Assignment 1c: Compiler organization and backend programming

Roel Jordans

2016

Organization

Welcome to the third and final part of assignment 1. This time 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.

3 Introducing a new intrinsic

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 usign 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.

3.1 Declaring the intrinsic to LLVM

So, how to get started? The LLVM documentation contains a nice page[1] 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.

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

Exercise 1

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 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.

3.2 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 Oncourse website together with this description.

Exercise 2

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 file containing the patch `frontend-0001-delay_cycles_support.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 ~/exercise2.patch
$ git apply --check ~/exercise2.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 ~/exercise2.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[2] 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.

3.3 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 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.

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

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 3

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 crashsite isn't really nice so we'll start by creating a new test for it.
This test should contain the following:

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

define void @test() {
  tail call void @llvm.avr.delay.cycles(i32 16000000)
  ret void
}

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.

3.4 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 4

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.

```
  // Allow i32 constants as argument of intrinsics
  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 629 of the same file. In this method you will see the following lines:

```
default:
  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 1, 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.

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
```

3.5 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

```
case ISD::Constant: {
  assert(N->getValueType(0) == MVT::i32 && "Expected Constant<i32>");
  // Lower Constant<i32> into a TargetConstant iff its use is an intrinsic
  //  otherwise split into two half-parts to allow further lowering
  bool split = true;
  for (const auto use : N->uses()) {
    if (use->getOpcode() == ISD::INTRINSIC_VOID) {
      split = false;
    }
  }

  const uint32_t C = cast<ConstantSDNode>(N)->getZExtValue();
  if (split) {
    // Split into Lo16, Hi16, low part first
    // FIXME
  } else {
    // Lower into a TargetConstant for __delay_builtin_ms
    // FIXME
  }
  break;
}
```

Listing 1: *The custom lowering snippet for the Constant node.*

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 1843, the pattern that is added for selecting the LSL operation with a variable amount of shift into a new pseudo operation `Lsl8`. 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 5

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)]>;
```

Which 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 6

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 ~/exercise6.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