Extending the internal assembler
How to add a new CPU feature

Kai Nacke
1 February 2015
LLVM devroom @ FOSDEM’15
Agenda

• Motivation

• Adding a CPU feature

• Adding an instruction

• Instruction selection
MIPS based embedded devices

A router
• Cavium Octeon CPU
• > 20 new instructions

A Rasberry Pi type device
• Ingenic JZ4780 CPU
• > 60 new instructions (MXU extension)
Motivation

• New instructions are quite common

• You can use inline assembly if the assembler knows the instructions

• Even better if the instruction is selected by the code generation
Target definition files

- TableGen language is used to define target specific information
- Declarative language
- Record based
  - `class` defines a record
  - `def` instantiates a record
Target definition files (2)

• Certain classes are predefined
  – E.g. Instruction for instruction encodings
  – See files
    include/llvm/Target/Target*.td

• llvm-tablegen generates C++ source (*.inc)
  – Generic tool, also used for Clang options

• Generated files are included at various places
  – #define‘s are used to alter behaviour
Add a subtarget feature

• Define the feature in Mips.td

```python
def FeatureCnMips:
    SubtargetFeature<$cnmips$,  // Commandline
                   "HasCnMips",  // Property name for feature
                   "true",      // Value for property if
                   // feature is selected
                   "Octeon cnMIPS Support",  // Help text
                   [FeatureMips64r2]>;  // Implied features
```

• Add property to class MipsSubtarget:

```c
// CPU supports cnMIPS (Cavium Networks Octeon CPU).
bool HasCnMips;

bool hasCnMips() const { return HasCnMips; }
```

• Initialize the property in the constructor!
Select feature by CPU

• Define the CPU in Mips.td

```python
def : Processor<"octeon", // Name
    MipsGenericItineraries, // Itineraries
    [FeatureMips64r2, FeatureN64, FeatureCnMips]>;
        // Implied features
```

• With scheduling model: ProcessorModel

• Run make!

• Check llc –march=mips64 –mcpu=help

• Code generated by llvm-tablegen is in file MipsGenSubtargetInfo.inc
Plan the first instruction

• Requires a predicate to toggle selection
  ```python
def HasCnMips : Predicate<"Subtarget->hasCnMips()">,
  AssemblerPredicate<"FeatureCnMips">;
  ```

• For the assembler
  – You need to know the encoding

• For code selection
  – You need to know the operation
Define Instruction Format Class

- Needed for each instruction
- Here: R-form

class POP_FM<bits<6> funct> : StdArch {
    bits<5> rd;
    bits<5> rs;
    bits<32> Inst;
    let Inst{31-26} = 0x1c;
    let Inst{25-21} = rs;
    let Inst{20-16} = 0;
    let Inst{15-11} = rd;
    let Inst{10-6} = 0;
    let Inst{5-0} = funct;
}

Format: DPOP rd, rs
Define Instruction Class

• Defines input and output parameters
• Defines the DAG pattern

```python
class Count1s<string opstr, RegisterOperand RO>:
    InstSE<(outs RO:$rd),                          // Output
        (ins RO:$rs),                           // Input
        !strconcat(opstr, "$rd, $rs"),        // Assembler
        [(set RO:$rd, (ctpop RO:$rs))],         // Pattern
        II_POP,                                 // Itinary
        FrmR,                                   // Encoding
        opstr> {                                // Opcode
            let TwoOperandAliasConstraint = "$rd = $rs";
        }
```
Define Instruction

• Uses the previous defined classes
• Uses predicate for conditionally selection

let EncodingPredicates = [],
    AdditionalPredicates = [HasCnMips] in {  
  // Count Ones in a Word/Doubleword
  def POP : Count1s"pop", GPR32Opnd>, POP_FM<0x2c>;
  def DPOP : Count1s"dpop", GPR64Opnd>, POP_FM<0x2d>;
}

• Run make!
• Check that the assembler now accepts dpop (test case!)
Instruction Selection

- Instruction selection is pattern based
- No selection if pattern is empty
- More complex selections and intrinsics can be coded in MipsISelLowering.cpp
- Required here: CTPOP is legal for the new feature

```cpp
if (Subtarget.hasCnMips()) {
    setOperationAction(ISD::CTPOP, MVT::i32, Legal);
    setOperationAction(ISD::CTPOP, MVT::i64, Legal);
} else {
    setOperationAction(ISD::CTPOP, MVT::i32, Expand);
    setOperationAction(ISD::CTPOP, MVT::i64, Expand);
}
```
Patterns

- Patterns are s-expressions like in Scheme
- Complex conditions are possible
- Can use predicates coded in C++
- Used in instruction definitions and aliases
Complex Pattern Example

- Performs \( rd = (rs + rt) \& 0xFF \)

```plaintext
// Unsigned Byte Add
let Pattern = [(set GPR64Opnd:$rd,
    (and (add GPR64Opnd:$rs,
        GPR64Opnd:$rt), 255))]

def BADDu : ArithLogicR"baddu", GPR64Opnd, 1,
    II_BADDU>, ADD_FM<0x1c, 0x28>;
```
Resources

• All shown code is in LLVM 3.5
• Creating an LLVM backend for the Cpu0 Architecture
• Building an LLVM Backend
• A deeper look into the LLVM code generator, Part 1
Backup
Adding assembler checks

- Most assembler parsers have a method called `processInstruction`.
- Called for every instruction.
- Typically checks operands, relocations, etc.