Assignment 1a: Compiler organization and backend programming

Roel Jordans

2016

Organization

In assignment 1 you will take a closer look at how a simple blinking led program for an Atmel AVR microcontroller (such as is found on the Arduino boards) is compiled. You will mostly be working with the LLVM compiler framework[1] for this, which is in the progress of obtaining support for the AVR architecture so plenty of holes still exist for you to fix! But for some parts we will also be using `avr-gcc`, which has much more mature support for the AVR architecture.

This assignment consists of three parts, this document representing part A of assignment 1, which are each tested with a set of questions[2], all constructed around the example C application shown in Listing 1. First you will investigate the steps that are taken during compilation and investigate the results of the compilation process. From these results we then identify some optimizations that were missed by the compiler. The second part of this assignment focusses on changing the compiler backend to generate better code. Finally, the third part compares the results with those obtained using `avr-gcc` and further improves our code generation to bring it on par with GCC (for this example).

```
#define F_CPU 16000000 /* Set clock frequency for delay timer */
#define BLINK_DELAY_MS 1000

#include <avr/io.h>
#include <util/delay.h>

int main(void) {
  /* set pin 5 of PORTB for output*/
  DDRB |= _BV(DDB5);

  while (1) {
    /* set pin 5 high to turn led on */
    PORTB |= _BV(PORTB5);
    _delay_ms(BLINK_DELAY_MS);

    /* set pin 5 low to turn led off */
    PORTB &= ~_BV(PORTB5);
    _delay_ms(BLINK_DELAY_MS);
  }
}
```

Listing 1: *A simple blinking led program*

[1] `http://llvm.org`

[2] Make sure to read the footnotes, they may contain useful information and outright hints.

1 Compilation flow

The first step in our process is to compile the application with the existing LLVM based compiler that can be found in your home directory on the server[3]. Currently, this compiler implements the front-end, optimization layer, and back-end. But does not provide a linker, to solve this we will use the linker from the AVR GCC to complete the compilation process.

1.1 Creating the object file(s)

To compile the source code of our application into an object file we execute the following command[4]:

```
$ clang --target=avr -mmcu=atmega328p -I/usr/lib/avr/include -Os -c led.c
```

First we use `clang`, the LLVM compiler driver and C front-end, to compile the input C file `led.c` into an ELF object file `led.o`. This command assumes that `led.c` is in your current working directory. The `-c` flag specifies that we want to stop the process at the object file generation want to produce a `.o` file, the default filename is the same as the input file except for the file extension. Furthermore, we specify that the include files for the AVR C library can be found at `/usr/lib/avr/include` using the `-I` option; that we want to optimize the code for code size using the `-Os` flag; and that we want to generate code for an AVR processor, and specifically the `atmega328p` using the `--target=avr` and `--mmcu=atmega328p` flags.

There are several other stages at which the compiler driver can produce its output, which can be very useful when trying to understand the internals of the compiler. Similar to the `-c` flag we can use one of `-E`, `-S`, or `-emit-llvm` combined with either `-c` or `-S` to select other stages to stop the process. Alternatively, you can provide the `-save-temps` flag to save all of these intermediate results in one run.

Exercise 1

Use the `-help` option of clang[a] to find out the meaning of the `-E`, `-S`, `-c`, and `-emit-llvm` options, which stages of compilation do they relate to.

[a] Hint: Most of tools are capable of providing basic usage information through a `-help` option, although some tools use longer (such as `--help`) or shorter option names (such as `-h` or `-?`) as alternatives.

Exercise 2

Next, compile the `led.c` code using LLVM with the command introduced above but add the `-save-temps` option. This produces a set of outputs corresponding to the compilation stages. Each of these outputs can be distinguished by their own extension. Which extension belongs to which of the compilation stages[a] and what optimizations[b] have been done at this stage?

[a] Hint: The compiler driver also has a `-v` flag which shows the commands that are executed for the internal stages.

[b] Hint: Some of the produced intermediate files will have a binary format and will require another tool if you want to inspect their contents. For example, `llvm-dis` allows you to inspect a `.bc` file and `avr-objdump` shows you the contents of an ELF object such as a `.o` file. Try using the `-d` and `-x` options of `avr-objdump`.

[3] If you have an Arduino board and the Arduino IDE installed you can also first check the program is working by copying it into the IDE and downloading it onto the actual board.

[4] In the command listings shown in this document you will see the `$` symbol used to represent the command prompt, many of the online instructions that you will find will use the same convention. The advantage is that this allows you to distinguish the commands entered from the output which will be shown without the `$` sign.

1.2 Linking the application into an executable

Once we have created all the object files for our program (in our case `led.o`) we can link the program with the C runtime using the linker. The AVR LLVM version we're using doesn't provide a linker yet so we'll use AVR GCC for this. The command can be seen below together with its output:

```
$ avr-gcc -mmcu=atmega328p led.o -o led.elf
led.o: In function ‘LBB0_1’:
led.bc:(.text+0x18): undefined reference to ‘__builtin_avr_delay_cycles’
led.bc:(.text+0x22): undefined reference to ‘__builtin_avr_delay_cycles’
collect2: error: ld returned 1 exit status
```

Oops, that didn't work nicely yet, the linker complains and gives an error...

So, what went wrong? The linker is complaining about an *undefined reference*, which means our program references some symbol (e.g. a variable or function) that is not available in the provided object files or in the standard C libraries that are included with the compiler. In this case it complains about the symbol `__builtin_avr_delay_cycles`. Inspecting the AVR assembly output (the `led.s` file in case you didn't figure that out yet) shows us that there are some calls to this function, so where did they come from and can we avoid those? The name of the function tells us that this is a compiler built-in function (or builtin in compiler speak), these functions can be used to tell the compiler to generate some specific code which is difficult (sometimes impossible) to recognize if you would describe its behaviour in C. Examples are telling the compiler to insert a cache flush command or (in our case) a delay loop.

Trick question 1

Why can't we just write a delay loop in C?[a]

[a] Hint: Answers to the trick questions can be found at the end of the assignment part.

It seems that our LLVM based AVR compiler doesn't implement this builtin yet, too bad. Inspecting the `util/delay.h` file in our include folder[5], shows us the implementation of the `_delay_ms` function and contains some extra documentation of that function in the comment just before it[6]. According to this documentation we can define `__DELAY_BACKWARD_COMPATIBLE__` in our source file[7] in order to avoid the builtin.

Trick question 2

The C library usually only contains a function definition in the header files (telling the compiler what the function looks like but not what it does) and keeps the implementation in a library that so that both parts can be combined during linking. This isn't done for the `_delay_ms` function...
What could be the reason for doing this here?

Well, that should fix it and give us a way to compile our program using LLVM.

Exercise 3

Make it happen, change the `led.c` file, add the `__DELAY_BACKWARD_COMPATIBLE__` define at the top, and compile and link it again. This time there should be no problems and you should end up with a `led.o` file generated by LLVM, and a `led.elf` file that was the result of your linking step using GCC.
What is the size of the `led.elf` file?

[5] Which was `/usr/lib/avr/include` as we told to the compiler with the `-I` option.
[6] Feel free to have a look, it's on lines 104–140 of the file.
[7] Make sure to define it before the inclusion of `util/delay.h` or it won't have any effect.

Let's inspect these files to see the differences before and after linking! As mentioned before[8] there exists a tool called `objdump`, or in case of our AVR flavor `avr-objdump` which allows us to inspect ELF[9] files. Let's have a better look at the output of this tool.

Exercise 4

First we will have a look at the section overview information of the previously generated `led.o` file. To achieve this use the `-h` option to show all section headers in the file.

```
$ avr-objdump -h led.o
```

This should show you that the object file contains three sections and provides their names, sizes (in hexadecimal), virtual memory address[a], load memory address[b], the location in the file, and the alignment[c] it should have in memory. It also lists some flags and actions that need to be taken for each of these sections.
Looking at the output of `avr-objdump`, answer the following questions:

- In which section can we find our program code?
- In which section should we put initialized[d] data?
- In which section should we put uninitialized data?
- For our `led.o` file, how many bytes of program memory will our program use?

[a] Where to put the data in the processor address space when it is loaded into RAM.
[b] Where to load the contents from ROM (if available).
[c] For example if this section is only allowed to start on even addresses when loaded dynamically.
[d] This is data that needs to have a specific value when the program is loaded such as an initialized global variable.

Exercise 5

Now lets have a look at the generated program code and use the `-d` option to disassemble the `.text` section of the object file.

```
$ avr-objdump -d led.o
```

This will show you for each address in the program (part) the assembly instructions of the program next to their encoding. Extra comments (starting with `;`) are shown to translate constants into decimal numbers and to show where the branch instructions jump. The dot (`.`) character is treated specially here, it represents the memory address of the next instruction, for example the `brne .-14` instruction on offset 0x1e will jump to offset 0x12 which is the start of `LBB_0`.

- How many operations are there in our program?
- How many bytes is one operation?
- Is the generated assembly code correct for our intended program[a]?

[a] Hint: where does the branch `brne .-2` jump to...

So it seems there is another problem. The program contains an infinite loop, the inner loop of the delay code doesn't seem to come out properly... This is the part that got generated from some inline assembly code in the C library `util/delay.h` file through calling the `_delay_loop_2` function from `util/delay_basic.h`. Listing 2 shows the assembly code from `led.s` that corresponds to the incorrectly generated code.

Looking at the assembly code tells us that it should work fine.

- The code defines a local label using a number followed by a colon symbol (`1:`) and puts it before the substract instruction that adjusts the loop counter.

[8] In one of the footnotes of the previous exercises, you should make sure to read the footnotes, they contain some really useful bits of information.
[9] Reading material: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

```
25    ;APP
26    1: sbiw R26,1
27    brne 1b
28    ;NO_APP
```

Listing 2: *The problematic loop's assembly code in led.s*

- From the instruction-set manual we know that the arithmatic instructions set the zero flag in the flags register when the result of the computation is 0.
- The `brne` instruction uses this flag to decide whether or not to execute the branch[10].
- The branch uses the branch target `1b`, which means branch backward[11] to the last local label with identifier 1.

All together this code should assemble to work correctly which means that the problem we're experiencing is somewhere in the translation of the inline assembler code into object code. Well, that's annoying but maybe a bit too much to fix at this point in this course[12] so we will avoid the problem and use the AVR assembler provided with GCC for now.

To do this, you can simply provide the `led.s` file obtained from clang to `avr-as`, which is the assembler tool. You will need to specify the architecture version to this tool. Together this adds up to the following sequence of commands for compiling our application:

```
$ clang --target=avr -mmcu=atmega328p -I/usr/lib/avr/include -Os -S led.c
$ avr-as -mmcu=atmega328p led.s -o led.o
```

Ok, back to our analysis of the compilation results. Running `avr-objdump -d led.o` again will now show some changes to the output. In particular you may notice that all the relative branches now look something like `brne .+0`, using `.+0` as their target. So, do they now all just branch onto the next instruction?

No, something else has happened. Inspecting the flags for the `.text` section of the program[13] shows a new flag named `RELOC` that popped up. This tells us that there are adresses which have not been resolved yet[14] and are left to the linker to be corrected. A special set of relocation records has been added to the ELF file to inform the linker where to point these branch instructions to.

Exercise 6

We can inspect the relocation records using the `-r` flag[a] of `avr-objdump`. This will show you a table with the offset of the branch operation, its type, and the actual address it should branch to. One of our problematic branch instructions was the instruction at offset 0x16, to which location should this instruction branch according to the relocation record?

[a] You can also use the `-x` flag which combines the information of all ELF headers into one nice overview.

[10] Equality from a compare is generally communicated inside a processor as the difference between the two compared values being zero. Hence, branch-if-not-equal can also be read as branch-if-not-zero.

[11] Similarly, `brne 1f` would mean branch-no-equal to the first label with identifier 1.

[12] Feel free to try and fix this for the official AVR LLVM for extra bonus credits.

[13] Using the `-h` flag of `avr-objdump`.

[14] Our branch targets are an example of this but there are other kinds of addresses as well. Another example would be global variables accessed by the code.

Exercise 7

Great, that's resolved. Let's link the application and see the effects linking has and start with checking the ELF headers of the linked program.

```
$ avr-gcc -mmcu=atmega328p led.o -o led.elf
$ avr-objdump -h led.elf
```

We now see that the program code has grown quite a bit during linking.

- How many instructions are there now in the program code[a]?
- At which address will our program be loaded in the program memory of the processor?

[a] Hint: Remember exercise 4?

Our ELF executable now contains all the instructions needed to run on the AVR processor. Including the parts that handle the processor initialization such as, setting up the environment (e.g. stack) that C expects, calling the main function, and halting the program execution when the main function returns.

Exercise 8

Disassembling the final `led.elf` result will give us a lot of output which won't fit on the screen. Linux allows you to redirect output of one program into the next using the pipe symbol (`|`), in this case it may be useful to combine it with `less` so that we can scroll through the output[a].

```
$ avr-objdump -d led.elf | less
```

We now see those new instructions that have been added to our program. There's a table called `__vectors` at the top as well as some other new functions. This table is our interupt vector table, each time an interupt signal reaches to processor it will jump to the matching entry in this table and execut the interrupt handler. Interrupt 0 is executed on reset and will be where our program starts running.

- Describe the order in which functions are called.
- What are the three mayor steps tasks of the `__ctors_end` function?
- What happens of we trigger an interrupt for which we didn't provide a handler[b]?

[a] You can use the q button to exit less.
[b] Hint: Any interrupt other than 0.

Trick question answers

Trick question 1: A loop without a result and we've told the compiler to optimize our code. Hmm, that sounds like a nice candidate for removal, it would make the program a lot faster...

Trick question 2: Implementing functions in a library makes sure that the object files don't get too big and avoids having to resolve duplicate symbols when a program with multiple object files is linked. This is very nice to have if you want to support bigger functions such as `fopen` or `printf`. However, it also prevents the compiler from inlining smaller library functions, since linking is done after the assembly code is generated! In our case that would mean that the delay code would have to take the time for the call into account (and that the resulting program would also be larger as we can't optimize the delay loop iteration count away like we can after inlining).