offsets needed to access the locals, but this is included in the address calculation section of the function template. The figure 6-2 shows an example program, using local variables.

The main program starts with a SBI 18,S instruction, subtracting from the stack pointer S and allocating space of 18 bytes with this. These 18 bytes are 3*2 bytes for int, 2*4 bytes for long and 1 byte for char, which is round up to 2 for keeping the stack aligned, plus 2 byte for the return address.

Local variables have to be initialized during runtime, which is done with the first instructions in the code. Using immediate instructions initializes integers and characters, long and float -variables using constants, which are generated automatically by the compiler and which are loaded during this phase.

```
/* testprogram for local variables */

main(void)
{
int i=0,j=24,k=3045;
long lgv,lgm;
char c1='a';
i=j+k;
j=i+k;
lgm=lgv;
return 0;
}
```

```
;compiled with LCC (c) Christoph Fraser and David
Hanson
;ported to ANTARES architectur by DIRK JANSEN (c)
7/2000
;++++++++++++              ANTARES_16_Version    1.0
++++++++++
;                  Version 4.6a

#ifndef MAINPROGRAM
#define MAINPROGRAM
#include <BIOS.asm>
#endif
CSEG    ORG PROGRSTART ;   code segment

main:
SBI 18,S
;{
;int i=0,j=24,k=3045;
CLR ABl
STS ABl,20-2
LDI 24,ABl
STS ABl,20-4
LPR 3045/256
LDI 3045%256,ABl
STS ABl,20-6
;char c1='a';
LDI 97,ABl
SBS ABl,20-17
;i=j+k;
LDS 20-4,ABl
LDS 20-6,ABh
ADD ABh,ABl
STS ABl,20-2
;j=i+k;
LDS 20-2,ABl
LDS 20-6,ABh
ADD ABh,ABl
STS ABl,20-4
;lgm=lgv;
LDS 20-12,ABl
LDS 20-12+2,ABh
STS ABl,20-16
STS ABh,20-16+2
;return 0;
CLR ABl
L1:
ADI 18,S
RET 2
CODE_END:      ;       CODE ends

; Thank you for compiling with LCC_ANTARES!
```

Figure 6-0-2: Test program using local variables

All accesses to local variables are done with the stack relative load instruction LDS and store instruction STS. The offset to S is calculated using the frame size, the relative displacement to the frame and eventually a further offset for long variables.

At the end of the routine, the stack register is corrected by adding the frame size, so the register S points to the return address of the function, stored in the stack. The return RET 2 corrects again the stack pointer, in this case by skipping the return address position.

The stack pointer must be on the same position after processing of the function than before. This allows an unlimited nesting of routines with local memory freed after leaving the subroutine.

When the compiler need additional local storage for register spilling or intermediate storage, this is done by automatically allocating memory on the stack. So the stack deepness may be higher than seen from the C-program source code and it is important that this mechanism works fully automatic.

Stack offsets larger than 127 after summation have to be handled by the assembler, inserting LPR instructions. This is seldom needed with good program stile.

Initializing local variables have to be used with care because it has to be done at run time, slows down processing performance and eats up the space with global constants used.

6.3 Using arrays and loops

The test programs for arrays and loops are in the annex. We will describe only the new constructs. Array addressing is done with an offset, added to the address of the first array element. The compiler uses the size – information to calculate the exact address-offset.

Index calculation is done using the X register. Size of the variables is used to calculate offset accordingly. Because all references made to memory are calculated in bytes, references to integers have to be multiplying by two, references to long variables by four. Instead of multiplication shift instructions are used.

The loop compiles very nicely and effective. All kind of loop constructs is implemented and works fine. The programs show very good performance with no further improvements possible.

6.4 Compiling function calls

The program `functs.c` and its translation `functs.asm` can be found in the annex. This program demonstrates how arguments are passed to the function body. As has been mentioned before, with a function call local variables are set up on the stack by adding a displacement to the stack pointer S, spanning up a frame. Arguments to the function are pushed on the stack before the function call, decreasing the stack pointer register with the amount of bytes of the arguments plus the size for the return address, which is two bytes.

With leaving the function, all this stack manipulation has to be removed. The stack pointer must have exactly the same content as it was before entering the function. This is done in two steps: first adding the frame size to the stack register, freeing the local variables, second increasing the stack pointer in the return statement, freeing the argument space and the space for the return address.

Access to the arguments is similar to access to local variables using stack relative addressing. Because these arguments are pushed on the stack before the call, they are placed above the return address. Address calculation is more complicated, because frame size as well as the relative position of the arguments has to be regarded. Assembler output for offset calculation is a mixture of these references, which has to be resolved by the assembler. The assembler has also to take care for introducing the prefix mechanism, if the offset exceeds 256 bytes. So for avoiding the prefix penalty, there should be only few arguments used with a function call, which is mostly the case.

6.5 Performance of long and float processing

The example programs `longtp.c` and `floats.c` are in the annex. Using two statements one after another to transport long variables over the 16-bit busses does long processing. Addition and subtraction of long variables is done without subroutine calls, the same is done with comparisons, although there are several instructions needed. Multiplication and division calls subroutines from the BIOS. This is of course not very effective. Because the "question mark mechanism" removes only the first MOV R, R – instruction, there are few superfluous MOV instructions, which have to be removed now by the assembler.

Floating point processing is even worse. Every manipulation calls a BIOS routine, which may contain several instructions.

Because there are only three double registers used for long- and float- processing, a more complicated expression will trigger register spilling as can be seen in `floats.asm`. Spilling is done by storing intermediate results in local variables, which are generated automatically by the compiler. It has to be discussed, if it is really worth the effort in a larger design to expand all the registers to 32 bit. Register spilling slows down only a little bit and is only used with complicated expressions.

Floating point processing and long processing works, but is not efficient, as we have known before. For applications with low requirements concerning mathematics calculation it will be sufficient.

Scaling up the architecture to 32 bit will improve floating point-performance significantly. From the compiler side, doubling the size of the ALU and the busses will be the most effective, allowing moving and manipulation of long variables with only one instruction. Adding further a barrel shifter to the ALU and sizing the multiplication device to 16 x 16 bit will further improve efficiency a lot. These measures will perhaps double the size of the ALU with an overall increase in area of the core of about 25%.

Doubling the register size would add much more area to the core, so keeping the registers to 16-bit and connecting them to double registers for 32 bit processing is a good compromise, as long as the addressing space is kept to 64 Kbytes.

6.6 Combined types and operations

The file `a_basic.c` is a test program and summarizes all basic operations. The file compiles to about 1340 statements in assembler. Although it doesn't use all possible constructs, is can be used as a general test file for testing the compiler.

6.7 Overall Performance Assessment

Compiler efficiency of integer processing using the ANTARES architecture is very good and can't be further optimized by hand, looking on the compilation results. Register usage is near optimal, spilling is done only with long and float-types with complicated expressions, so the number of registers is sufficient. Further optimization may be done with combining more than one C-code expression, removing some memory accesses with a more powerful optimization strategy. For the intended application area this is not needed, until the full potential of the LCC inherent mechanisms are used. Some are still switched of, so register variables are mapped to local memory at the moment and some other implicit optimization, which should not further discussed here.

Float- and long processing performance is poor in velocity and code size, directly linked to the 16-bit structure and can only be improved by scaling the architecture. But it's still much better than similar codes for many legacy cores with 8 bit and also 16-bit architecture. At the moment this is an impression taken from experience, detailed comparison has to be done later.

One goal of the ANTARES development is to keep the memory requirement as low as possible. The SWI mechanism, allowing fast access to the BIOS routines, supports this and keeps the main application program free from adding big libraries. So we are able to use the comfort of high level language programming without paying the penalty of adding large libraries, blowing up code size. Care has to be taken when translating the ANSI-C libraries, some may have to be rewritten to ANTARES. This task has to be done in combination with the development of the operation system BIOS, optimizing performance and code size at the same time. Because this task doesn't influence the architecture any more, we will not further discuss this here.

7 Assembler

The compiler produces code in the assembler mnemonic language, which has to be processed further to executable object code. To get an executable, three tasks have to be done:

- Assembling the mnemonics,
- Resolving all references to memory addresses and external function calls,
- Linking the codes to an executable, which can be loaded by a loader.

These tasks are not yet all done and there have to be some more details decided, where certain information on memory mapping, constants loading etc is specified.

At the moment, we adapted an existing retargetable assembler called "CRASH", **casm.exe** made by Daniel Vogel [DaVo], which have been developed in Offenburg and is used in several applications with success. CRASH is retargetable with a configuration file `ANTARES.bef`, defining the coding of each instruction. The file is included in the annex. For documentation of CRASH and the used syntax, see the related website [DaVo].

CRASH produces directly loadable files in INTEL-Hex format, used broadly for this purpose. For human control, a list-file is produced, showing all codes and addresses as well as the compiled symbol table.

Most of the requirements of ANTARES to the assembler can be done by CRASH without modification. But there have to be some functions added:

- CRASH cannot be configured for the prefix mechanism for addresses yet,
- Handling of multiple references, coming from separate compilation of source files, has to be improved.
- Mapping of different memory segments is not yet solved satisfyingly.

CRASH was mainly invented for smaller programs. Bigger codes coming from high-level language programming rise new challenges and require further development.

The examples, shown here, are all avoiding the above constraints, so they process nicely. All examples from table 6-1 are usable. In the annex there is the biggest file `a_basic.c` processed to the `a_basic.lst`, which contains the rudimentary `BIOS.asm` too. There may be still some bugs, a detailed control of the object code is only possible with a simulator.

CRASH is integrated into the Windows–IDE together with a multi document editor and the simulator environment, shown later.

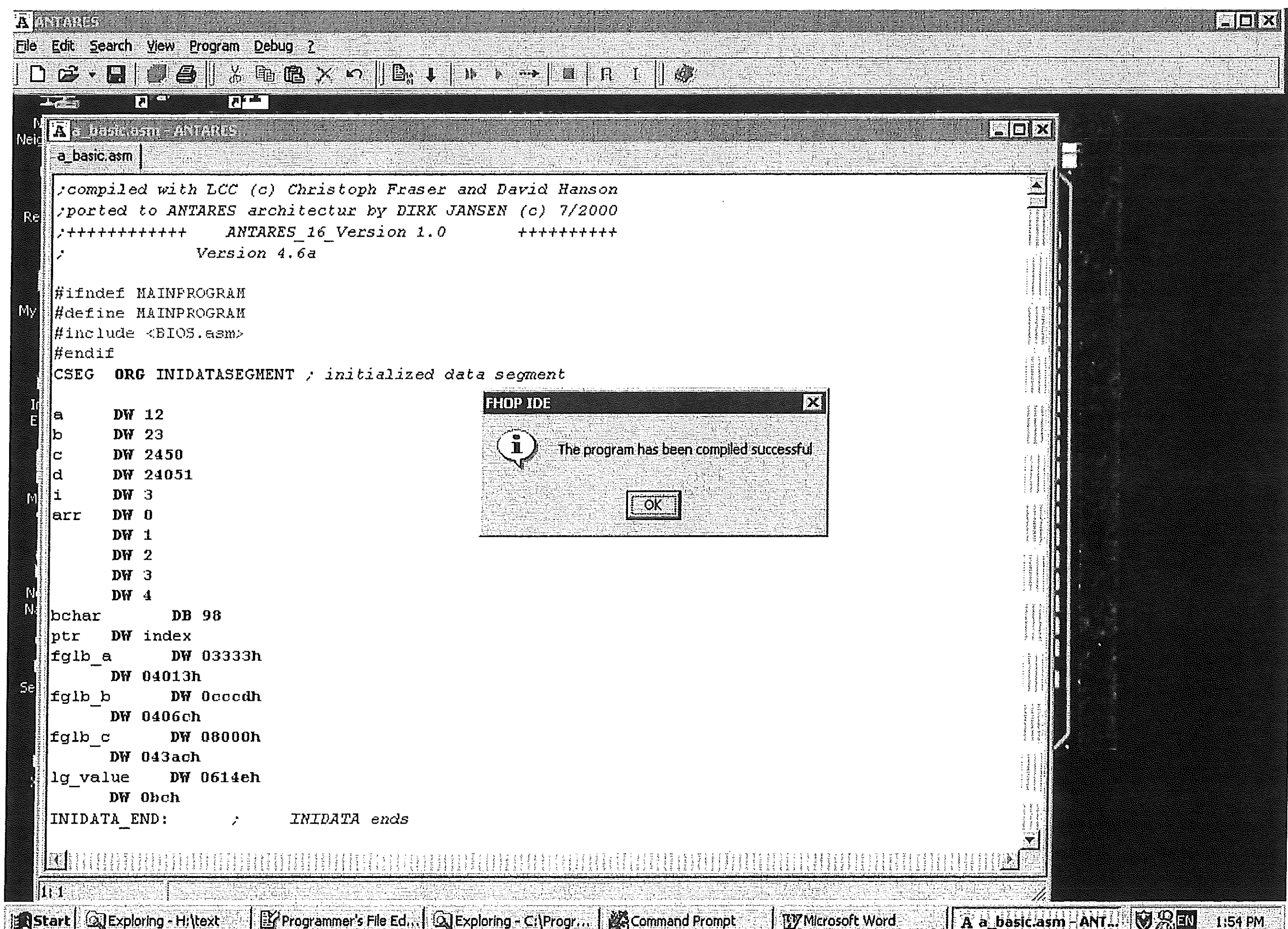

Figure 8-1: ANTARES – Integrated Development Environment (IDE), Multi Document Editor and Assembler.

8 Simulator

Performance verification as well as programming of applications needs an instruction-exact simulator. ANTARES simulator is integrated in an integrated development environment (IDE) with multi document editor, C-compiler, assembler and simulator tightly coupled. Alternative solutions may be based on [LEUSIM], but the IDE already exists for the predecessor FHOP and is only rewritten in part. The IDE is a Windows-application, running on PCs with NT operating systems. Figure 8-1 shows the IDE with code loaded and figure 8-2 in simulation mode.

The IDE simulator loads the hex-file, produced by the assembler to the specified RAM or ROM space with additional options concerning RAM and ROM-mapping in the address-space.

Program execution starts with "reset". There are 3 modes applicable:

- **Step wise processing of code** which needs an interactive mouse click for each instruction processed,
- **Slow automatic processing** of code with configurable delay between instruction execution, allowing a time zoomed program execution,
- **Run mode** with execution of code as fast as possible.

Pressing the "stop" - button can stop a running program. Breakpoints may be placed into the code before start of execution. The usual functions of a debugger are all present.

There are windows to show the contents of the registers, the flags and memory. Simulation of hardware interrupt is possible by configuring a button or with the OLE-functionality, described later.

Ports are included allowing program controlled input

Figure 8-2: ANTARES-Simulator with trace-window (left), source-window (middle), port-window (half-right), stack-window (right), register-window (down-left) and memory-window (down-middle).

and output. Periphery modules are simulated by separate windows applications, coupled to the simulator via an OLE-interface. OLE allows communication of data and events via a standardized protocol, delivering data to the ports and taking data from the ports.

Examples for such "soft components" are the timer-component figure 8-3, the serial-IO-component, the interrupt controller-component etc., units which are all existing in hardware and which are used together with the core. Newly developed component can be added as needed with limited effort in programming.

The simulator is taken from an existing IDE developed for FHOP [FHPSIM] with new definition of instruction processing, now adapted to ANTARES.

For simulation of future large chips it is intended to set up a SpecC – simulation model [GAJ] of the core to further facilitate development in an overall C-environment. Using SpecC the core can be capsulated with true timing specified on the bus, allowing socketizing several components in an IP-oriented integration manner.

9 Conclusion

The concurrent design of architecture and compiler shows clearly the interdependence of instruction set definition and coding, hardware architecture and processing performance.

Taking the C-code constructs and rebuilds them as instructions on RISC architecture, very good performance for the intended application area of SOCs is achieved. Real ANSI-C- styles C-programs are demonstrated on a small chip-size model. This will allow huge gains in programming efficiency.

The design shows very good performance for integer processing and control applications, the performance of long- and float- processing is poor, as expected and decided by using a 16-bit architecture. The constraints putted on the architecture are linked to the small size of the registers of 16 bit, taking only half of the operand. Some improvement is possible by linking two registers to one 32-bit register and extending the ALU to 32-bit width. This may be done in a larger version, if mathematical calculation gains more attention.

The design uses a special, Huffman-style coding of the

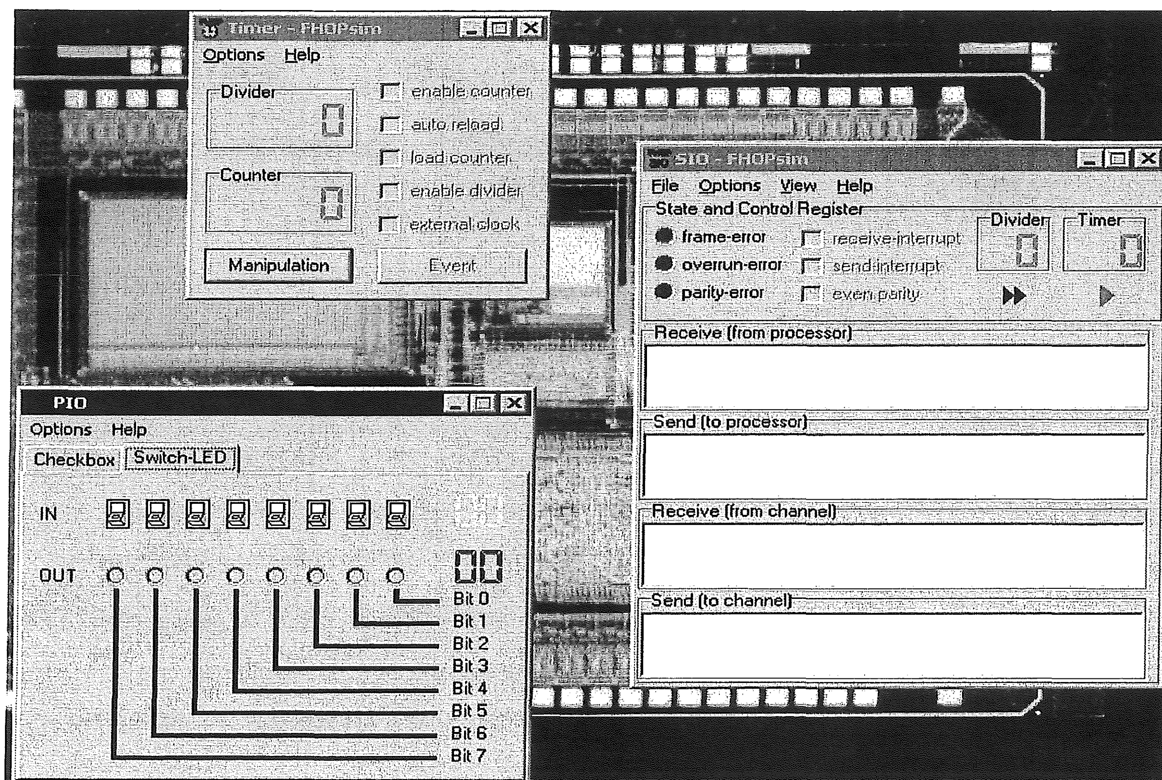

Figure 8-3: Soft-Components for Simulation of Peripheral Units with the Simulator: Programmable Interface Unit PIO (left, Timer Unit (above), Serial Input Output Unit SIO (right).

instructions, allowing unifying memory access to data or instructions both as 16 bit. This allows a simple pipeline scheme and fast processing as well as an easy bus interface to memory. The limitation of instruction size to 16 bits leads to the creation of a prefix mechanism for access to large constants or addresses, which has to be managed by the compiler. This again demonstrates the strong interaction of compiler, assembler and architecture.

The compiler is able to compile full ANSI-C-standard as wanted, demonstrated by several examples. Dynamic storage of local variables is managed with an effective stack-concept, global variables are placed in the low RAM for easy access. For further performance evaluation, benchmark programs have to be compiled and simulated on the architecture. This will allow a better comparison to existing legacy cores and similar development nowadays. Before this, BIOS has to be written and the libraries compiled, all influencing the overall performance.

ANTARES will run with an intended above 100 MIPS performance and an estimated chip area of below 2 mm² in a 0.5u CMOS technology and may be mapped to every new technology without redesign. It will also be suited to FPGA applications, because it uses only gates and flops. With this data it will be one of the smallest cores with this performance.

The design will now be further detailed in a hardware description language (VHDL), synthesized to a CMOS standard library and verified on silicon by an example application chip. This will finally show, if the goals concerning power consumption, core area, and processing speed can be fulfilled in the expected way. So ANTARES will be the next generation core for all system chips designed in Offenburg and perhaps, somewhere else.

10 References

[FRAHAN] *C. W. Fraser, D. R. Hanson*: "A retargetable C Compiler: Design and Implementation.. "Addison-Wesley Publishing Comp., Redwood City, 1995.

[LBURG]: *C.W. Fraser, D.R. Hanson and T.A. Proebsting*: "Engineering a Simple, Efficient Code Generator Generator". ACM Letters on Programming Languages and Systems 1, 3 (1992)

[GAJ] *Daniel D. Gajski, et.alt.*: "SpecC: Specification Language and Methodology". Kluwer Academic Publishers, 2000.

[ACK] *Ernest C. Ackermann*: "The essentials of C Programming Language". Research and Education Association, 1998.

[INT51] http://developer.intel.com/design/: Datasheet 80C51 Series Processor

[INT96]] http://developer.intel.com/design/: Datasheet MCS 96 Series Processor

[INT86] http://developer.intel.com/design/: Datasheet 80x86 Series Processors.

[FHOP] *D. Jansen et.alt.*: "Application specific System Engineering with the embedded Microprocessor Kernel FHOP." European Microelectronics Conference, EMAC 97, Barcelona 1997.

[FHPSIM] website: http://www.asic.fh-offenburg.de/

[DAVO] *Daniel Vogel*: CRASH-Assembler Homepage on http://www.asic.fh-offenburg.de/Homepages/daniel-vogel.html.

[LEURET] *Rainer Leupers*: "Retargetable Code Generation for Digital Signal Processors. Kluwer Academic Publishers, 1997

[LEUSIM] *Rainer Leupers*: "Generation of Interpretive and Compiled Instruction Set Simulator." ASP-DAC, Hong Kong 1999.

[MARW] *Peter Marwedel*: "Code Generation for Embedded Processors: An Introduction." Kluwer Academic Publishers, 1995.

Acknowledgement

This work has been done at the Center for Embedded Computing Systems, University of California, Irvine. I have to thank Prof. Dr. Gajski and all the colleagues in CECS for the opportunity to work on this subject and for many interesting discussions.