Extended GCC Compiler

The Extended GCC compiler development system is a compiler generator that automatically creates an executable compiler out of an architecture description. In our case, ASIP Meister automatically creates this architecture description itself and an afterwards running program provided by the developers of ASIP Meister. In this chapter, some basics about the buildup of retargetable compilers, creation and usage of a GCC compiler for our specific ASIP Meister CPU's are explained.

Basics about Retargetable Compilers

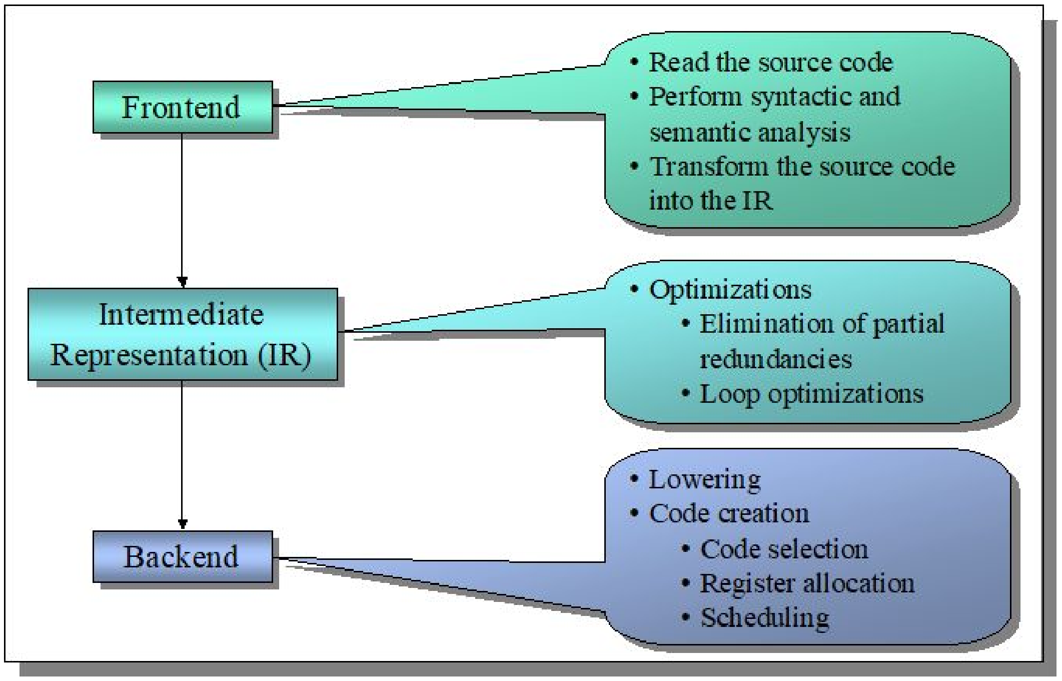
Frontend
Intermediate Representation (IR)
Backend
• Read the source code
• Perform syntactic and semantic analysis
• Transform the source code into the IR
• Optimizations
• Elimination of partial redundancies
• Loop optimizations
• Lowering
• Code creation
• Code selection
• Register allocation
• Scheduling

Figure 8-1:
A Typical Buildup for a Retargetable Compiler

A typical retargetable compiler is separated into **three** phases, as shown in Figure 8-1. The first stage is architecture independent, but source language dependent. This phase reads the source code, inspects it for syntactic and semantic correctness, and transforms it into the second phase, the intermediate representation (IR). This IR is as well source language independent as architecture independent and it is used to perform the optimizations. This implies, that the optimizations can be easily reused, if the source language or the architecture changes. The third phase is source language independent, but highly architecture dependent. This backend first transforms the IR into a low-level form. That means, that high-level structures, like polymorphic procedure calls are replaced by jump tables or that complex data structures are disassembled into elementary memory accesses. Afterwards the final assembly code has to be created. This part is separated into **three** steps. In the first step, code has to be selected out of the lowered IR. This code selection is not unique, as there are always different possibilities to represent some statements in assembly language. This code selection works with virtual registers, which are replaced by real registers in the second step. This register allocation might lead to additional stack accesses for swapping values out, if no free real register can be found to hold the value of a virtual register. In the third step, the code is scheduled, to minimize penalties for data dependencies. Every step from this code selection has a great influence on the outcome of the other steps. The sequence of these steps is not determined and different compilers work with different sequences. The above given order is just exemplary.

Creating the Extended GCC Compiler in ASIPmeister

The ASIP Meister complier generation supports the base processor Brownie. In order for the compiler to generate instructions that are extended from the Brownie processor, some definitions are necessary. Using [C Definitions], you can define the C description specifications that are supported by the ASIP Meister compiler and the C description variables that support the instructions extended from the Brownie processor. The “Compiler Generation” is possible only for the processor extending a base processor Brownie. One can define how to represent the extension instruction added to the base processor Brownie in C code during the “C Definition” stage. Extension instruction definition enables a complier to output assembly code complying with the extension instruction.

C Definition

On the “Ckf Prototype” tab of “C Definition” sub-window in ASIPmeister, you can define all the newly defined instructions as described in the Page-65 of [TUT] and on the Page-57 of [UM]

Compiler Generation

At “Compiler Generation” stage in ASIPmeister, the processor compiler and the Binutils can be generated. The processes involved in the “Compiler Generation” phase are as follows.

1. Input Description Generation: select and run.

2. GNU Tools Generation: select and run.

If you select “Input Description Generation” from the drop down list, the compiler extended description will be generated. After the generation terminates successfully, if the design data file was called” browstd32.pdb”, a new directory called” browstd32.swgen” will be generated inside ”meister” directory, and inside this directory the compiler extended description is generated in a file named ” browstd32.xml”. The generated compiler and Binutils supports the Ckf defined at “C Definition”. After you select “GNU Tools Generation” and the generation terminates successfully, the compiler and the Binutils will be generated in default directory called ”browstd32.swgen” in the “meister” directory.

Using custom instruction in the C Program

Once the custom instruction is defined in the “C Definition”, they can be used in the C program as described on the Page-63 in [UM] and on Page-68 in [TUT]. There are two methods to use the custom instruction here:

1. When you want to write the extended instruction description in C code, you have to add “___builtin_brownie32_” directive to the “ckf definition”. An example is demonstrated in the following for a custom instruction AVG with three parameters (AGV rd, rs1, rs2).

```
int a=12,b=23,c;

c = ___builtin_brownie32_AVG(a, b);
```

2. In some cases, above method does not work, e.g., when an instruction returns a “void”. Standard inline assembly directives can be used to write the extended custom instruction, as follows:

```
___asm___ volatile (

"avg %[my_out], %[my_op1], %[my_op2]\n"

: [my_out] "=&r" (c)

: [my_op1] "r" (a),[my_op2] "r" (b)

);
```

Using the Extended GCC Compiler

In the lab, the Makefile is automatically doing all the steps. However, you can also use the generated compiler separately; the following demonstrates a method for cross compiling a C program file called “bubble.c”.

1. First, you have to add the path “browstd32.swgen/bin” to the $PATH environment variable.

```
export COMPILER_DIR=/brownie/meister/browstd32.swgen/bin
```

or

```
export PATH=/brownie/meister/browstd32.swgen/bin:$PATH
```

2. Compile the C program into .s assembly file

```
${COMPILER_DIR}/brownie32-elf-gcc -S -combine -O3 bubble.c -o bubble.s
```

3. Assemble bubble.s file into .o object file. Also, assemble startup.s (startup code) and handler.s (interrupt handler) file to object files.

```
${COMPILER_DIR}/brownie32-elf-as -o startup.o startup.s;
```

```
${COMPILER_DIR}/brownie32-elf-as -o handler.o handler.s;
```

```
${COMPILER_DIR}/brownie32-elf-as -o bubble.o bubble.s;
```

4. Link the object files using the linker script “browtb.x”. This script declares some important information about the stack pointer, text and data sections, program counter address after the resetting the CPU.

```
${COMPILER_DIR}/brownie32-elf-ld -o bubble -T browtb.x bubble_uart.o
startup.o handler.o
```

5. Converting compiled object file to memory image of the C program. Use “gccout2img” for a file in elf format obtained through normal compilation. The script “gccout2img” outputs *TestData.IM* and *TestData.DM*, with which the user can perform simplified test simulations.

```
gccout2img bubble
```

Library with Standard Functions for ASIP Meister / GCC / Hardware Prototype

Many applications use some standard library calls like *printf*, *malloc* or *atoi* that are not declared in the standard of the C programming language, but which are nevertheless declared in the C standard library. Now, GCC is a compiler for the C language and does not provide an implementation for the standard library (in fact there are some huge fragments that would have to be adapted to our specific environment, what has not been done due to the complexity of a full run-time implementation).

To close the gap between a plain C compiler and the wish of letting complex algorithms run in hardware and produce understandable output we are providing some basic *stdlib* functionality, which is dedicated to the environment of ASIP Meister, the GCC compiler and our hardware prototype. This basic *stdlib* functionality is extended on demand to reflect the latest changes of our environment. Thus, it is not explained exhaustively or in high detail. Instead, the underlying concepts and the needed steps for using the basic *stdlib* library is explained here, plus some of the main functions for using our touch screen LCD with some examples.

All typical functionalities of our *stdlib* implementation are available in the directory */home/asip00/epp/StdLib/.* The functionality is encapsulated into a header file that is providing the interface and a documentation of the functionality and a C file for implementing the header. You can use these files by linking/copying them into your application project subdirectory and using “*make sim*” to compile your application and the *stdlib* files into one binary and simulation file, as it will be shown in the example below. Linking to the files instead of copying them has the advantage, that you always have the latest version of these files in your project. Some of the files in the *stdlib* directory have dependencies to other files of this directory. Thus, you can get a compiler error, that a specific header file was not found when you try to compile your project. You then have to manually link to the dependent files as well. Just linking to all available files is generally not a good idea, as this makes the compilation process take longer and it increases the needed memory size of your application.

Now we will demonstrate how you can create a small application that is using the LCD of the hardware prototype:

- Create a new subdirectory inside the application directory of your ASIP Meister project and change into this new directory
- Copy or Link to *lib_lcd*, *loadStorByte* and *string*, by executing:

  ln –s /home/asip00/epp/StdLib/lib_lcd_320.* .

  ln –s /home/asip00/epp/StdLib/loadStoreByte.* .

  ln –s /home/asip00/epp/StdLib/string.* .

  lib_lcd has a dependency to *loadStoreByte* and *string*, that is why you need both.

  The *lib_lcd_320* exists in different C implementations, depending on your target LCD. One C file is for the LCD with a 240x128 resolution, the other one is for the 320x240 resolution, and the last one is for the dlxsim simulation. Make sure that at most, one of these C files is available in your application subdirectory; otherwise, the linker will complain about multiple implementations (one per C file) of the LCD functionality.
- Prepare an "*env_settings*" file, as shown in Chapter 6.4.
- Add a new C file that contains your main method. This main method will contain the following code as example:

  t_print("Hello World\r\n");

  t_printInt(42);

  The LCD needs the "$\backslash r \backslash n$" in the string, as it is handling carriage return (\r) and line feed (\n) independently.
- Compile your project by executing "*make sim*".

The resulting program can be simulated with dlxsim or ModelSim or it can be uploaded to the hardware prototype (explained in Chapter 6.4) depending on the selected LCD library.

After you have once compiled a C-Code that rarely changes (e.g. the libraries), you can reuse the created assembly code for future compilations. That will enormously speed up the whole compilation process. This is especially good for all applications in the *StdLib* directory. Just compile them one time with "*make sim*" and then copy the created *.asm* file from the "*BUILD_SIM*" directory to the directory of your other application files, but name it .s instead of .asm. Then remove the C code (but not the header) to make sure the code does not exist twice (in the C code and in the Assembly file). For instance:

cp BUILD_SIM/loadStoreByte.asm loadStoreByte.s

rm loadStoreByte.c

However, when moving from dlxsim to FPGA implementation then you have to delete these assembly files, link to the C files, and recompile them. If you are often switching between dlxsim and FPGA, then it may be beneficial to create two application directories with the different libraries and "*env_settings*" specified for dlxsim and FPGA respectively and using the same application-specific source code in both directories (e.g. with a link).

As mentioned above, the "*lib_lcd*" exists in different variants. The files "*lib_lcd_240.c*" and "*lib_lcd_320.c*" are for the FPGA prototype. They implement the real protocol for communicating with the I^{2}C bus and thus with the LCD. For a simulation, we cannot use this implementation, as it is waiting for certain responses from I^{2}C/LCD, but in dlxsim/ModelSim simulation, these answers will never appear. We have therefore implemented "*lib_lcd_dlxsim.c*" which simply writes every single character that is send to the LCD to an output file (a virtual LCD). This is a very good debugging possibility for printing text, although it is not helpful for any graphic output to the LCD (e.g. lines, bars ...), as these control words are hard to understand manually. For dlxsim, the characters are either printed to the console whenever they appear or they are collected and printed to a file (parameter "-lf" in Chapter 3.2.1). For ModelSim, the characters are printed to a file "*lcd.out*" in the ModelSim directory. The header file "*lib_lcd.h*" is valid for all C files, but you have to make sure, that at most one of the both C files is available in the directory when compiling with "*make sim*". Otherwise, you will get two implementations of the LCD functions and the assembler will complain, that he cannot decide which one to choose.

Functions of the LCD Library

This chapter summarizes some of the available functions to access the features of the LCD of our hardware prototyping board. The library also contains some high-level functionality that is useful, but introduces a dependency to "*string.c*". Therefore, this library has to be provided as well when using "*lib_lcd*".

The most important basic I/O instructions are: "*t_print(char*)*", "*t_printInt(int)*", and "*t_printHex(int value, int digits)*". You can use them to print strings and numbers, for instance:

char tempString[] = "\r\n\t\t";

t_print("Hello World!");

t_print(tempString);

t_printInt(23);

t_print(“ = “);

t_printHex(23, 0);

t_print(tempString)

t_printInt(42);

t_print(“ = “);

t_printHex(42, 4);

The “*printHex()”* function can trim the output to a given number of digits (4 in this case). To not trim the number of digits you have to give ‘0’ as parameter. The output of the above example looks like:

Hello World!

23 = 0x17

42 = 0x002A

In the following, we describe further functions of the LCD. Some of them are generally sending a command to the LCD (you have to use the LCD manual [eDIP] to find the available commands) and some of them are offering typical commands (e.g. *drawline*) as convenience functions.

int checkbuffer()

This function returns the number of available bytes in the send buffer of the LCD. It can be used to wait for a return value of the LCD. For example when pressing a button on the touch panel, a value of this button is written to the LCD send buffer.

int getbytes (char* dest, int bytes_to_read)

This reads a specific number of bytes from the send buffer. With the “*checkbuffer”* function, you can test how many bytes are available in the buffer.

int sendcommand (const char cmd0, const char cmd1, const int options[], const char text[],
int intcount, int charcount, int address)

This is a general function for sending commands to the LCD. The following commands internally use “*sendcommand”* to realize their functionality. The parameters of “*sendcommand”* are:

- Two chars, specifying the command
- Depending on the type of the command, some options
- Depending on the type of the command a string with a predefined size
- The address of the LCD (defined in *lib_lcd.c*)

int t_print (const char* str)

This function writes a string to the LCD terminal. The "*t_*" indicates a command for the console mode of the LCD, compared to the graphic mode (*g_*) of the LCD, which is explained later.

int t_cursor (int onoff)

Turns on (1) or off (0) the blinking cursor of the terminal.

int t_enable (int onoff)

Turns the display on (1) or off (0). When the display is turned off, all submitted data is ignored. Previously sent data (when the display was on) is buffered and will become visible again, after the display will become turned on again.

int g_print (const char* str, int x, int y)

Writes a string to the coordinates (x, y). You must not send control signals like \n in this function. They are only available for the t_print function.

int g_drawrect (int x1, int y1, int x2, int y2)

int g_drawline (int x1, int y1, int x2, int y2)

Draws a rectangle/line.

int g_makebar (int x1, int y1, int x2, int y2, int low_val, int high_val, int init_val, int type, int fill_type, int touch)

Creates a bar graph at the defined coordinates. "*low_val"* and "*high_val"* describe the minimal and maximal (at most 254) value of the bar graph. "*init_val"* defines the initial value and "*type"* and "*fill_type"* adjust the appearance of the graph. Type=1 will draw a bar in a box and the "*fill_type*" (in the range of 1 to 15) then defines the fill pattern. Type=3 will draw a line in a box and the "*fill_type*" will then define the thickness of the line. For more details refer too [LCD].

With "*touch"* you can define, whether the bar graph will be user changeable by the touch screen. Every bar graph gets a unique number, which is returned by the function. At most 32 bar graphs are supported by the display. When the touch screen functionality is activated and the user changes the value of the bar graph, the LCD automatically writes the number of the changed bar graph and its modified value to the buffer, from where it can be received with the checkbuffer and getbytes function.

int g_setbar (int barnum, int value)

Sets an existing bar graph to a specific value.

int g_makeswitch (const char* str, int x1, int y1, int x2, int y2, int down, int up)

Creates a switch (button) with a label. The parameter "*str*" contains this label, preceded by a control char which defines the alignment of the label. "C" means centered, "L" means left- and "R" means right-aligned. *down* and *up* define the code, which the LCD will write into the send buffer, when the button is pressed or released. When 0 is provided as parameter for up or down, then nothing will be written to the buffer for the specific activity.

int g_makeradiogroup (int group_number)

A new defined switch can be assigned to a specific radio group. A radio group automatically makes sure, that at most one switch of the group is pressed.

int g_makemenubutton (const char* str, int x1, int y1, int x2, int y2, int down, int up, int select, int space)

A menu item is created. With "*str*" you can define the appearance and the menu entries. The first character defines the direction in which the menu will open ("L": left, "R": right, "O" up, "u": down). The second char defines the alignment of label on the menu item ("C": centered, "L": left aligned, "R": right aligned). The labels of the menu entries follow afterwards and are separated by a "|" sign. "*down*" defines the value that will be written to the buffer, when the menu item is pressed and up defines the value that will be written to the buffer, when the menu is closed, without choosing a specific entry (aborted). "*select*" defines the base value that is used to compute the value that will be written to the LCD buffer, when a menu entry is chosen. This value is computed as: base value + entry number – 1. "s*pace*" defines the gap in pixels between the menu entries. This is a global value, thus it will also change the appearance of already existing menus.

Functions of Further Libraries

Besides the LCD, further functionality and helping functions are available. They will be summarizes in this section to give an overview of the available features. You should look into the libraries to find out more.

lib_uart: besides printing to LCD, you may also print to the UART port using the "*u_print*", "*u_printInt*", and "*u_printHex*" functions (similar to "*t_print*" etc.). You can read from the UART port with "*u_getbytes*". However, you need a HyperTerminal running at the pc to which the UART is connected (see Chapter 6.4.2 for details about the HyperTerminal). In Modelsim, uart.out is automatically created while for dlxsim you can pass the parameter "-uf" to print text in a file instead of appearing on the console.

lib_audio: you can write uncompressed PCM audio data to an audio port using the "*audioAddressR{L/R}*" pointers. The corresponding sample width of the corresponding IP core in the VHDL-framework (default 16 Bit) can be configured in "*AudioOut_Types.vhd*" and the sampling rate (default 40 KHz) can be configured in "*dlx_Toplevel.vhd*" (search for 'i_audio_out').

lib_clock: you can use the pointer "*clockAddress*" to read and write to access a special register that automatically increments by one in each cycle. You can use this to benchmark the performance of a certain C-Code by e.g. writing a '0' to it to start the benchmark and to read it again at the end of the benchmark. Make sure that the code that will be benchmarked does not contain any unnecessary I/O instructions (e.g. "*t_print*") as they are extremely slow.

String: contains some typical string manipulation functions, e.g. "*strlen*", "*strcat*", "*strcpy*" … It additionally contains two functions to convert an integer number to a string representation, as it is used in the I/O libraries, e.g. "*t_printInt*" or "*u_printHex*".

Changing the Frequency

In order to change the frequency of your design and see if it still works or not (required for the last Session), the Framework contains a DCM (Digital Clock Manager) that generates five different clocks. By changing the knob on the external PCB, a multiplexer (in the "dlx_toplevel.vdh" file) selects the corresponding frequency that you want to apply on your design.

Knop value	Frequency (MHz)
0	100
1	80
2	66
3	50
4	40
5	25
Else	100

Figure 8-1: Knob Position & Corresponding Frequencies