Assignment 1: Compiler organization and backend programming

Roel Jordans

2021

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, which are each tested with a set of questions[2], all constructed around the example C application shown in Listing 1. In the first part, 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).

```
1  #define F_CPU 16000000 /* Set clock frequency for delay timer */
2  #define BLINK_DELAY_MS 1000
3
4  #include <avr/io.h>
5  #include <util/delay.h>
6
7  int main(void) {
8    /* set pin 5 of PORTB for output*/
9    DDRB |= _BV(DDB5);
10
11   while (1) {
12     /* set pin 5 high to turn led on */
13     PORTB |= _BV(PORTB5);
14     _delay_ms(BLINK_DELAY_MS);
15
16     /* set pin 5 low to turn led off */
17     PORTB &= ~_BV(PORTB5);
18     _delay_ms(BLINK_DELAY_MS);
19   }
20 }
```

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 Installing and building LLVM

Before we can get started we need to get our copy of LLVM onto the servers. Part of this has been prepared in your home directory on the servers but the last bit still needs to be finished.

First we have created a directory named `avr` for assignment 1 on the server for you:

```
$ mkdir avr
```

Then we changed into the directory[3] and checked out a copy of version 5.0.1 of the LLVM framework and its clang fronted.

```
$ git clone --depth=1 -b release_50 https://github.com/llvm-mirror/llvm.git
$ cd llvm/tools
$ git clone --depth=1 -b release_50 https://github.com/llvm-mirror/clang.git
```

Using the `--depth=1` flag to make sure we only get the latest version and skip downloading the entire history of the project[4], and the `-b release_50` to select the 5.0 release branch. The last version on that branch is version 5.0.1 which we'll be using in this assignment.

Now to get everything built we need to create a build directory and configure LLVM to only build the components that we'll be using. We'll be using CMake for this, which generates Unix Makefiles to control the build process[5].

```
$ cd ~/avr
$ mkdir build
$ cd build
$ cmake -G 'Unix Makefiles' -DLLVM_LINK_LLVM_DYLIB=ON -DLLVM_TARGETS_TO_BUILD="X86" \
    -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=AVR ../llvm
```

Up to this point, the commands listed above have already been executed in your home directory. The next step is to kickstart the build process which will create the compiler for you to use during this assignment. To make sure we change to the build directory that we create above[6] and then kick make into action to get things to build. This building process takes a long while so use the time to have a quick look at the remainder of the assignment and consider using this time for doing the Makefile tutorial.

```
$ cd ~/avr/build
$ make
```

If at any point you need to kill the process press ^C (Control-C). Make allows you to continue building later on by running `make` again, it will figure out where it was in the process and continues automatically[7].

[3] Using the `cd avr` command

[4] LLVM already takes up quite a significant amount of diskspace without all of its history

[5] If you haven't worked with Makefiles yet then consider doing the Makefile tutorial on `https://makefiletutorial.com/`, while waiting for some of the commands in this section to finish

[6] You should already be there if you had run the commands above but will still need to change into this directory if you just logged in.

[7] You can also start a background session by using a program like `tmux`. To get more info on `tmux` have a look at the manual page, you can get that by running `man tmux`.

2 Compilation flow

The next step in our process is to compile the application with the new LLVM compiler that we just built in your home directory on the server[8]. This compiler implements the front-end, optimization layer, back-end, and an experimental linker. However, for this assignment, we will use the linker from AVR GCC to complete the compilation process.

2.1 Creating the object file(s)

To compile the source code of our application into an object file we execute the following command[9]:

```
$ 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`.

[8] 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.

[9] 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.

2.2 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 provides an experimental linker, but for this assignment we'll use GCC's linker (GNU ld) instead. 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[10], shows us the implementation of the `_delay_ms` function and contains some extra documentation of that function in the comment just before it[11]. According to this documentation we can define `__DELAY_BACKWARD_COMPATIBLE__` in our source file[12] 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?

[10] Which was `/usr/lib/avr/include` as we told to the compiler with the `-I` option.
[11] Feel free to have a look, it's on lines 104–140 of the file.
[12] 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[13] there exists a tool called `objdump`, or in case of our AVR flavor `avr-objdump` which allows us to inspect ELF[14] 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 let's 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 .-2` instruction on offset 0x1e will jump to the same offset.

- 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 .+0` 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.

[13] 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.
[14] Reading material: `https://en.wikipedia.org/wiki/Executable_and_Linkable_Format`

```
26 .Ltmp0:
27     sbiw    r26, 1
28     brne    .Ltmp0
```

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

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 (`Ltmp0:`) and puts it before the subtract instruction that adjusts the loop counter.
- From the instruction set manual we know that the arithmetic 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[15].
- The branch uses the branch target `.Ltmp0`.[16]

All together this code should assemble to work correctly, so what is happening here? Going back to our analysis of the compilation results, we see that `avr-objdump -d led.o` shows 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[17] shows a flag named `RELOC`. This tells us that there are addresses which have not been resolved yet[18] 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.

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?

[15] 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.

[16] Alternative branching patterns exist. For example, `branch 1b` would mean "branch backward to the last local label with identifier 1", while `brne 1f` translates to "branch-no-equal to the first label with identifier 1".

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

[18] 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.

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 major 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).

3 Improving code generation

In the previous part, we managed to compile and link the blinking light application into `led.elf`, and we analyzed the assembler code to check if the program should still work correctly. Before we will continue with our exercise we will first check if the program did indeed work correctly.

3.1 Running in the simulator

Since our servers don't have Arduino boards connected[19], we will be using a simulator to verify the behaviour of our program. The simulator that we have selected for this course is based on `simavr`[20] and is configured to generate a trace of the output port of our processor. However, in order to use this simulator, we will need to adapt the C program a bit so that some information on the processor version and clock frequency is embedded in the ELF object file. Listing 3 shows how this is done. Lines 8–10 have been added and make sure that the required information is added to the ELF file. For your convenience we have also provided a Makefile[21] for this project in Listing 4 which automates the build process and add a `run` target to demonstrate the loading of the program into the simulator.

Exercise 9

Add the Makefile to your work directory and make the appropriate changes to the `led.c` file. You can now run `make all` to build the `led.elf` executable. Inspect the new `led.elf` file with `avr-objdump` and you will find that a new section has been introduced to the file[a].

- What is the name of this section?
- Does the addition of this section change the behaviour of the program when it is executed on hardware?

[a] You can find out more about the contents of a specific section of an ELF file by using the `-j` *<section name>* and `-S` options. Which will show you the contents of the named section.

Ok, so you have successfully compiled the program for usage with our simulator. Using the Makefile to run the simulator you should now be able to see the following output:

```
$ make run
simulator_cli led.elf
Loaded 184 .text at address 0x0
Loaded 0 .data
firmware led.elf f=16000000 mmcu=atmega328p
Hit any key to exit the simulator.
Cycle count: 0000000081473531 Pin changes: 00000006
```

As the simulator already states, you can hit any key to stop the simulator but it's good to wait a while to see the *pin changes* number go up a several times. This number indicates how often the pin to which our LED is connected has changed in value. That the change is indeed happening is a good indication that our program is still working properly but to be sure we will also inspect the timing of our program on the simulated processor.

[19] You still wouldn't be able to see the LED blink if they had.
[20] `https://github.com/buserror/simavr`
[21] Which you should be able to understand if you have completed the Makefile tutorial.

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

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

/* Code required by simulator. */
#include <avr_mcu_section.h>
AVR_MCU(F_CPU, PART);

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 3: *The led.c program adapted to work with our simulator.*

```
PART:=atmega328p
CC:=clang --target=avr
CFLAGS:=-Wall -Os -mmcu=${PART} -I/usr/lib/avr/include/ -I/home/pcp17/material/include/
     -DPART=\"${PART}\"
LDFLAGS:=-mmcu=${PART}
ASFLAGS:=-mmcu=${PART}
PROGRAM:=led

.PHONY: all run clean

all: ${PROGRAM}.elf

${PROGRAM}.s: ${PROGRAM}.c
    $(CC) $(CFLAGS) -o $@ -S $^

${PROGRAM}.o: ${PROGRAM}.s
    avr-as ${ASFLAGS} -o $@ $^

${PROGRAM}.elf: ${PROGRAM}.o
    @# Use driver to figure out right linker flags.
    avr-gcc -o $@ $^ ${LDFLAGS}

run: ${PROGRAM}.elf
    simulator_cli $^

clean:
    rm -f ${PROGRAM}.elf ${PROGRAM}.o gtkwave_output.vcd ${PROGRAM}.s
```

Listing 4: *A Makefile to organize the build process and testing of the blinking LED application.*

Exercise 10

After you have quit the simulator you can see that there is a new file in your work directory called `gtkwave_output.vcd`. This file contains a timed trace of the output of port B of the AVR processor, the port to which our LED is connected. You can use `gtkwave` (shown in Figure 1 to view the waveform and inspect the timing. The command is as follows:

```
$ gtkwave gtkwave_output.vcd
```

When the `gtkwave` screen opens[a], click on *logic* in the top-left side pane, select *portb[0:7]* in the bottom left side pane, and click the insert button at bottom. This adds the trace of our port B to the view area. You can now use the magnifying glass with the minus sign inside in the top toolbar to zoom out[b] until you start to see the signal changing on a regular basis. Does the blinking interval now match with the one given in the C file?

[a] Make sure you have X forwarding enabled as was explained in the *Tool Setup* tutorial.

[b] You will need to hit this button quite a few times to get there. Alternatively, you can also use the button with the magnifying glass to zoom out and view the whole trace in one step.

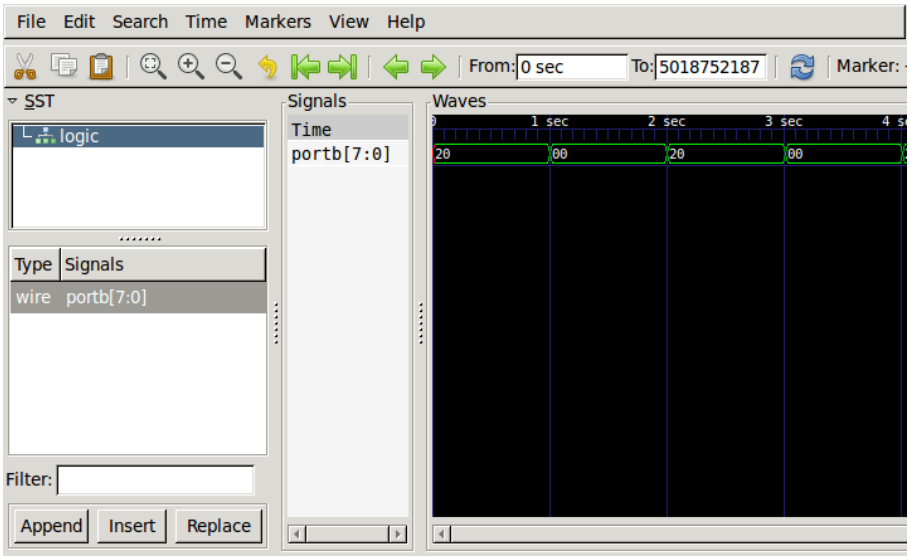

Figure 1: *GTKWave showing the trace of our application*

3.2 The problem of the day

Yay, it worked! But can we make this work even better?

It seems we can. If you inspect the assembler output more closely you can see a pattern that appears two times, once for each delay loop. Our loop counter for the outer loop of the delay code is a 16-bit counter which gets decremented in line 30, a two-step comparison of the counter value with 0 then follows in lines 31–32, and finally a branch is done based on this comparison. All together this is a valid solution for iterating over our loop but it can be done more efficiently. The `sbiw` instruction, like most arithmetic operations, already sets the flags register based on the value of the result compared to zero. So the compare is already done implicitly and is completely unnecessary here[22]. Let's get rid of it and start improving LLVM!

[22] You can remove them from the assembly file and re-run the application in the simulator if you don't believe us. Or alternatively you can read the description of the `sbiw` operation in the AVR instruction-set manual.

```
30        sbiw    r30, 1
31        cp      r30, r20
32        cpc     r31, r21
33        brne    LBB0_2
```

Listing 5: *The superfluous compare operations that can be removed without penalty*

To make sure that we don't accidentally break another part of the compiler and indeed have fixed this issue we will start by adding a small test that demonstrates our problem. Adding this test to the AVR backend testbench will also make sure that no-one else will accidentally be breaking our improvement in the future.

LLVM stores its tests in the `test` subdirectory of the source tree[23], since we are working on the code generation for the AVR backend we will put our new tests in the `CodeGen/AVR` subdirectory[24] of the test set. As you can see in this directory, each test consists of a `.ll` file containing the IR code for testing that specific bit of the code generation. Using IR code here ensures us that there are fewer optimizations that can interfere with our test cases which makes it more likely that we are actually testing the behaviour that we would like to verify.

To run the tests we use LLVM's integrated tester, called `llvm-lit`, and combine it with `FileCheck` to check the output of our tools. To do this change into the build directory[25] which you can find in the `avr/build` subdirectory of your home.

Exercise 11

To run the testsuite, or in this case the part of the testsuite that we are interested in use the following commands:

```
$ cd ~/avr/build
$ llvm-lit test/CodeGen/AVR
```

This will produce a line of output for each of our testcases, stating if it was successful, if it was unsupported in our configuration, if it failed, or if it was expected to fail[a], followed by a nice summary.

How many expected failures are there currently?

[a] Expected failures are useful to keep track of known broken things, this is usually called test-driven development: `https://en.wikipedia.org/wiki/Test-driven_development`.

3.3 Extracting a test-case

OK, so how does such a test-case look? Listing 6 shows us an example. I have added two tests, one checking the 8-bit version of our problem and a second checking the problem that we have uncovered with our program. Lines starting with a `;` are comments in the IR and some of these comments get interpreted by `llvm-lit` and `FileCheck`. In particular, the `RUN:` line will tell `lit` which command to run, and the `CHECK` lines tell `FileCheck` to test the output for presence, or absence, of the given patterns.

In this case, we use `lit` to run LLVM's code generation tool `llc` so that we avoid all of the optimization steps from the other compiler layers, it takes as input %s, which is short for the current test file, and

[23] You can find the source tree for our version of LLVM in your home-directory's sub-directory `avr/llvm` on the server.
[24] Making the complete location `$HOME/avr/llvm/test/CodeGen/AVR`.
[25] This is where we have configured make to run and put all of the output files so they don't interfere with the source code.

runs `llc` with the argument `-march=avr` to instruct it to target AVR. We then send the output of this tool to `FileCheck`, which compares it to the check lines in this file (again provided using `%s` as argument).

```
; RUN: llc < %s -march=avr | FileCheck %s

; CHECK-LABEL: @nocmp8
; CHECK: sub
; CHECK-NOT: cp
; CHECK: brne
define i1 @nocmp8(i8 %a, i8 %b) {
    %diff = sub i8 %a, %b
    %cmp = icmp ne i8 %diff, 0
    ret i1 %cmp
}

define void @nocmp(i16 %a) {
entry:
    br label %l.loop
l.loop:
    %a.phi = phi i16 [ %a, %entry ], [ %a.new, %l.loop ]
    %a.new = sub i16 %a.phi, 1
    %cmp = icmp ne i16 %a.new, 0
    br i1 %cmp, label %l.loop, label %l.end
l.end:
    ret void
}
```

Listing 6: *Two testcases for checking there are no compare operations when we don't need them.*

Exercise 12

The tests in Listing 6 only check the first function. Copy this listing into a file called `nocp.ll` in the AVR CodeGen tests directory and add the missing tests. Remember, you can use `llc` to check the current output assembly for this test using:

```
$ llc -march=avr < nocp.ll
```

Re-running `llvm-lit` from your build directory again will now show you have one failing test. To mark your test as expected failure[26] you can add the following line at the top of the file:

```
; XFAIL: *
```

3.4 Detecting instruction patterns

The AVR backend of LLVM is found, just like those for the other targets, in the subdirectory `lib/Target` of the LLVM source tree. In the AVR backend subfolder you will find several `.td` files, which contain the target definition in LLVM's TableGen DSL. We can use the TableGen language to select combinations of instructions similarly to selecting individual instructions. This works as long as the pattern is relatively simple, with the main constraint being that it only works for patterns that have a single output in the graph. Which is sufficient for us since the output that is of interest to us is the status register set by the compare operation and we mainly want to remove the compare operation. These

[26] This will mark this test as expected to fail for all platforms (the * matches the platforms for which this is broken).

patterns live, just like the normal instruction patterns, in `AVRInstrInfo.td`. Let's start by looking at some of the existing patterns, you can find them at the end of the file.

A funny one is the one that is given for our `sbiw` instruction which we will be needing for our own patterns. You can find this one on line 2031 and in Listing 7. The first part of the pattern defines the pattern we want to match and the second part represents the replacement after instruction selection. This means that we can use the same kind of operations to select our patterns as we did in the instruction selection for the `cpi` instruction, but that we should produce target specific versions of the operations as output of our pattern.

```
2031 def : Pat<(add IWREGS:$src1, imm0_63_neg:$src2),
2032          (SBIWRdK IWREGS:$src1, (imm0_63_neg:$src2))>;
```

Listing 7: *The definition of the `sbiw` instruction selection pattern.*

So, why take this `sbiw` instruction as an example? That's because it is special and shows us something that has happened during the earlier optimizations. Looking at the pattern we see that we match an *add* operation and not a subtraction. This is because of the normalization that the compiler applies to the operation graph before entering instruction selection. Usually there are many ways of writing code that does basically the same thing. To avoid having to detect all possible variations of a pattern the compiler applies a normalization step. The effect of this step is that common operation patterns get changed into a standardized form. In this case the addition and subtractions with immediate values have been replaced using only additions. Basically $a - k$ has been replaced with $a + (-k')$. So we will need to take this into account when selecting the subtract instruction. Secondly, operations using an immediate value as input are usually limited in how big this value can be. For the `cpi` instruction this was 8-bits as we didn't need to encode much more than a target register, but for the `sbiw` instruction this range is smaller. In this case we can only use our operation if the immediate value is between 0 and 63. The `imm0_63_neg` in the first part of the pattern definition is a pattern leaf which detects if this is the case for our operand[27], the `imm0_63_neg` in the second part applies the transformation of $(-k)$ into k using the `imm16_neg_XFORM` defined on line 80 of the same file. Finally, you can also see that a special register class is used for the inputs of this operation. The `cpi` operation accepted all 8-bit registers as input and used the GPR8 class for it's input selection. The `sbiw` instruction on the other hand is much more restrictive and allows only a limited number of registers at its arguments. These register sets are defined in `AVRRegisterInfo.td`. All together these constraints result in the `sbiw` operation being selected with the proper operands and when it is possible.

Exercise 13

Open the `AVRRegisterInfo.td` file and check which register pairs are available as arguments for the `sbiw` operation.

Now, with this knowledge we should be able to add the patterns for our improvements to the file. At the end of `AVRInstrInfo.td` we can add the following code to fix the 8-bit case as is shown in Listing 8.

```
2096 // Fix arithmetic operations followed by compare 0 to skip the compare and use
2097 // the SREG values already present
2098 def : Pat<(AVRcmp (sub GPR8:$src1, GPR8:$src2), (i8 0)),
2099          (SUBRdRr GPR8:$src1, GPR8:$src2)>;
```

Listing 8: *The added pattern for avoiding comparisons in the 8-bit case.*

[27] It is defined on line 85 in `AVRInstrInfo.td`.

That is all. We have a normal subtraction with two register operands in our test-case so we can just match the following:

- A compare using `AVRcmp` like we did for the `cpi` instruction.
- A subtraction of two input registers.
- An immediate with value 0.

And we replace that with just the `SUBRdRr` instruction which is the one that subtracts two registers that we found on line 424.

Exercise 14

Add the missing instruction selection pattern for the remaining 16-bit test case based on those of the 8-bit test case and the other patterns available in `AVRInstrInfo.td`. After that, change into the build directory and build LLVM. Since we're now only using and testing `llc` it may be wise to only build that part of the project[a]. You can achieve this using the command `make llc`. Finally, re-run the test suite to verify your pattern works as expected, while making sure no other test cases are now suddenly broken.

[a] Doing a full build of the entire compiler each time will take quite a bit more time than just building the tools that you need for testing.

Great, so you found your first bug in the compiler, created a test, and managed to fix the problem. Let's submit the result! Submitting code changes to such a big project as LLVM requires review from the project maintainers to make sure that the quality of the compiler keeps improving. To keep this review manageable you are only allowed to submit a description of the changes you made in the form of a patch.

Exercise 15

Before we continue with the exercises and submit your patch using git we will need to do a one-time configuration of git. Most importantly, you will need to tell git about your name and email address it will put in the changeset message to track who fixed, changed, or broke, what. Change `Your Name` and `your-email` in the next lines to match your name and student email and execute the commands on the server.

```
$ git config --global user.name "Your Name"
$ git config --global user.email your-email@student.tue.nl
```

Exercise 16

Change back to your LLVM source tree, this folder is under version control using `git`. You can type the following command to get an overview of the changes that have been made:

```
$ git status
```

It should tell you that you have changes to your `AVRInstrInfo.td` file and that there is one untracked file now (the patterns to avoid the extra `cp` operations and your corresponding test cases).

To prepare our changes for admission we first need to select the changes that we want to have included in our changeset. This is done using the `add` method of git and giving it the file with changes that we would like to add. Make sure to keep patches small so that they are easy to check by others and keep changes for different problems in separate changesets. For now, lets start by making a patch for fixing the redundant comparisons:

```
$ git add lib/Target/AVR/AVRInstrInfo.td
$ git add test/CodeGen/AVR/nocp.ll
```

If you now run `git status` again you will see that both your changed files are now staged to be committed. However, before we can commit anything we should first check if our changes are indeed the ones we intended to commit. We can check the contents of our prepared changeset using the following command:

```
$ git diff --cached
```

Committing the changes is the next step indeed. You can do this with the `commit` method of git. If you use this without arguments it will open up a text editor for you so that you can provide your commit message. This commit message is a short description of your changes and is required for git to accept your changeset as a commit[a]. Run the following command, enter a meaningful message, save it, and quit the editor. This should complete this step.

```
$ git commit
```

Your changes will now be permanent in your version of the source code repository and you can now create a patch and submit it for review. This last step is done using the `format-patch` method of git and requires you to specify a version from where to start making patches. The latest version in your repository is called `HEAD`, earlier versions can be described using the `~` symbol and a number. Use the following command to format a patch for your last committed change.

```
$ git format-patch HEAD~1
```

This will create a new text file in your current directory, its name starting with a number `0001` followed by the first part of your description. You can submit the contents of this file for review, in this case to our Canvas website.

[a] Alternatively you can use the `-m` option of `git commit` to supply your commit message directly from the command line.

OK, that's it for this part of the exercise! You've fixed the compiler and submitted your first patch for review.

The assembly code for the first loop of the target application should now look like Listing 9.

```
17    sbi 5, 5
18    movw   r30, r24
19 LBB0_2:                                ; %while.body.i
20                                        ;   Parent Loop BB0_1 Depth=1
21                                        ; =>  This Inner Loop Header: Depth=2
22    movw   r26, r18
23    ;APP
24 .Ltmp0:
25    sbiw   r26, 1
26    brne   .Ltmp0
27    ;NO_APP
28    sbiw   r30, 1
29    brne   LBB0_2
```

Listing 9: *The newly improved assembler code*

Trick question answers

Trick question 1: Otherwise we are sure to break this test once we fix the subtract followed by compare-with-zero later.

4 Introducing a new intrinsic

In this third and final part, we will try to further improve the code generation capabilities of LLVM by adding support for the `__builtin_avr_delay_cycles` intrinsic. As you will find, this change touches code on many levels of the compiler.

To introduce a new intrinsic, we'll first need to define the intrinsic in the LLVM framework, then register it with clang to get frontend support for using it in C code. This allows us to use the intrinsic in our C code but we didn't define how to handle it yet so it will cause all kinds of funny crashes. From that point we'll need to add code in several of the code generation stages to get to the point where we can actually generate the right instructions. All together this will take quite a few steps, so lets get started.

4.1 Declaring the intrinsic to LLVM

So, how to get started? The LLVM documentation contains a nice page[28] that explains the steps for introducing a new intrinsic to LLVM. Apparently this is done by adding its definition to the target intrinsics in `include/llvm/IR/Intrinsics*.td`. However, looking at the files in that folder shows us that we don't have an `IntrinsicsAVR.td` yet, so we'll need to create that from scratch. The other directions given by the documentation are that we should *1)* add some documentation for our new intrinsic, *2)* add constant-folding rules (if applicable), and *3)* add a test-case to check if we don't break anything later on.

Exercise 17

Great, that sounds like a relatively easy point to start with. We can copy the file header from `IntrinsicsARM.td` in the same folder in our includes as a starting point, that should give us the smallest effort of editing to do to make it match our AVR target. We'll also need to make sure that we include our new `IntrinsicsAVR.td` file in the main `Intrinsics.td` file so that the framework knows of its existence. Now, all that remains is for us to add the definition of our intrinsic to the file.

This definition should look as follows:

```
let TargetPrefix = "avr" in { // All intrinsics start with "llvm.avr.".

def int_avr_delay_cycles : GCCBuiltin<"__builtin_avr_delay_cycles">,
    Intrinsic<[], [llvm_i32_ty], []>;

} // end TargetPrefix
```

We start by defining the `TargetPrefix` for our architecture, this is a part of the file header that you can copy from the ARM example but make sure to rename the prefix to `avr`. After this prefix we define our new intrinsic. LLVM already has a nice way of specifying intrinsics definitions that have been copied from GCC variations of the compiler for our target architecture. We first define a name for the node in our internal representation `int_avr_delay_cycles`, followed by providing the name for the GCC builtin `__builtin_avr_delay_cycles`, and finally we provide a description of the function signature (i.e. how it should be called). This signature is composed of three parts, the return type (void) `[]`, the argument type(s) (i32) `llvm_i32_ty`, and a set of flags that give hints on how to handle this intrinsic during optimization (no optimizations allowed for our case).

Ok, that should be it, you can complete these changes an see if everything still compiles by rebuilding LLVM from your build directory. This will take some time since we've messed around

[28] `http://llvm.org/docs/ExtendingLLVM.html`

in a core component and there will be many dependant parts which now need to be rebuilt to support our new intrinsic. You can prepare a patch for these changes for the quiz while its building.

That's it, now LLVM shouldn't complain about an unknown intrinsic when we try to use it later on.

4.2 Adding frontend support

Which brings us to the 'using it' point. It would be nice if we could use this intrinsic in our C code, which means we will also need to inform the frontend about its existence. This part is quite similar to the previous step but would also require you to add some further bits of code since the AVR backend didn't support any of these builtin functions yet. We've decided to provide these changes to you as a nice patch which you can find on the Canvas website together with this description.

Exercise 18

So you got a patch? Let's see how to apply it. In this case it's a patch to LLVM's frontend, `clang`, which lives in its own repository in the LLVM source tree. But before we start, copy the patch file `exercise18.patch` into your home directory on the server. In the commands here I will assume that you have copied it into the top level directory of your home directory, adjust the commands accordingly if you put it someplace else.

First we check what the patch plans to change and if it will apply cleanly. If it does we will then apply it and add our own signature to tell others that you were the one who OK-ed the change.

So, to check a patch, change into the target repository, see which files the patch wants to change, and check if the patch will apply without complaints...

```
$ cd ~/avr/llvm/tools/clang
$ git apply --stat ~/exercise18.patch
$ git apply --check ~/exercise18.patch
```

The `--stat` option should show you a short list with the files that are changed and the `--check` option should complain about all the problems[a] that will be encountered when merging this patch into your version of the code.

Hopefully you didn't get any complaints from `git apply --check` and we can now believe we trust the integrity of this patch so we can apply it. Applying a patch from your mailbox is done through `git am`. You can use the `--signoff` option to automatically put your signature under the commit message so that others can see that it was you who OK-ed the change.

```
$ git am --signoff ~/exercise18.patch
```

OK, that's it, a quick question for the quiz though, which files did we change?

[a] There shouldn't be any at this point.

Yay, both our frontend and the framework now[29] know about the intrinsic so we can go back to the backend and make sure that it will now 'do the right thing' when it encounters the intrinsic.

4.3 Code generation

Good, so let's see how far this will get us and where we now need to start fixing problems. Running `make all` from our target application now demonstrates that we have indeed managed to break the

[29] Well, technically only after you've rebuilt the compiler with this last change.

compiler. Or at least, it should if you have removed the __BUILTIN_DELAY_BACKWARDS_COMPATIBLE define from the code and rebuilt your compiler to include the updates.

The message we get with the crash is as follows:

```
ExpandIntegerOperand Op #2: t10: ch = llvm.avr.delay.cycles t7, TargetConstant:i16
    <375>, Constant:i32<16000000>

Do not know how to expand this operator's operand!
UNREACHABLE executed at /home/rjordans/avr/llvm/lib/CodeGen/SelectionDAG/
    LegalizeIntegerTypes.cpp:2604!
...
```

Where the ... represent a whole lot more output in the form of a call trace that shows us how we got here.

It looks like we've found our first problem. The problem is occurring in the `LegalizeIntegerTypes.cpp` file, which tells us that this problem is appearing during type legalization. The first step of the lowering process. The error message above also gives us the intrinsic line in the code and tells us that it doesn't know how to expand integer operand #2, which is the `i32` argument of our intrinsic. That makes sense, the AVR architecture usually doesn't do operations with `i32` arguments so it is perfectly reasonable for it to not support those. However, we now have an operation that does want one and the backend is confused.

Exercise 19

If you manage to break something then it will also be your task to fix it. Let's give it a go. But this having to use clang and our target application to get to our crash site isn't really nice so we'll start by creating a new test for it.

This test should contain the following:

```
1 ; RUN: llc -march=avr < %s
2
3 define void @test() {
4   tail call void @llvm.avr.delay.cycles(i32 16000000)
5   ret void
6 }
7
8 declare void @llvm.avr.delay.cycles(i32)
```

Add the test to your AVR tests and check if it provides you with the following output.

```
$ llc -march=avr < test-delay.ll
        .text
        .file   "<stdin>"
ExpandIntegerOperand Op #2: t3: ch = llvm.avr.delay.cycles t0, TargetConstant:i16
    <375>, Constant:i32<16000000>

Do not know how to expand this operator's operand!
UNREACHABLE executed at /home/rjordans/avr/llvm/lib/CodeGen/SelectionDAG/
    LegalizeIntegerTypes.cpp:2604!
...
```

Ok, so how do we test this pattern later on in a way that the test won't break if we make some small changes in the actual operations that we output for this intrinsic or if the allocation of register values is changed?

Right, with our shiney new test in hand we can now use llc to do our backend debugging and development again without having to bother with all the extra layers of the compiler frontend and optimizations messing about.

4.4 Type legalization

Our error message told us that the compiler doesn't know how to expand the `i32` operand of our intrinsic. Which means we'll need to teach it either how to do that, or that it is allowed to keep it in it's current form. The second option sounds nice, let's do that and tell the lowering to allow `i32` which are used by an intrinsic.

Exercise 20

This requires us to add some code in `AVRISelLowering.cpp`[a], which defines a lot of our custom lowering bits in a nice central place. Add a custom lowering for the `Constant` node with `i32` type.

```
122   // Allow i32 constants as argument of intrinsics
123   setOperationAction(ISD::Constant, MVT::i32, Custom);
```

That should allow the constant number creation to pass into the next stage and end up at the `LowerOperation` method on line 667 of the same file. In this method you will see the following lines:

```
669   default:
670     llvm_unreachable("Don't know how to custom lower this!");
```

Remove them, they don't belong here! The `return SDValue()`[b] at the end of this function is there to signal to the LLVM framework that we didn't have a custom lowering for our node after all and allows us to use the pre-existing lowerings as a fall-back option. We want to be able to use that for our operation.

To complete the lowering of the `Constant` node we will now need to add the part that decides if we should keep it or if we should split it. This is done in the function `ReplaceNodeResults`, here we add a new case to the switch statement that checks if we are working on a constant. If we are then we will first need to check if there is a *use* of this operation that is an intrinsic call. If we don't have any then we will decide to split the current constant into two parts, while if we do have an intrinsic node as user we will tell the backend that it's OK to keep the constant node by translating it into a `TargetConstant`.

We've given you a most of the required code to start with in Listing 10, try to complete it[cde].

[a] You can find this file with the other parts of the backend in `lib/Target/AVR`.

[b] You will see in the lowering code that most of it works on `SDNode` and `SDValue` elements, these elements are IR components that have made it into the `SelectionDAG` layer. Operations here are still mostly in the IR format, which have their opcode defined as part of the ISD namespace, but can be extended with custom nodes for the target architecture, in our case from the AVRISD namespace. You've seen examples of this in the instruction-selection lecture. For example, `ISD::LSL` is the code for the IR node version of the `shl` operation, while `AVRISD::LSL` is the node that has been legalized for the AVR backend.

[c] Hint: You can get a new constant using `DAG.getConstant(Value, DL, Type)` where the value and type are something for you to figure out. The DL field is keeping track of the debug location so that we can still have some idea on where this value was defined in the original C code.

[d] Hint: Creating a new `TargetConstant` works mostly the same as a `Constant` with the main difference being that you need to use `DAG.getTargetConstant()`.

[e] Hint: The new results of this operation should be put into the Results vector using `Results.push_back()`, as you can see from the other cases in the switch statement.

```
713   case ISD::Constant: {
714     assert(N->getValueType(0) == MVT::i32 && "Expected Constant<i32>");
715     // Lower Constant<i32> into a TargetConstant iff its use is an intrinsic
716     //  otherwise split into two half-parts to allow further lowering
717     bool split = true;
718     for (const auto use : N->uses()) {
719       if (use->getOpcode() == ISD::INTRINSIC_VOID) {
720         split = false;
721       }
722     }
723
724     const uint32_t C = cast<ConstantSDNode>(N)->getZExtValue();
725     if (split) {
726       // Split into Lo16, Hi16, low part first
727       // FIXME
728     } else {
729       // Lower into a TargetConstant for __delay_builtin_ms
730       // FIXME
731     }
732     break;
733   }
```

Listing 10: *The custom lowering snippet for the Constant node.*

With the addition of this new code you can again compile llc and check if it works. This time it should get passed the type legalization stage and up to our next point of failure. It seems that this next bump is going to be instruction selection. No crash this time, just a fatal error.

```
$ llc -march=avr < test-delay.ll
        .text
        .file   "<stdin>"
LLVM ERROR: Cannot select: intrinsic %llvm.avr.delay.cycles
```

4.5 Instruction selection

Next step, we've made it through the legalization stages and have come to the instruction selection part. But what to select now? The goal of our intrinsic is to insert a delay loop in our program, that sounds like a bit much to just do in an instruction selection pattern... Luckily for us we have a nice example on how to do this already in the AVR backend which we can use for inspiration! The shifting operations of the AVR architecture only support single step shifts. As a result the AVR backend needs to insert a new loop if it wants to support shifting by a variable amount. We can use that as our example. In the `AVRInstrInfo.td` file we find, on line 540, the pattern that is added for selecting the MUL operation. The block in which this operation is defined starts with `let useCustomInserter = 1`, which tells us that these use a custom instruction selection emitter. Such a custom emitter is useful when you want to have late additions to your selected instructions but still want to do them before getting started on scheduling and register allocation. Just where we would like to have our code emitted as well.

Exercise 21

We can now insert a pseudo operation definition for our intrinsic in a similar way. You can use the following snippet to add in your code.

```
let Defs = [SREG] in
```

```
def DelayCycles : Pseudo<(outs),
                         (ins i32imm:$delay),
                         "# DelayCycles PSEUDO",
                         [(int_avr_delay_cycles i32:$delay)]>;
```

This change should allow us to pass into the final stage of our changes, actually emitting operations for our intrinsic.

Great, almost there! The only part that's still missing is the pseudo operation expansion where we'll get to generating the new loop for our intrinsic. If you try to run your new version of llc with our test-case at this point then you will see that we've got ourselves into a new crash of the compiler. This time to trace starts with an assertion failure in `AVRTargetLowering::EmitInstructionWithCustomEmitter`, so that's the place where we need to start fixing stuff.

Exercise 22

Looking at the `EmistInstructionWithCustomEmitter` function shows us that there are some examples we can follow. Both the shift and multiplication operations already get detected in a switch statement at the top from which they are handled in their own functions. I've prepared a patch for you that does the same for the delay intrinsic. It's an incomplete patch so we'll apply it in a different way than we did with the frontend patch.

This time[a] you don't use the `git am --signoff` command, the changes aren't in a state that we want to have them permanently in our changelog, but use the following[b].

```
$ git apply ~/exercise22.patch
```

The code that I've provided you with adds a skeleton for the `insertDelayCycles` function which you'll need to complete. We now need to work with *machine instructions* since we're now running after instruction selection. You can have a look at the insertion routines for multiplication and shifting operations in this same file if you want to get some more hints on how to build new machine instructions. The pattern isn't too complex, you call `BuildMI` with the basic block to which you want to add the operation as its first parameter, the second is our debug location `dl`, and the third is the operation we want to insert. As you can see from the examples in this file you can get the third operand of `BuildMI` by using `TII.get(AVR::name)` together with the name of the instruction from our `AVRInstrInfo.td`.

Have fun!

[a] After checking the integrity and contents of the patch.
[b] I'm assuming again that you copied the patch file into your home directory.