Porting GNU Compiler Collection to Eco32

Jens Mehler - June 23, 2014

Contents

Chapter 1

Abstract

This document deals with the theoretical and practical background of the implementation for the EGG-Project[1]. The idea is to enable the participants of the Eco32-Project to port another operating system to run on the Eco32. The work itself is split into two projects that form a complete compiler tool-chain. The final result will be a basic cross-compiler for the Eco32. It can easily be advanced to fit additional needs.

GCC-Backend

The GCC[2] Backend enables the GCC to translates C-Source-Code into the Eco32[3] assembler instructions. To obtain this goal a complete backend, consisting of a machine description as well as some additional macros have to be implemented. The corresponding files specify the present registers, their use and their functions. Some of the registers of the Eco32 processor have special functions like holding function arguments or holding the stack-pointer. Those information have to be made know to the new back-end for GCC to actually emit correct assembler code. Not all of the registers can be used to store variables in a save way over a function-call. Those registers are not call-save and have to be marked as such. The same goes for registers that can actually save variables over a function call. That specification have to be known to the compiler because it has to save those variables onto the stack for later. Furthermore the compiler suite would be able to translate any other language than C[4] if the necessary specification is given. Nearly the complete instruction-set of the Eco32 is implemented and can be easily extended or modified by anybody who wants to by editing the file eco32.md. This LISP[5]-like language tells the Gnu Compiler Collection about the Eco32 instruction-set and how to match it against its own RTL[6] representation of the source code. In addition the way of addressing the memory has to be implemented for the load and store instructions.

[1] Eco32 Gnu C-Generator
[2] Gnu Compiler Collection
[3] Educational 32 bit computer
[4] only C is currently supported
[5] LISt Processing
[6] Register Transfer Language

Chapter 2

Introduction

This document deals with the Eco32-GCC-Backend and how it can be build and better understood. Most of the implementation is sufficiently documented in the code. This guide should shine some light on how to actually write a port.
The intent of this document is to bring my work out into the world and let people have fun reading it.

2.1 Motivation

The current compiler, and its tool-chain, available for theEco32 and the Eco32-OS[1] are based on an ANSI-C Compiler. The LCC is an easy retargetable compiler in combination with its tool-chain written by Prof. Dr. Geisse. These tools allowed the porting of Unix v7 for the Eco32. For the first steps into a completely new environment the Little C Compiler is a great middle ground. Since the Eco32 teams wants a more flexible and more adjustable C Compiler the EGG-Project was born.

The most commonly known compiler in the posix world is the Gnu Compiler Collection. This compiler-suite is able to translate any programming language to any processors assembly instructions. That is true for the case where there is a front-end, for the language, and a back-end, for the processor, present.

The compiler of choice, the Gnu Compiler Collection, doesn't know the Eco32 at the moment. Therefore it needs a new back-end, that will emit the assembler instructions for the Eco32. In addition there was also the question whether the old tool-chain would still work or not. The tool-chain for the GCC will be build with the Gnu Binutils[2].

2.2 The EGG project

The aim of the EGG-Project[3] is to provide a working Cross-Compiler for the Eco32 processor. In combination with a port for the latest Gnu Binutils it features a full compiler-suite.

To be more specific this project centres on implementing a new back-end for the Gnu Compiler Collection to generate a C-Compiler for the Eco32 processor designed by Hellwig Geisse. In addition the Gnu Binutils are ported to generate the executables for the CPU[4].

To make this huge project more understandable it is split into two smaller parts. Those two parts

[1] based on UNIXv7
[2] The Gnu tool-chain including an assembler, linker and binary file descriptor.
[3] subject to change
[4] This is still under heavy development

cover the implementation of the GCC-back-end, including background information about how the Gnu Compiler Collection works and how to port the Gnu-Binutils to support the Eco32.

2.3 The Gnu Compiler Collection - GCC

The GCC is more than just a C/C++ compiler. Divided into three sections, the Front-, Middle- and Back-end, it can generate any compiler for any CPU, as long as both the programming language and the CPU are properly implemented. The first part of the EGG-Project is a compatible and correctly implemented GCC-Backend to make the Eco32 known to the GCC.
A closer look on how the GCC works can be found in chapter 5. Any important explanation on terms can also be found in chapter 5
The explanation about the implementation can be found in chapter 4.

Chapter 3

Eco32

The target processor is a general-purpose 32 bit RISC processor, designed by Prof. Dr. Hellwig Geisse at the University of Applied Science in Giessen. It is used for educational purpose therefore its name. Eco32 stands for "Educational 32 bit computer".

The complete documentation about this processor can be found in the Eco32 User Manual[2].
For the purpose of this work I want to draw some attention to the parts that are essential for this project. Most of the sections below can be found in this or nearly the same way in the Eco32 User Manual, since it is a reliable source there is no reason to rephrase it too much.

3.1 General

The processor has a set of 32 registers, each register is 32 bits wide. The instruction set, currently without instructions for floating-point operations, contains 61 instructions. The sequential processing of instruction is designed to get complex constructs by combining multiple instructions. Only the most basic arithmetical and logical as well as instructions for loading from and storing into memory are present. Branches and Jumps each have their own instructions and data type conversion can also be handled.

The data types processed by the Eco32 are starting with the most significant bit. The following data types can be processed:

- Byte: 8 bit unit
- Half-Word: 16 bit unit
- Word: 32 bit unit

Memory addressing is achieved with a 32 bit unsigned value for RAM, ROM or for a peripheral device. The complete physical memory is in the range from 0x000000 through 0x2FFFFFFF. The Eco32 supports one way of addressing its memory. Memory is always addressed the following way: *REG+OFFSET*. While *0+OFFSET* is the direct way of addressing memory one can always use another register to store a base address. That way is called indirect addressing.[1]

"The Program Counter (PC) is a virtual register that contains the address of the next instruction to be executed"[2]. Instructions can also modify the PC thus the PC will be set to the instruction which has to be executed next. Normally the Program Counter would get incremented by 4 bytes (32 bits). That moves the PC to the next instruction because one instruction is always 32 bit (4 bytes) long.

[1] The compiler can and will use both ways to address memory in the stack. See chapter 4
[2] Taken from Eco32 User Manual[2] Page 6

3.2 Hardware Register

The Eco32 contains a set of two different types of register.

3.2.1 General purpose registers

To compute the result of an instruction the processor has to store the data in its register. Still not all registers are available to store data because they are already predefined by the hardware layout. This has also be taken into account for the implementation in the compiler back-end.

Table 3.1: Eco32 general purpose register

Name	Description
$0	Tied to Zero by Hardware
$1 .. $29	Register without special purpose
$30	Interrupt Return Address, not used by Compiler
$31	Function Return Address

The data for the instructions is stored in one of the register from 0 through 29. Any result of an instruction is also stored in one of those registers. Aforementioned registers can also store a base address for accessing memory that can be used in any load and store instruction.

The registers $0 through $29 will be used in the compiler. More information about that later.

3.2.2 Special purpose register

The special purpose registers of the processor are accessed with two instructions *MVFS* and *MVTS*. The following special purpose registers are present.

Table 3.2: Eco32 special purpose register

Index	Name
0	PSW
1	TLB Index
2	TLB Entry High
3	TLB Entry Low
4	TLB Bad Address

The first register is the processor status word. It controls the parameter for the processor. While the remaining registers take care of communication with the Memory Management Unit. Those registers can only be accessed while the CPU runs in Kernel Mode. For a further explanation about the MMU and the privileged modes please read the Eco32 User Manual[2].

3.3 Addressing Modes

Addressing memory is handled in two ways, both following the same pattern: *REGISTER+OFFSET*. The two separate ways are called "Direct Addressing" and "Indirect Addressing"

The direct way of addressing memory in the RAM is to use the register $0 in addition with an offset.

The indirect way of addressing memory in the RAM is to use any of the non fixed general purpose registers and store a base address in it. Using these registers in addition with an offset is called indirect addressing.

3.4 Load and Store architecture

To handle access to memory the processor uses two specific instructions. If a value from the memory should be written into a register it has to be loaded from there. Therefore the instruction load[1] exists.

Every time a value should be written into the memory the instruction store is used[2]

Both of those instructions come in variants to either store a word, a half-word or a byte.

3.5 Instruction Set

A set of 61 instructions are currently present for the Eco32. No floating-point operations are present at the moment. The instructions can be categorised as followed:

1. Computation Instructions
    - addition
    - subtraction
    - multiplication
    - division
    - logical operations
    - logical shift
    - arithmetical shifts
2. Control Flow Instructions
    - Branches
    - Jumps
3. Load and Store Instructions
    - load from memory
    - store in memory
4. System Instructions
    - trap instruction
    - modify special register

Every computation instruction has two variants. Those two variants deal either with an all register based instruction or an instruction that contains an immediate value[3]. For a better understanding, look at the following example.

The instruction add, takes three operands: a target register, a first source register and a second source register. This instruction adds up the two source register and stores the result in the target register.

```
1 add $2,$3,$4
```

The immediate variant also takes three operands with the difference, that the third is a numeric value, fitting into 16 bit.

[1] Actually there are more specific instructions. They will be explained in chapter 4

[2] it's the same as with load

[3] A number directly contained in the instruction itself

```
addi $2,$3,0xFFFF
```

The Eco32 can also handle immediate values with more than 16 bit. This case will briefly be discussed in Chapter 4 Section 4.2.1.

Chapter 4

Eco32 ABI - Application Binary Interface

The following section gives an overview on theoretical part of the implementation.

This section was inspired by the "MIPS RISC Prozessor Supplement 3rd Edition"[4]. Before getting started in porting the programs some constraints need to be put in place. Therefore this chapter will explain what the constraints of this project are.

4.1 Register Usage

The original register-usage still resides in the "Eco32 User Manual"[2]. Since it was necessary to change this layout in favor of this project I proceeded with the following specification.

Table 4.1: EGG Register usage

Name	Alias	Description
$0	zero	Tied to Zero by Hardware
$1	ast	Assembler temporary Register, not used by Compiler
$2 .. $3	frv_n	Function Return Value
$4 .. $7	fa_n	The first four Function Arguments
$8 .. $15	tmp_n	Used for temporary Results, Caller Save
$16 .. $23	loc_n	Used for local Variables, Callee Save
$24 .. $26	os_n	Reserved for OS, not used by Compiler
$27	ap	Argument Pointer
$28	fp	Frame Pointer
$29	sp	Stack Pointer
$30	ih	Interrupt Return Address, not used by Compiler
$31	ret	Function Return Address

n: is 0, 1 to n, depends on the number register available

The new register-layout allows the usage of a frame- and argument pointer. Those two pointers will come in handy when porting the GCC.

4.2 Instruction Set Architecture

The Eco32 Instruction Set can be classified with different formats, these formats are important for the assembler and will be used later on.

The original toolchain honored table 4.2 in all aspects. For the Binutils-Port the RRX and RRY instructions have been dropped.

Table 4.2: Eco32 Instruction Formats

Format	Description
N	no operands
RH	one register and the lower 16 bits of a word
RHH	one register and the upper 16 bits of a word
RRH	two registers and a zero-extended halfword
RRS	two registers and a sign-extended halfword
RRR	three registers
RRB	two registers and a sign-extended 16 bit offset
J	no registers and a sign-extended 26 bit offset
JR	one register
RRX	three registers, or two registers and a zero-extended halfword
RRY	three registers, or two registers and a sign-extended halfword

Table 4.3 still shows RRX and RRY formatted instructions. Please note that RRX is a bundle of the corresponding RRR and RRH instructions - the sames goes for RRY. It's a bundle of corresponding RRR and RRS instructions. In short op2 can either be a register or a halfword (honoring the constraints given by table 4.2).

Dropping RRX and RRY results in a larger instruction table for the assembler but it results in easier coding.

4.2.1 Assembler Instruction Expansion

There is a special case to this instruction set. Since the computation instruction that handle immediate values can only hold up to 16 bit there has to be a way to handle large immediate values. The following example will explain the expansion of an assembler instruction that deals with an immediate value greater 16 bit.
Looking at the following assembler instruction that wants to add $3 and 0x12345 and store the result in $2

```
addi $2,$3,0x12345
```

and the definition of addi in the ECO32-User Manual
"**ADDI**: The ADDI instruction computes the sum of a 32-bit register operand and a sign-extended 16-bit immediate operand, truncated to 32 bits."

0x12345 doesn't fit into a 16-bit immediate operand. But this can easily be worked around by using the instruction LDHI.

From the ECO32-User Manual
"**LDHI**: The LDHI instruction is used to generate large constants. The upper 16 bits of the result are taken from the 16-bit immediate operand. The lower 16 bits of the result are 0."

```
ldhi $1,0x00010000
```

LDHI loads the high-part into $1 - the assembly register. Now just put the rest of the immediate value into this register without changing the high-part. The instruction ori comes in handy here

From the ECO32-User Manual
"**ORI**: The ORI instruction computes the bitwise OR of a 32-bit register operand and a zero-extended 16-bit immediate operand."

```
ori $1,$1,0x2345
```

After this instruction the full constant resides in $ast and the next instruction can use this as the second source register.

Since we wanted to add an immediate value to a register; we now have to use add.

```
add $3,$2,$1
```

This simple pattern can be used anytime an immediate value bigger 16 bits comes up in any immediate computation instruction. The Gnu Assembler will take care of this whenever it is necessary.

4.2.2 Assembler Pseudo Instruction

All assemblers support pseudo instructions which are used to controll the haviour of the assembler. The most interesting and noteworthy pseudo instructions are:

```
.align n - Padd addr n.
.p2align n - Padd addr to n^2
```

In case the current addr is already at the right alignment nothing is done. This will insert some nops to the output file. This is usefull for function alignment.

Additionally the Eco32 supports its own assembler pseudo-operations. Currently there are only two of them.

```
.syn - Enables synthesized instructions.
.nosyn - Disables synthesized instructions.
```

Taken from [2, Eco32 User Manual page 61]: These control parameters may be set up to allow synthesized instructions. These look like single instructions in the input file, but are actually assembled as short instruction sequences. Synthesized instructions exist purely for convenience when writing assembler code manually.

4.3 Addressing Modes

The Eco32 supports exactly one kind of addressing mode: REGNO+OFFSET. REGNO is either the Stack Pointer, the Frame Pointer or $0. The special case of $0 + offset is that this is direct addressing while addressing memory with an offset from the Stack- or the Frame-Pointer is called indirect addressing. The addressing modes and therefore the kind of addressing the Eco32 uses is important for loading from and storing into memory.

The instructions *stw* and *ldw*, with their different flavors take care of moving data in and out of memory.

```
stw $src,$base,offset
sth $src,$base,offset
stb $src,$base,offset
```

These three instructions store a value from register *$src* to the address specified with *$base+offset*.

```
ldw $dest,$base,offset
ldh $dest,$base,offset
ldhu $dest,$base,offset
ldb $dest,$base,offset
ldbu $dest,$base,offset
```

These five instructions load a value from memory *$base+offset* into register *$dest*
The offset for all instructions is a 16 bit immediate value.

4.4 Stack Layout

When defining the stack layout several things have to be taken into account. The sections of the stack need to be placed in order to address it only with the stack pointer[1].
The stack holds the following sections in corresponding order.

1. local and temporary variables: allocated in function call prologue
2. Register Save Area for ret, sp, tmp_n, loc_n : allocated in function call prologue
3. dynamic sized space for alloca(): allocated in function call
4. Argument Block: allocated in function call prologue

All sections can be addressed with a positive offset to the stackpointer. To calculate the right section the actual frame-size is needed, which can be calculated at compile time.
All offsets are positive because the stack grows downward and the frame grows upward. The stack-pointer points to the top of the stack which is a lower address than the element to be addressed.

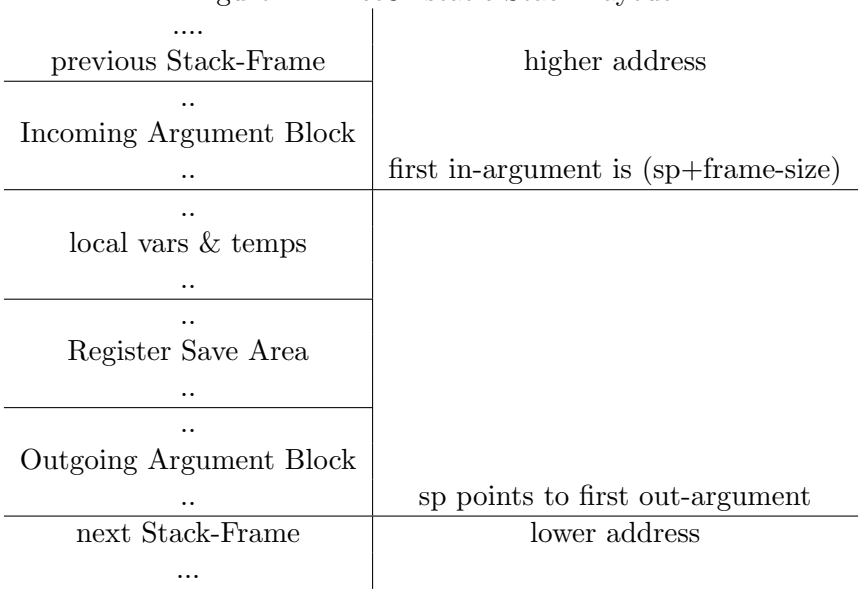
A difference in the layout appears if we look at dynamic and static memory allocation. Static memory allocation can be handled by the compiler and the right amount of size can be computed at compile-time. The dynamic-memory allocation on the stack can't be computed at compile-time but there is a way to handle that.

[1] Special cases excluded

4.4.1 Static Stack Frame Layout

View from current Stack-Frame for static size:

Figure 4.1: Eco32 static Stack Layout



Bottom of the picture is the top of the stack.

4.4.2 Dynamic Stack Frame Layout

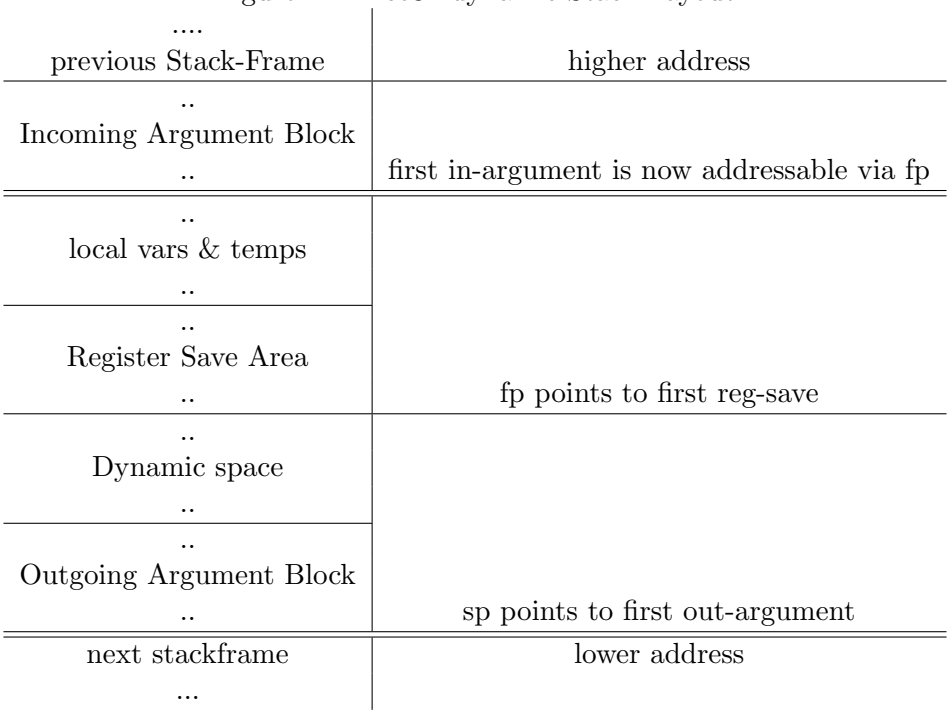
If alloca(size_t size)[1] is called and space of unknown size is allocated on the stack, further addressing won't be possible without the help of a framepointer. A framepointer will be created for a function that needs dynamic allocation of space on its stack. It will be held in a callee save register to be preserved during function calls.
The following steps are required to allocate dynamic sized space on stack:

1. Allocate size for local & temp vars, Register-Save-Area and Outgoing Arguments; Create a framepointer and set stackpointer to the top of the stack. Set the framepointer to first reg-save, from now on the framepointer will work like the stackpointer for static data on the stack except for the outgoing arguments. Those will still be addressed by the stackpointer with an offset.

2. Move down the stackpointer to enlarge the frame to the needed size[2]. Thus the outgoing arguments will move down, in front of the dynamic allocated space. The stackpointer will now still point to the first outgoing Argument on the stack. While the framepointer still points to the first reg-save position.
   In short terms:
   The stackpointer will address the outgoing arguments and the dynamic allocated area while the framepointer takes care of addressing reg-save- and local-vars and temp area.

For a better overview see the picture below. View from current stackframe with dynamic size:

Figure 4.2: Eco32 dynamic Stack Layout



This is currently not implemented. If it will be implemented this way has to be tested in the future.

[1] This function allocates *size* bytes on the stack - since the correct size if not know at compile time the compiler can't assume anything and has to create a frame pointer.

[2] This happens during runtime of the program.

4.4.3 Local- and temporary Variables

The lowest address of this block is the first variable. This area holds all local and temporary variables declared in C-Source.

4.4.4 Register Save Area

The Register-Save-Area is divided into three different blocks:

1. Global-Register-Save-Area - sp,fp,ret
2. Callee-Register-Save-Area - $8-$15
3. Caller-Register-Save-Area - $16-$23

The allocation is starting from lowest address in the following order:
Global-, Caller-, Callee-Registers.
The Callee- and Caller-Registers will be saved in the order of the register numbers from low to high.

This leads us to the following table:

...	
Callee-Saves	eight slots max
Caller-Saves	eight slots max
Global-Saves	three slots fix
...	

4.4.5 Outgoing Argument Convention

The argument at the lowest address is the first outgoing argument. The first four arguments passed in registers have their respective slots on the stack! They are not actually there but this makes calculating the number of out-outgoing arguments a bit easier.

Argument-n	last argument
...	..
Argument-3	third argument
Argument-2	second argument
Argument-1	first argument

4.4.6 Elimination of Argument- and Frame Pointer

The stack of the Eco32 is designed to only be addressed by the stackpointer. The GCC still wants to know which registers hold the argument- and framepointer. Those two pointers would normally occupy two register that can be used to hold other and more valuable data.
The difference between the argument- and frame- pointer is that the compiler can eliminate the first one without any concern. The framepointer may be used in special cases as described above and in some functions where the compiler can't make any sense of the program without the frame pointer. In both cases the framepointer is forced to be used even if the backend specifies that no framepointer should be used.

4.5 Call Convention

During a called function several registers that still hold valuable data for the caller, may get clobbered and therefore lose their original content. To solve this matter of losing important data during a function call the registers in question get saved on stack. The GCC takes care of this mostly on its on. In addition some registers might get saved by the callee if they should be restored in functions epilogue.

4.5.1 Function Prologue & Epilogue

Every function has a prologue and epilogue, a point where actions are taken that are not specified in the source, e.g. saving and restoring registers, setting up the stack-frame etc.

Caller

1. ...
2. store first four arguments into fa_n
3. store remaining arguments on stack
4. save: Global- and Caller-Saver-Registers in his Register Save-Area and fa_ into incoming argument space.
5. call Callee

Callee

6. allocate memory for new function and arrange function arguments
7. save Callee-Save-Registers into his Register Save-Area
8. load first four arguments from fa_n
9. load remaining arguments from previous stack-frame
10. ...
11. restore Callee-Save-Registers from Register Save-Area
12. write return value, if needed, into frv_n and return
13. free memory

Caller

14. restore Caller-Save- and Global-Registers
15. ...

These actions are taking in function prologue and epilogue and expand the functionality of the GCC.

4.5.2 Caller- and Callee-Saves

The main difference between Caller- and Callee-Save registers is that the compiler saves the Callee-Save-Registers on its own when a function is called while the Caller-Save-Registers are checked against the registers used in the called function. If a Caller-Save-Register already holding a value is used in the called function the compiler will save the respective register into the register-save-area on the stack.

4.5.3 Passing Arguments

The Eco32 passes the first four functions arguments in registers. In addition the GCC can split larger arguments over more registers and the available stack-space.

There are three different cases we need to look at:

1. Registers are free and size fits
2. there is not enough space in the register to hold the complete Argument

(a) Enough registers are free and can hold the argument

(b) there are not enough free registers

The first case is really simply. We use the already allocated Stack-space to store the remaining arguments.
The second case is rather interesting: If an argument doesn't fit into one register it gets split
a) over multiple registers or
b) over register and stack

Example a) Passing four Integer values:

$4	$5	$6	$7
int	int	int	int

Example b) Passing two Integer values and a long long:

$4	$5	$6	$7
int	int	high-part of long long	low-part of long long

Example c) Passing three Integer values and a long long:

$4	$5	$6	$7
int	int	int	high-part of long long

Stack-Argument-Area-Slot 5 holds the low-part of the long long.

This behaviour is due to Eco32 being a Big-Endian machine.

4.6 Returning

Register $2 and $3 are meant to hold return values of a called functions. This will be the case for primitive data types like int, etc.
When returning a struct from a function this has only to be handled on the stack because the load of patching a split struct from several registers back together is much greater than loading the struct into the memory.

The convention for returning are as followed: If the return value is a primitive data type it will get returned in $2. If $2 can't hold the complete value $3 will cover as a temporary register.
If the return value is a struct we need to improve the behaviour of the Caller and the Callee:

1. Caller passes the address of the stack-space where the result can be placed in $3 to the Callee.
2. The Callee copies the return-value to the place specified in $3. A pointer to the returned value will be placed in $2.

The space where the returned result will be placed is within the local variables & temps - Area of the Caller. This preserves the contract of the stack- and framepointer offsets plus that we don't need to 'waste' an additional stack-area for function-return-values.

Table 4.3: Eco32 Instruction Set

Mnemonic	Operands	Description	Format
add & addi	dst, op1, op2	dst := op1 + op2	RRY
sub & subi	dst, op1, op2	dst := op1 - op2	RRY
mul & muli	dst, op1, op2	dst := op1 * op2, signed	RRY
mulu & mului	dst, op1, op2	dst := op1 * op2, unsigned	RRX
div & divi	dst, op1, op2	dst := op1 / op2, signed	RRY
divu & divui	dst, op1, op2	dst := op1 / op2, unsigned	RRX
rem & remi	dst, op1, op2	dst := remainder of op1/op2, signed	RRY
remu & remui	dst, op1, op2	dst := remainder of op1/op2, unsigned	RRX
and & andi	dst, op1, op2	dst := bitwise AND of op1 and op2	RRX
or & ori	dst, op1, op2	dst := bitwise OR of op1 and op2	RRX
xor & xori	dst, op1, op2	dst := bitwise XOR of op1 and op2	RRX
xnor & xnori	dst, op1, op2	dst := bitwise XNOR of op1 and op2	RRX
sll & slli	dst, op1, op2	dst := shift op1 logically left by op2	RRX
slr & slri	dst, op1, op2	dst := shift op1 logically right by op2	RRX
sar & sari	dst, op1, op2	dst := shift op1 arithmetically right by op2	RRX
ldhi	dst, op1	dst := op1 shifted left by 16 bits	RHH
beq	op1, op2, offset	branch to PC+4+offset*4 if op1 == op2	RRB
bne	op1, op2, offset	branch to PC+4+offset*4 if $op1 != op2$	RRB
ble	op1, op2, offset	branch to PC+4+offset*4 if $op1 <= op2$ (signed)	RRB
bleu	op1, op2, offset	branch to PC+4+offset*4 if $op1 <= op2$ (unsigned)	RRB
blt	op1, op2, offset	branch to PC+4+offset*4 if $op1 < op2$ (signed)	RRB
bltu	op1, op2, offset	branch to PC+4+offset*4 if $op1 < op2$ (unsigned)	RRB
bge	op1, op2, offset	branch to PC+4+offset*4 if $op1 >= op2$ (signed)	RRB
bgeu	op1, op2, offset	branch to PC+4+offset*4 if $op1 >= op2$ (unsigned)	RRB
bgt	op1, op2, offset	branch to PC+4+offset*4 if $op1 > op2$ (signed)	RRB
bgtu	op1, op2, offset	branch to PC+4+offset*4 if $op1 > op2$ (unsigned)	RRB
j	offset	jump to PC+4+offset*4	J
jr	register	jump to register	JR
jal	offset	jump to PC+4+offset*4, store PC+4 in $31	J
jalr	register	jump to register, store PC+4 in $31	JR
trap	-/-	cause a trap, store PC in $30	N
rfx	-/-	return from exception, restore PC from $30	N
ldw	dst, reg, offset	dst := word @ (reg+offset)	RRS
ldh	dst, reg, offset	dst := sign-extended halfword @ (reg+offset)	RRS
ldhu	dst, reg, offset	dst := zero-extended halfword @ (reg+offset)	RRS
ldb	dst, reg, offset	dst := sign-extended byte @ (reg+offset)	RRS
ldbu	dst, reg, offset	dst := zero-extended byte @ (reg+offset)	RRS
stw	src, reg, offset	store src word @ (reg+offset)	RRS
sth	src, reg, offset	store src halfword @ (reg+offset)	RRS
stb	src, reg, offset	store src byte @ (reg+offset)	RRS
mvfs	dst, special	dst := contents of special register	RH
mvts	src, special	contents of special register := src	RH
tbs	-/-	TLB search	N
tbwr	-/-	TLB write random	N
tbri	-/-	TLB read index	N
tbwi	-/-	TLB write index	N

Chapter 5

The Gnu Compiler Collection

The GCC formerly known as Gnu C Compiler was renamed to Gnu Compiler Collection. This renaming was necessary due to the fact that this compiler suite was able to generate more than just a C compiler.[1] The C and C++ version is usually shipped with most Unix operation systems. Additional version are likely to be found in repositories or can be build from source[2].

5.1 General

The most important structure within the compiler is a tree. Everything the GCC does is handled in trees. Be it the abstract syntax for the parsed source code, or the matching of expressions against the machine dependent implementation. Even the optimizations in the middleend are only run on tree structures.

After parsing the source code and creating a parse tree the middleend runs multiple optimization passes. The AST is converted into another type of tree for these passes. Just to mention them, this type of tree is called GIMPLE[3]. Finally, after no more optimizations can be made, the middleend creates a Register Transfer Language representation. This representation is created in combination with the target specific backend.

The important part for this project starts after the middleend has run its optimization passes.

5.2 Static Single Assignment

A static single assignment assigns every variable used in a program code exactly once. If a variable is assigned twice that variable gets rewritten. The following example explains this in a short and neat way.

The following source code is translated into the SSA notation.

```
a = 2;
a = b * a;
```

becomes

```
a.0 = 2;
a.1 = b * a.0;
```

Optimizations in the middleend rely on this representation.

[1] It is also possible to generate a C++, Objective C, Fortran, Java, Ada and Go - other front-ends are available but listing them all would blow this document

[2] See chapter 6 for a short tutorial to build a C compiler from source

[3] GIMPLE is derived from GENERIC and influenced by SIMPLE. Further information about GIMPLE: `http://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`

5.3 Middleend

The parse tree that was converted into a GIMPLE representation by the frontend is now run through several optimization passes. To run these passes more efficiently the representation is changed to SSA. Some of the optimization include return value optimization, redundancy elimination and dead-code-elimination.
At last it converts the GIMPLE tree into an RTL tree and hands it over to the backend.

5.4 Register Transfer Language

RTL is inspired by LISP. It defines the instruction set of the targeted processor in an abstract way. This notation is used by the backend to match the generated RTL patterns to specific assembler instructions. Register transfer language representation is very close to the target specific assembler syntax.

5.4.1 Oprand Modes

Every RTL object can be accessed in a particular mode. This mode restricts the data types that can be used to put into this object. In case of the Eco32 backend the following data types are important. The size of a "byte" has to be specified in the target backend with the macro "BITS_PER_UNIT".

1. QImode(QI)
   Quarter Integer Mode - a single "byte" treated as an integer.

2. HImode(HI)
   Half Integer Mode - an operand of the size of a two "byte" integer.

3. SImode(SI)
   Single Integer Mode - an operand of the size of a four "byte" integer.

5.4.2 Predicates

A predicate helps the GCC to decide whether a "match_operator" or "match_operand" and therefore the whole expression matches. Only matching expression will be used for assembler generation. The following predicates are target independent and can be used in any backend.

1. general_operand
   Allows immediate values, register values and memory addresses.

2. register_operand
   Only register operands can be matched against this predicate.

3. immediate_operand
   Allows any constant that fits in the mode.

4. memory_operand
   Allows valid memory addresses for the target.

Constraints

Constraints specify the matching progress for RTL expression. Constraints tell the compiler what specific kind of register, memory or immediate operand has to match the expression. It is manly used to fine tune the matching process.

Understanding RTL representation

The idea behind this representation is to use an abstract machine with an infinite number of registers. Later on those pseudo registers are eliminated and replaced with the registers present for the target machine. This abstraction allows the compiler to handle most of the code in a target independent way. The best example for this is the algorithm for register allocation on any target. Due to this representation the compiler can do this for any target without taking into account how many registers are present.

The following statement

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

as an RTL expression

```
(set (reg:SI 140)
     (plus:SI (reg:SI 138)
              (reg:SI 139)))
```

The registers 139 through 140 are pseudo machine registers that are allowed only to be accessed in SImode.
The *set* statement takes two arguments, a target and a source. The target is register 140. As for the source, the *plus* statement is taken. This also takes two operands: Register 138 and 138.
Analogous statements exist for multiplications, division and any arithmetic operation. Some operations can be created by combining two ore more simple statements.

```
(set (reg:SI 140)
     (not:SI (xor:SI (reg:SI 138)
              (reg:SI 139)))
)
```

The combination of *not* and *xor* creates a valid statement for a *xnor* statement to match that operation.
The next piece of source code saves the value of variable c in variable d.

```
int d = c;
```

As an RTX it would look like this if variable d resides on the stack:

```
(set (mem:SI <address of variable d>)
     (reg:SI 140)
)
```

The *mem* indicates an address in the memory for a variable.

Giving a complete introduction to the many different kinds of RTX would give this document no more sense in explaining what this kind of representation is used for.

In theory the complete abstract representation of RTL expression is valid but not all targets work the same way such as storing values in registers in single integer mode. Therefore the target's implementation of its instruction set is taken into account for generating such expressions. Those target implementations are implemented in the machine description in the backend.

5.5 Backend

Beside from defining the targeted processor there are some more optimizations to be run in the RTL expressions. Those optimizations are critical to generate the correct assembler instructions. These optimizations regard the correct allocation of registers present for the target machine.

The information about the number of registers present is implemented in the machine description that is given for each targeted machine. Generally each machine description is split into two parts. First there is the .md file that contains the assembler instructions for the target machine. These instructions are implemented in RTL to be matched against. At this pass all pseudo registers must have been allocated to a real register present on the machine. Secondly there is a set of macros and C-Functions. The macros are used to tell the compiler anything that can't be expressed in the .md file. Such as the number of registers. The functions are hooks on functions the compiler calls to generate code for the target. Since the compiler doesn't know anything about how the target works this gives it a way to "understand".

5.5.1 Machine Description

In general all instructions from the targeted machine need to be implemented to tell the compiler how they are output as real assembler instructions. Every instruction is built as followed:

```
(define_insn "<name>"
 [(<add RTL Template here>)]
  ;; Condition Handling written in C
  ""
  ;;Output string
 "<assembler output here>"
  ;;optional Attributes Vector
  <attribute vector here>
)
```

To generate the list of RTL expressions for the target the compiler only takes the named *define_insn* into account or those who are defined to be expanded. If there is a *define_expand* one of three things can happen. Either the logic behind creating the target insn_list expands the insn to a real *define_insn* and invokes DONE. Second way is to invoke FAIL and tell the compiler to use another way of handling this task. If neither DONE or FAIL are invoked the *define_expand* itself is included into the list of instructions. Actually the RTL template given in this definition is included into the list. These templates are optimized and placed back into the list to be matched against the implemented definitions in the .md file. This allows a nearly target independent representation before the assembler instructions are emitted.

The RTL templates allows the matching of patterns and how to find their operands. The condition handling can narrow down under which circumstances this particular instruction can be used. The output string is the actual assembler instruction.
Some of the most important definitions are given below with concrete examples.

- define_insn

```
(define_insn "addsi3"
      [(set (match_operand:SI 0 "general_operand" "=r,m")
            (plus:SI (match_operand:SI 1 "general_operand" "0,0")
                     (match_operand:SI 2 "general_operand" "g,r")))]
      ""
      "@
       addr %2,%0
       addm %2,%0")
```

This definition of an add instruction makes use of several constraints. The *set* pattern matches a single integer operand that can either be a register(r) or memory(m) address. The special case of the constraint *=r* tells the compiler that the register is only meant to be write only for this instruction. In addition this pattern is also a multi alternative. Meaning: It can either produce a add into memory and to add into an register. The output label for a multi alternative starts with an @. Every line afterwards defines a new output. Empty lines are ignored.

Another important definition is *define_expand*. This pattern generate sequences of instructions to handle the task specified. Every instruction emitted by this kind must match a real *define_insn*.

- define_expand

```
(define_expand "movsi"
 [(set (match_operand:SI 0 "general_operand" "")
 (match_operand:SI 1 "general_operand" ""))]
 ""
 "")
{
 /* If this is a store, force the value into a register. */
 if (GET_CODE (operands[0]) == MEM)
 operands[1] = force_reg (SImode, operands[1]);
}

(define_insn "*movsi"
 [(set (match_operand:SI 0 "general_operand" "=r,r")
 (match_operand:SI 1 "general_operand" "r,i"))]
 "register_operand (operands[0], SImode)
 || register_operand (operands[1], SImode)"
 "@
 str %0,%1
 sti %0,%1"
)
```

This pattern consisting of a *define_expand* and a *define_insn* implements a fictional store instruction. That either stores an immediate value in a register or a value of a register in another register. If another movsi where defined storing to memory can also be taken into account. If the condition code is in *define_expand* is taken away this can be an implementation of load and store for the Eco32 when the right *define_insn* is given.

- define_split

```
(define_split
  [(set (match_operand:SI 0 "gen_reg_operand" "")
        (sign_extend:SI (match_operand:HI 1 "gen_reg_operand" "")))]
  ""
  [(set (match_dup 0)
        (ashift:SI (match_dup 1)
                   (const_int 16)))
   (set (match_dup 0)
        (ashiftrt:SI (match_dup 0)
                     (const_int 16)))]
  "
{ operands[1] = gen_lowpart (SImode, operands[1]); }")
```

This pattern replaces a sign extension with pair of shift instructions. If a RTL expression matches against this pattern the entry in the expression list is replaced by the two new defined templates from this *define_split*. Since the new set of instructions to be added to the list is a list itself there can be any number of replacement instructions.

5.5.2 Target Hooks and Macros

Since a backend needs further information about the targeted processor and some of those information can not be implemented like mentioned above, those information gets implemented as macros or target hooks.

Macros define the usage of registers, the size of data types and the general alignment for the targeted processor.

Target hooks are C functions that are only valid for the target they are implemented for. Hooks are taking care of handling function prologue and epilogue, calculating the offset between the stack- and frame- pointer for the targeted system. Since the GCC can't assume anything about those and other special characteristics of new target it needs to be redefined.

The hooks are held in the specific target.c file. All hooks for a target must be defined in the global *targetm* variable. "Many macros will change in future from being defined in the .h file to being part of the targetm structure", from the GCC Online Manual [**?**].

Redefining a hook is pretty simple. All that is needed is the name of the hook to be reassigned to a function for the target like this.

```
#include "target.h"
#include "target-def.h"

/* functions here */

#undef <HOOK_NAME>
#define <HOOK_NAME> <redefinition>

struct gcc_target targetm = TARGET_INITIALIZER;
```

The redefinition is a function that is only valid for the target it was implemented for. The complete set of macros and target hooks is documented in the GCC Online Manual[**?**]. Chapter 6 documents some important functions for a new target backend. The appendix contains the minimal set for a working compiler.

Chapter 6

Eco32 GCC backend

A good reference to create a new backend for the GCC is "Using and Porting Gnu C Compiler" by Richard M. Stallman[**?**] in addition "GCC Online Manual"[**?**] and "How to re-target the GNU Toolchain in 21 patches"[3][1]

6.1 Preparing GCC for Eco32

In order to make the compiler aware of the new target that is going to be implemented the following files need to be modified.

Directly in the top directory of the GCC source the config.sub contains a case for "$basic_machine". This case recognizes the CPU type without any attachment. Include "eco32" in here. Further below in the same section the CPU type gets recognized with a company name. Include "eco32-*" to enable GCC to match "egg-elf32" which is the actual target.

The next file in the top directory is the configure.ac file. Only the case for disabling the Java libffi support matters. Include "eco32-*-*" to enable all possible matches for an Eco32 backend.

With this adjustments the GCC source is ready to be built for the Eco32 as soon as the the machine description and target macros and hooks are implemented.
It's best to keep the new backend implementation out of the original source tree until the first few tests have passed.

The following directories build the new development environment.

```
$BUILD_HOME
|-- build
|-- gcc
    |-- gcc-obj
    `-- patch
        `-- gcc
            `-- config
                `-- eco32
```

The eco32.c, eco32.h and eco32.md file can be found in "$BUILD_HOME/gcc/config/eco32/". They build the main part of this project and are well documented.

Since the source code is well commented only a few macros and hooks that need further explanation will be discussed in the following sections.

[1] This is deprecated but a great tutorial to get a grip on what you are up to.

6.2 Machine Description

The machine description describes the different machine instructions in LISP form. Those expressions will be used to matche the intermediate middle-end representation to the correct machine instruction.

6.2.1 Predicates

Predicates determines whether the instruction specified can be used for the combination of operands. Generally it is not necessary to define new predicates because the GCC has enough to define any instruction in the backend. In this case it was more easy to define the following predicate.
"eco32_mvsrc"
It matches memory references, constant integers, registers, symbol and label references. It is used to determine if a combination of operands designed to build a move instruction actually can be used for one. If the matching test doesn't succeed a general_operand is retuned. In that case the combination of operands can't be used for a move instruction and the compiler has to try again.

6.2.2 Constraints

Constraints allows the GCC to perfectly match instructions to an RTX. Constraints define what kind of operand an instruction takes. For the Eco32 there are four specially designed and new constraints.

- "A" A represents an absolute address. This can either be a symbol reference or a label reference or a constant address in a variable.
- "B" B represents an offset address. This is used to identify the stackpointer, framepointer, argumentpointer or $0.
- "W" W represents a register indirect memory operand. A memory address that was loaded into any register available for general use can identified with this.
- "O" O stands for the constant "zero"(0). Sometimes it's easier to use the $0 register in assembly code. Otherwise the compiler would use an immediate assembler instruction or in worst case load the value zero(0) into another register. To avoid this the constraint "O" was defined.

6.2.3 Prologue and Epilogue expansion

The machine description can define a pattern for a function prologue and epilogue. Just like the target macros it can also call an expansion.

6.2.4 Returner

This instruction is only used in a function epilogue. It emits "jr $31" when called.

NOP

NOP - No operation.
This instruction is needed by the GCC to be emitted when nothing else matches but should.

6.2.5 Branches

The ECO32 Assembler contains 10 branch instructions. From that 10 four can be signed or unsigned.

```
(define_expand "cbranchsi4"
  [(set (pc)
  (if_then_else (match_operator 0 "ordered_comparison_operator"
           [(match_operand:SI 1 "register_operand" "")
      (match_operand:SI 2 "nonmemory_operand" "")])
          (label_ref (match_operand 3 "" ""))
          (pc)))]
```

```
  ""
  {
  /* Force the compare operands into registers.  */
  if (GET_CODE (operands[1]) != REG)
  operands[1] = force_reg (SImode, operands[1]);
  if (GET_CODE (operands[2]) != REG)
  operands[2] = force_reg (SImode, operands[2]);
 })

(define_insn "*cbranchsi4"
  [(set (pc)
  (if_then_else (match_operator 0 "ordered_comparison_operator"
           [(match_operand:SI 1 "register_operand" "r")
      (match_operand:SI 2 "register_operand" "r")])
          (label_ref (match_operand 3 "" ""))
          (pc)))]
  ""
  {

  switch(GET_CODE(operands[0]))
  {
  case EQ:
    return "beq  %1,%2,%l3";

  case NE:
    return "bne  %1,%2,%l3";

  case GT:
    return "bgt  %1,%2,%l3";
  case GTU:
    return "bgtu %1,%2,%l3";

  case LT:
    return "blt  %1,%2,%l3";
  case LTU:
    return "bltu %1,%2,%l3";

  case GE:
    return "bge  %1,%2,%l3";
  case GEU:
    return "bgeu %1,%2,%l3";

  case LE:
    return "ble  %1,%2,%l3";
  case LEU:
    return "bleu %1,%2,%l3";

  default:
    gcc_unreachable ();
  }
  return "";
  })
```

The interesting part about this implementation is the in-line C-Code used to differ between the different kinds of branches. Normally you would have to define an instruction for every flavour of a branch. This short-term writing keeps the machine description clean. The C-Code just checks which kind of branch is currently given and returns the right assembler instruction. On a side note this could easily be done with help of a function in the eco32.c file. Still the machine description allows to use in-line C-Code!

6.2.6 Shifts

The two different kind of shifts, logical and arithmetical are mixed up in this section. The following rules apply to the ECO32:

GCC	ECO32
Unsigned Arithmetical Shift Left	Shift Logical Left
Arithmetical Shift Left	Shift Logical Left
Logical Shift Right	Shift Logical Right
Arithmetical Shift Right	Shift Arithmetical Right

6.3 Target Macros

```
enum reg_class
{
  NO_REGS, GENERAL_REGS, ALL_REGS, LIM_REG_CLASSES
};

#define REG_CLASS_CONTENTS  \
{  \
{0x00000000},    \
{0xA3FFFFFC},    \
{0xFFFFFFFF}     \
}
```

The bitmasks are checked to see whether a register is in a class or not. The GENERAL_REGS contain all registers except $31, $30, $28, $27, $26 and $0, $1. These register are either to be used by an operation system or serve another purpose.

6.3.1 Register Classes

Register classes help the GCC to determine which register can hold what data type. The general purpose registers of the Eco32 are all equivalent and can therefore be used for the same purpose. The following register classes are present.

1. NO_REGS
   No registers in this class - helps to determine that a pseudo register is no yet matched to a real register.

2. GENERAL_REGS
   Register for normal use - Any register in this class can be used for any operation the processor supports.

3. ALL_REGS
   All Registers available

The corresponding bit mask in the eco32.c file allows the GCC to determine the Class of each register. There are several support functions to make this calculation easier.

These classes are used in a later build of the GCC to distinguish between memory accessing assembler instructions and general computation instructions. Every register can and will be used as a base-register. Base-register can contain a memory-address. In addition with an offset the stack on the memory can be accessed.

```
#define ELIMINABLE_REGS  \
      {{ARG_POINTER_REGNUM, STACK_POINTER_REGNUM}, \
      {FRAME_POINTER_REGNUM, STACK_POINTER_REGNUM}}

#define INITIAL_ELIMINATION_OFFSET(FROM, TO, OFFSET) \
  do {                                               \
    (OFFSET) = eco32_initial_elimination_offset ((FROM), (TO)); \
  } while (0)
```

As described in the ABI the argument and framepointer shouldn't be used by the compiler. The macros ELIMINABLE_REGS tells the compiler which register is to be eliminated with which other register. Finally the function eco32_initial_elimination_offset takes over and tries to calculate an offset.

```
#define PRINT_OPERAND(STREAM, X, CODE) \
  eco32_print_operand (STREAM, X, CODE)

#define PRINT_OPERAND_ADDRESS(STREAM ,X) \
  eco32_print_operand_address (STREAM, X)
```

Any target can specify how its operands and operands addresses are to be output. These two macros call functions defined in the eco32.c file.

```
#define GO_IF_LEGITIMATE_ADDRESS(MODE,X,LABEL) \
  do { \
    if (GET_CODE(X) == PLUS) \
      { \
  rtx op1,op2; \
  op1 = XEXP(X,0); \
  op2 = XEXP(X,1);  \
  if (GET_CODE(op1) == REG \
      && CONSTANT_ADDRESS_P(op2) \
      && REGNO_OK_FOR_BASE_P(REGNO(op1))) \
    goto LABEL; \
     } \
    if (REG_P (X) && REGNO_OK_FOR_BASE_P (REGNO (X))) \
      goto LABEL; \
    if (GET_CODE (X) == SYMBOL_REF \
  || GET_CODE (X) == LABEL_REF \
  || GET_CODE (X) == CONST) \
      goto LABEL; \
  } while (0)
```

In some cases the GCC needs to legitimize an address. Legitimization of addresses is important because the GCC doesn't guess how the target addresses its memory.
Normally this would be in the target hooks, since the target hook was spitting errors it was decided to work with this macro. The target hook in question is TARGET_LEGITIMATE_ADDRESS_P which would be defined as a function in the eco32.c file.

An address is legitimate if it is in a register or offsettable or if it is a symbolic or label reference.

6.4 Target Hooks

```
void eco32_print_operand_address(FILE*, rtx);
```

In case of the Eco32 an operand's address can either be a register or combination of a register and an immediate offset. Any of those cases is handled in this function and the right output is produced.

```
void eco32_print_operand(FILE*, rtx, int);
```

Just to make sure that the GCC doesn't try to emit pseudo registers into the assembler this function checks every operand that is about to be printed. It handles the output of registers and delegates the output of any other operand to the corresponding functions implemented in the GCC.

```
struct GTY(()) machine_function;
```

This struct is present for every function the compiler parses. Every "instance" of this struct is initialized with zero. The contents of this struct are to be filled by the function "eco32_compute_frame".

```
static void eco32_compute_frame(void);
```

This function calculates the size of the frame needed for a function. The size calculated can later be used to determine the offset from the stackpointer to place the save registers and function arguments on the stack. It will even fill in a padding if the frame is not aligned right. The function also stores the size of the local variables, saved registers and the size for adjusting the stackpointer in the GTY(()) machine_function struct. Furthermore it stores if a functions needs a framepointer and therefore the offset from it to the stackpointer. In some cases the expanded prologue of a function doesn't need to save the return register. That is the case whenever the compiler deals with a leaf function.

```
int eco32_initial_elimination_offset(int, int);
```

The helper funtion for the macro INITIAL_ELIMINATION_OFFSET. It checks if the two incoming register numbers are frame- or argumentpointer and the stackpointer. The argumentpointer offset from the stackpointer is zero. That is because the outgoing arguments are always on top of the stack. The framepointer offset is the size of the outgoing argument area plus the size of the register save area.

```
static bool eco32_must_pass_in_stack (enum machine_mode, const_tree);
```

This functions checks the arguments to be passed to another functions and decides whether or not it shall be passed in the stack rather than in a register. It returns true for every argument that is a bulk of data.

```
eco32_setup_incoming_varargs (CUMULATIVE_ARGS *, enum machine_mode, tree, int *, int);
```

Functions which have a variable number of arguments are created by using this function. To me more specific: Only the code to set up the arguments on the stack is produced by this function. The previous version of this function was creating bogus code. This was fixed by Stefan Kristiansson.

6.5 Building the Eco32-GCC

After implementing the new backend into the GCC the last step is to build the new cross-compiler. This is done by entering the following command into the shell after entering the gcc-obj directory.

```
../gcc-4.7.1/configure --target=eco32-elf \
--prefix=$BUILD_HOME/../build/ --enable-languages=c \
 --disable-multilib --disable-nls --enable-checking \
 --without-headers
```

```
make all-gcc && make install-gcc
```

The parameters for the configure script have the following meaning:

1. "–target=eco32-elf" Sets the target for the configuration. Any backend has its own name with its own modifiers. Normally there is an operating system to be named. Since we don't actually have one available to interpret the elf32 binaries it's left out.
2. "–prefix=$BUILD_HOME/../build" Sets the installation path for the make script. Everything generated from the make script is stored in this directory.
3. "–enable-languages=c" Enables only the C frontent for this compiler
4. "–disable-multilib" Disables support for libraries that can be used by 32 bit and 64 bit systems.
5. "–disable-nls" Forces any output generated by the compiler to be in English.
6. "–enable-checking" Allows gdb(Gnu Debugger) to trace the stack while debugging the GCC. (This can actually be left out in the final version)
7. "–without-headers" Since there is no port for any library that the GCC could use any header files are disabled. Therefore no standard includes will work.

It is possible to build a running compiler with the configure script in $BUILD_HOME/gcc directory. For first use please use "./configure –new"!

Final Assembler output

The following small C program shall be compiled into ECO32 assembler code. To accomplish this task the compiler has to know how the function arguments are stored, which assembler instruction realizes calls and addition. This and the convention for returning is already specified in the Eco32 ABI.

```
int foo(int a, int b, int c, int d,char e)
{
        int x = a+d;
        return x;
}

int main()
{
        return foo(1,2,3,4,'E');
}
```

This small program calls the function "foo" with five parameters. The first four have to be stored in the corresponding argument register while the fifth should reside in the stack.

This is the output for the small C-Program given above[1]. The output is reasonable and is correct. (Annotations by hand)

```
        .file   "main_10.c"
        .text
        .align 2 # word-alignment for this function
        .global foo
        .type   foo, @function
foo:
        add  $2,$4,$7 # store result of addition in $2
```

[1]With the compiler option "-O2": This enables some optimization passes in the middleend to eliminate unnecessary code. The comments are not emitted by the GCC and were added for a better readability.

```
        jr  $31 # return
        .size   foo, .-foo
        .align 2
        .global main
        .type   main, @function
main:
        sub  $29,$29,24 # enlarge frame
        stw  $31,$29,20 # store register $31 on the stack
        add  $8,$0,69 # store the char 'E' in register $8
        stw  $8,$29,16
        # store the value of $8 as the fifth argument on the stack
        add  $4,$0,1
        add  $5,$0,2 # set the argument registers
        add  $6,$0,3
        add  $7,$0,4
        jal  foo # call foo
        ldw  $31,$29,20 # restore $31 from the frame
        add  $29,$29,24 # release the allocated memory
        jr  $31 # end
        .size   main, .-main
        .ident  "GCC: (GNU) 4.7.1"
```

With this simple implementation a new backend for the GCC can be created. Advancing this implementation is just a matter of time and calls for a good documentation like it was given above in this document. This compiler in any version won't be able to use any predefined header files as long as they are not present for the eco32.

Chapter 7

Special Thanks

Special thanks go to Stefan Kristiansson for patching the GCC Backend causing it to spill the arguments in the correct order and fixing the machine description for signed values.

Glossary

ABI Application Binary Interface
Description of the low-level interface between a computer program and another program or hardware.. 35

AST Abstract Syntax Tree
A compiler-internal representation of source code.. 20, 35

Eco32 Educational 32-bit computer
32 bit RISC processor with 32 32 bit general purpose register.. 4, 35

EGG *E*co32 *G*nu C *G*enerator
GCC backend and Gnu Binutils implementation for the Eco32.. 4, 35

GCC Gnu Compiler Collection
Compiler for many programming languages that supports a lot of processors.. 4, 35

LCC Little C Compiler
A highly retargetable ANSI C compiler. 4, 35

LISP LISt Processing
One of the first high level programming languages.. 21, 35

MMU Memory Management Unit
Hardware that translates virtual memory addresses into physical memory addresses.. 7, 35

RAM Random Access Memory
Memory that can be read and written.. 6, 35

RISC Reduced Instruction Set Computer
A CPU design that defines system with a small but highly optimized instruction set.. 6, 35

ROM Read Only Memory
Memory that can't be written to.. 6, 35

RTL Register Transfer Language
A internal representation for register allocation in the GCC.. 21, 35

RTX RTL Expression
A internal expression of operations to be emitted as assembler output for the GCC.. 22, 35

SSA Static Single Assignment
An intermediate representation for variables in compilers.. 20, 35

List of Figures

List of Tables

Bibliography

[1] Eco32-Instructions:
Hellwig Geisse
http://homepages.thm.de/~hg53/cb-ws1112/instrs.html

[2] The Eco32 User Manual
Martin Geisse
homepages.thm.de/~hg53/eco32-on-s3e/eco32.pdf

[3] How to retarget the GNU Toolchain in 21 patched
Anthony Green
http://atgreen.github.com/ggx/

[4] MIPS RISC Prozessor Supplement 3rd Edition
The Santa Crus Operation Inc.
www.uclibc.org/docs/psABI-mips.pdf