Tutorial: Building a backend in 24 hours

Anton Korobeynikov
anton@korobeynikov.info

Outline

1. Codegen phases and parts
2. The Target
3. First steps
4. Custom lowering
5. Next steps

Codegen Phases

- Preparation Phases
- Selection DAG Phases
- Late Optimizations
- Register Allocation
- Post-RA Phases
- Code Emission (Assembler Printing and/or Binary Code Emission)

SelectionDAG Phases

- Lower
- Combine
- Legalize
- Combine
- Instruction Selection
- Schedule

Check excellent Dan’s talk at 2008 Dev. Meeting!

Register Allocator

Here magic starts - virtual registers are turned into physical ones ...

See Lang's talk today for more information

Post-RA Phases

- Prologue / Epilogue Insertion & Abstract Frame Indexes Elimination
- Post-RA Scheduler
- Branch Folding
- Target-specific passes (e.g. IT block formation on Thumb2, delay slot filler on Sparc)

Backend

- Standalone library
- Mixed C++ / TableGen
- TableGen is a special DSL used to describe register sets, calling conventions, instruction patterns, etc.
- Inheritance and overloading is used to augment necessary target bits into target-independent codegen classes

Backend

- Target & Subtarget: X86Subtarget.cpp, X86Target.cpp
- Lowering: X86ISelLowering.cpp
- Register Set: X86RegisterInfo.td
- Register Information: X86RegisterInfo.cpp
- Instructions: X86InstrInfo.cpp, X86InstrInfo.td & around
- Instruction selection: X86ISelDAGToDAG.cpp
- Calling Convention: X86CallingConv.td

+ asm printer, JIT hooks, etc.

MSP430 MCU

- 16-bit RISC-based MCU
- 1/8/16-bit Data &16-bit Pointers
- Powerful addressing modes
- Simple instruction set:
  27 instructions in 3 groups
- Most instructions are available in 8-bit and 16-bit variants

First Steps

Consider the following code:

```
define void @foo() {
entry:
    ret void
}
```

What should we implement to let this code compile?

First Steps

Consider the following code:

```
define void @foo() {
entry:
    ret void
}
```

What should we implement to let this code compile?

Unfortunately, a lot ...

Skeleton Backend

- Implement blank backend classes
- Provide data layout (aka ‘TargetData’)
- Describe register set (types, allocation order, ...) for native types (8 and 16 bits)
- Hook everything into build system & target registration facilities

Calling Convention

1. Create MSP430CallingConv.td:
   - Describe argument & return value passing rules

Calling Convention

1. Create MSP430CallingConv.td:
   - Describe argument & return value passing rules

```
def CC_MSP430 : CallingConv<[
  // Promote i8 arguments to i16.
  CCIfType<[i8], CCPromoteToType<i16>>,

  // The first 4 integer arguments of non-varargs functions are
  // passed in integer registers.
  CCIfNotVarArg<CCIfType<[i16],
                CCAssignToReg<[R15W, R14W, R13W, R12W]>>>,

  // Integer values get stored in stack slots that are 2 bytes in
  // size and 2-byte aligned.
  CCIfType<[i16], CCAssignToStack<2, 2>>
]>;
```

Calling Convention

1. Create MSP430CallingConv.td:
   - Describe argument & return value passing rules
2. MSP430TargetLowering.cpp:
   - Implement LowerFormalArguments() method
   - Implement LowerReturn() method
   - Add target node for return instruction
3. MSP430InstrInfo.td:
   - Add isel pattern for return instruction

LowerFormalArguments()

1. Assign locations to all incoming arguments (depending on their type)
2. Copy arguments passed in registers
3. Create frame index objects for arguments passed on stack
4. Create SelectionDAG nodes for loading of stack arguments

Same for LowerReturn(), but in “reverse” direction

Refinements: hooks

1. MSP430RegisterInfo.cpp:
   - getReservedRegisters()
   - hasFP()
   - getCalleeSavedRegs() / getCalleeSavedRegClasses()
   - emitPrologue() / emitEpilogue()
2. MSP430AsmPrinter.cpp:
   - Add instruction printing skeleton

Result

```
$ llc -march=msp430 00-RetVoid.ll -o -

	ret
```

But we actually implemented more!

Result

```
$ llc -march=msp430 00-RetVoid.ll -o -

       ret
```

But we actually implemented more!

What's about this?

```
define i16 @foo() {
entry:
    ret i16 0
}
```

or

```
define i16 @bar(i16 %a) {
entry:
    ret i16 %a
}
```

Moves

- Register-immediate:

```
let isReMaterializable = 1, isAsCheapAsAMove = 1 in {
def MOV16ri : Pseudo<(outs GR16:$dst), (ins i16imm:$src),
                     "mov.w\t{$src, $dst}",
                     [(set GR16:$dst, imm:$src)]>;
}
```

Moves

- Register-immediate:

```
let isReMaterializable = 1, isAsCheapAsAMove = 1 in {
def MOV16ri : Pseudo<(outs GR16:$dst), (ins i16imm:$src),
                     "mov.w\t{$src, $dst}",
                     [(set GR16:$dst, imm:$src)]>;
}
```

```
define i16 @foo() {
entry:
    ret i16 0
}
```

→

```
        .text
foo:
        mov.w   #0, r15
        ret
```

Moves

- Register-register:
  1. MSP430InstrInfo.cpp:
     implement copyRegToReg() & isMoveInstr()
  2. MSP430InstrInfo.td: provide instruction

```
let neverHasSideEffects = 1 in
def MOV16rr : Pseudo<(outs GR16:$dst), (ins GR16:$src),
                     "mov.w\t{$src, $dst}",
                     []>;
```

Moves

- Register-register:
  1. MSP430InstrInfo.cpp:
     implement copyRegToReg() & isMoveInstr()
  2. MSP430InstrInfo.td: provide instruction

```
let neverHasSideEffects = 1 in
def MOV16rr : Pseudo<(outs GR16:$dst), (ins GR16:$src),
                     "mov.w\t{$src, $dst}",
                     []>;
```

```
define i16 @bar(i16 %a, i16 %b) {
entry:
    ret i16 %b
}
```

→

```
        .text
bar:
        mov.w   r14, r15
        ret
```

2+2=4

We have everything ready for reg-reg & reg-imm arithmetics.

Just add bunch instruction patterns:

```
let Defs = [SR], isTwoAddress = 1, isCommutable = 1 in {
def ADD16rr : Pseudo<(outs GR16:$dst), (ins GR16:$src1, GR16:$src2),
                     "add.w\t{$src2, $dst}",
                     [(set GR16:$dst, (add GR16:$src1, GR16:$src2)),
                      (implicit SR)]>;
}
```

Same for addc, sub, and, or, xor, subc, ...

2+2=4

We have everything ready for reg-reg & reg-imm arithmetics.

```
define i16 @foo(i16 %a, i16 %b) {
entry:
      %c = add i16 %a, %b
      ret i16 %c
}
```

↓

```
        .text
foo:
        add.w   r14, r15
        ret
```

Too few instructions...

Now we have a problem:

```
define i16 @foo(i16 %a) {
entry:
      %c = ashr i16 %a, 2
      ret i16 %c
}
```

Too few instructions...

Now we have a problem:

```
define i16 @foo(i16 %a) {
entry:
        %c = ashr i16 %a, 2
        ret i16 %c
}
```

MSP430 (not MSP430X) has only single bit shift...

Too few instructions...

Now we have a problem:

```
define i16 @foo(i16 %a) {
entry:
      %c = ashr i16 %a, 2
      ret i16 %c
}
```

MSP430 (not MSP430X) has only single bit shift...

We need to custom lower the shift to target-specific single bit one

Custom Lowering

- Mark ISD::SRA operation as ‘custom lowered’
- Add new target node for single-bit shift
- Expand multi-bit shift to series of single-bit ones
- Declare new target node in MSP430InstrInfo.td & add instruction pattern:

```
let isTwoAddress = 1 in
def SAR16r1 : Pseudo<(outs GR16:$dst), (ins GR16:$src),
                     "rra.w\t$dst",
                     [(set GR16:$dst, (MSP430rra GR16:$src)),
                      (implicit SR)]>;
```

Some optimizations can be inserted as well, e.g.

```
foo >> (8 + N) => sxt(swpb(foo)) >> N
```

Custom Lowering

```
define i16 @foo(i16 %a) {
entry:
      %c = ashr i16 %a, 2
      ret i16 %c
}
```

$$\downarrow$$

```
        .text
foo:
        rra.w   r15
        rra.w   r15
        ret
```

What’s next?

- Memory operands - target-specific DAG matching
- Prologue / Epilogue emission & stack frame handling
- Handling of callee-saved registers
- Comparisons, jumps, calls, globals, alloca’s, jump tables
- Other arithmetics (muls, divs, rems)
- Hook into clang

+ Many interesting optimizations

Conclusion

Backend building is not that hard!

(Given that your target CPU is sane enough ...)

Conclusion

Backend building is not that hard!

(Given that your target CPU is sane enough ...)

- Look into MSP430 & SystemZ backends (all step-by-step history was preserved)
- Some ‘feature’ tests can be found in test/CodeGen/Generic & test/CodeGen/SystemZ
- For more complex cases look into other backends (X86, ARM, etc.)
- Code from this talk will be available soon

Q & A