Assignment 1b: Compiler organization and backend programming

Roel Jordans

2016

Organization

Congratulations, you have completed the first part of this assignment. If not, you should. This second part of the assignment continues where we stopped last time. Today we will have a closer look at the generated assembler code and we will start implementing some small optimizations in the AVR backend.

2 Improving code generation

Last time 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.

2.1 Running in the simulator

Since our servers don't have Arduino boards connected[1], 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`[2] 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 1 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[3] for this project in Listing 2 which automates the build process and add a `run` target to demonstrate the loading of the program into the simulator.

Exercise 1

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.

[1] You still wouldn't be able to see the LED blink if they had.

[2] `https://github.com/buserror/simavr`

[3] 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 1: *The led.c program adapted to work with our simulator.*

```
PART:=atmega328p
CC:=clang --target=avr
CFLAGS:=-Wall -Os -mmcu=${PART} -I/usr/lib/avr/include/ -I/home/pcp16/material/include/
     -DPART=\"${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 2: *A Makefile to organize the build process and testing of the blinking LED application.*

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.

Exercise 2

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.

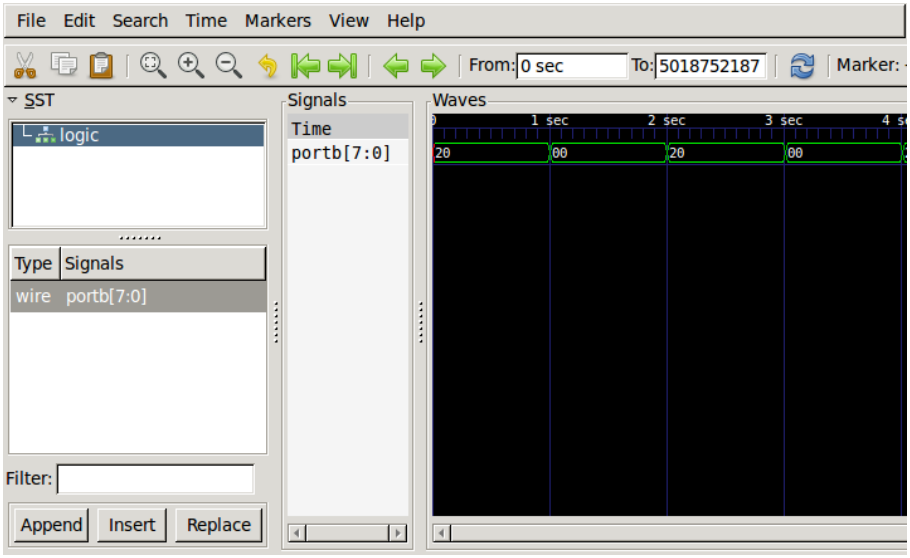

Figure 1: *GTKWave showing the trace of our application*

2.2 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 29, a two-step comparison of the counter value with 0 then follows in lines 30–31, 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 efficient. 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[4]. Let's get rid of it and start improving LLVM!

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

Listing 3: *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[5], since we are working on the code generation for the AVR backend we will put our new tests in the `CodeGen/AVR` subdirectory[6] 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[7] which you can find in the `avr/build` subdirectory of your home.

Exercise 3

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`.

[4] 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.

[5] You can find the source tree for our version of LLVM in your home-directory's sub-directory `avr/llvm` on the server.

[6] Making the complete location `$HOME/avr/llvm/test/CodeGen/AVR`.

[7] 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.

2.3 Extracting a test-case

OK, so how does such a test-case look? Listing 4 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 are 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 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 4: *Two testcases for checking there are no compare operations when we don't need them.*

Exercise 4

The tests in Listing 4 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[8] you can add the following line at the top of the file:

```
; XFAIL: *
```

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

2.4 A side quest into instruction selection

Looking at our 8-bit test we can find another point of our compiler that is up for improvement so let's fix that as well and use that as an example on how to submit a change for evaluation. The AVR architecture also has an instruction for comparing an 8-bit value with a small immediate but looking at the output of `llc` on our new test-case we find that it does a `ldi rtemp, 0; cp r, rtemp` combination instead. Again, we start by adding a small test-case to prove our point and demonstrate when the issue has been fixed. This time the test-case is the one listed in Listing 5 and it is only a small variation from the previous one. The main difference is that we now do not have the subtraction anymore in the test.

```
1 ; RUN: llc < %s -march=avr | FileCheck %s
2
3 ; CHECK-LABEL: @cpi
4 ; CHECK: cpi
5 ; CHECK: brne
6 define i1 @cpi(i8 %a) {
7     %cmp = icmp ne i8 %a, 0
8     ret i1 %cmp
9 }
```

Listing 5: *Test if the `cpi` instruction gets selected correctly.*

Trick question 1

Why do we need a new test for this? Our previous test was already capable of demonstrating the problem...

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 there you will find several `.td` files, which contain the target definition in LLVM's TableGen DSL. Since we're now concerned with the instruction selection part we will start by looking at the `AVRInstrInfo.td` file, which contains all the instruction definitions. In this file you can search for `cpi` which will lead you to line 870, on which this instruction is already defined as can be seen in Listing 6.

```
868 // CPI Rd, K
869 // Compares a register with an 8 bit immediate.
870 let Uses = [SREG] in
871 def CPIRdK : FRdK<0b0011,
872                 (outs i8imm:$k),
873                 (ins GPR8:$rd),
874                 "cpi\t$rd, $k",
875                 []>;
```

Listing 6: *The definition of `cpi` in `AVRInstrInfo.td`.*

Ok, so there's our problem, that definition doesn't have a pattern (the last part between the square brackets) and there are some other mistakes as well. Just for completeness, the three main mistakes here are; *1)* `cpi` doesn't use the status register as input (line 870 states differently) but defines it as output, *2)* an immediate value is not an output of `cpi`, it's an input, and *3)* as noted above, this instruction is missing a selection pattern.

For suggestions on the correct patter we quickly look above to the other compare patterns and see that we can match to a `AVRcmp` node for the compare instructions. It also shows us how we define

the status register as implicit output as part of the instruction pattern. The only thing that we need to figure out still is the right input types for our operation. In general, for the pattern we can use the register classes, as defined in `AVRRegisterInfo.td`, as well as, `imm` to denote immediate values in the pattern part. Looking more closely at the instruction description though shows us that for the `ins` we use a different notation for the immediate values. This is because we need to check here if the size of the immediate is valid as an input for our pattern, in this case to see if it fits in an 8-bit integer.

Putting these fixes together results in the code shown in Listing 7.

```
868 // CPI Rd, K
869 // Compares a register with an 8 bit immediate.
870 def CPIRdK : FRdK<0b0011,
871                 (outs),
872                 (ins GPR8:$rd, i8imm$k),
873                 "cpi\t$rd, $k",
874                 [(AVRcmp GPR8:$rd, imm:$k), (implicit SREG)]>;
```

Listing 7: *The fixed definition of `cpi` in `AVRInstrInfo.td`.*

Exercise 5

Make the changes required to fix the selection of the `cpi` instruction and add the test for this problem in the AVR code generation test folder with filename `cpi.ll`. 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 and check if the selection of the `cpi` instruction now works as expected and that there are no other test-cases that are now suddenly broken.
Did everything work for you? Probably not, there is a typo in the above pattern which you will need to fix before everything works. Fix the problem using the other operation definitions as reference.

[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 6

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 wich 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 7

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 are two untracked files now (your test cases for the extra `cp` operations and the one for the `cpi` instruction).
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 the `cpi` instruction selection.

```
$ git add lib/Target/AVR/AVRInstrInfo.td
$ git add test/CodeGen/AVR/cpi.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 Oncourse website.

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

OK, that finishes our side-quest. You've fixed the compiler and submitted your first patch for review. Back to our main exercise for today.

2.5 Detecting instruction patterns

We can also 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 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 1900 and in Listing 8. 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.

```
1900 def : Pat<(add IWREGS:$src1, imm0_63_neg:$src2),
1901          (SBIWRdK IWREGS:$src1, (imm0_63_neg:$src2))>;
```

Listing 8: *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[9], the `imm0_63_neg` in the second part applies the transformation of $(-k)$ into k using the `imm16_neg_XFORM` defined on line 78 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 8

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 9.

[9] It is defined on line 83 in `AVRInstrInfo.td`.

```
1965 // Fix arithmetic operations followed by compare 0 to skip the compare and use
1966 // the SREG values already present
1967 def : Pat<(AVRcmp (sub GPR8:$src1, GPR8:$src2), (i8 0)),
1968          (SUBRdRr GPR8:$src1, GPR8:$src2)>;
```

Listing 9: *The added pattern for avoiding comparissons in the 8-bit case.*

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 390.

Exercise 9

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`, test your new patterns, and prepare and submit a patch[a] like you did in exercise 7.

[a] Remember, a good patch should contain the changes needed for a new feature (or fix) and test(s) that prove it works (and keeps working). Furthermore, unrelated changes should go into separate patches.

Ok, that's it for this part of the exercise!

The assembly code for the first loop of the target application should now look like this:

```
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     1: sbiw R26,1
25     brne 1b
26     ;NO_APP
27     sbiw   r30, 1
28     brne   LBB0_2
29 ; BB#3:                                ; %_delay_ms.exit
```

Listing 10: *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.