The implementation of AArch64 Neon™ in LLVM

Jiangning Liu
Principal Software Engineer
ARM

Agenda

- AArch64 Overview
- AArch64 Neon™ implementation
- Clang/LLVM vs. GCC
- Acknowledgments

AArch64 Overview

- AArch64 is orthogonal with ARM v8.
- 32-bit instruction width.
- Conditional or predicated execution is unavailable.
- VFP and Neon™ share the same register file.

AArch64 Neon™

- 32 128-bit registers.
  - 64-bit or even smaller registers are 1:1 map with 128-bit.
- Has cmp&select but no branch instruction
- SIMD and SISD support
- Support FMA following IEEE754
- Some misc but powerful instructions
  - Saturating calculation
  - Permutation/extract/table lookup

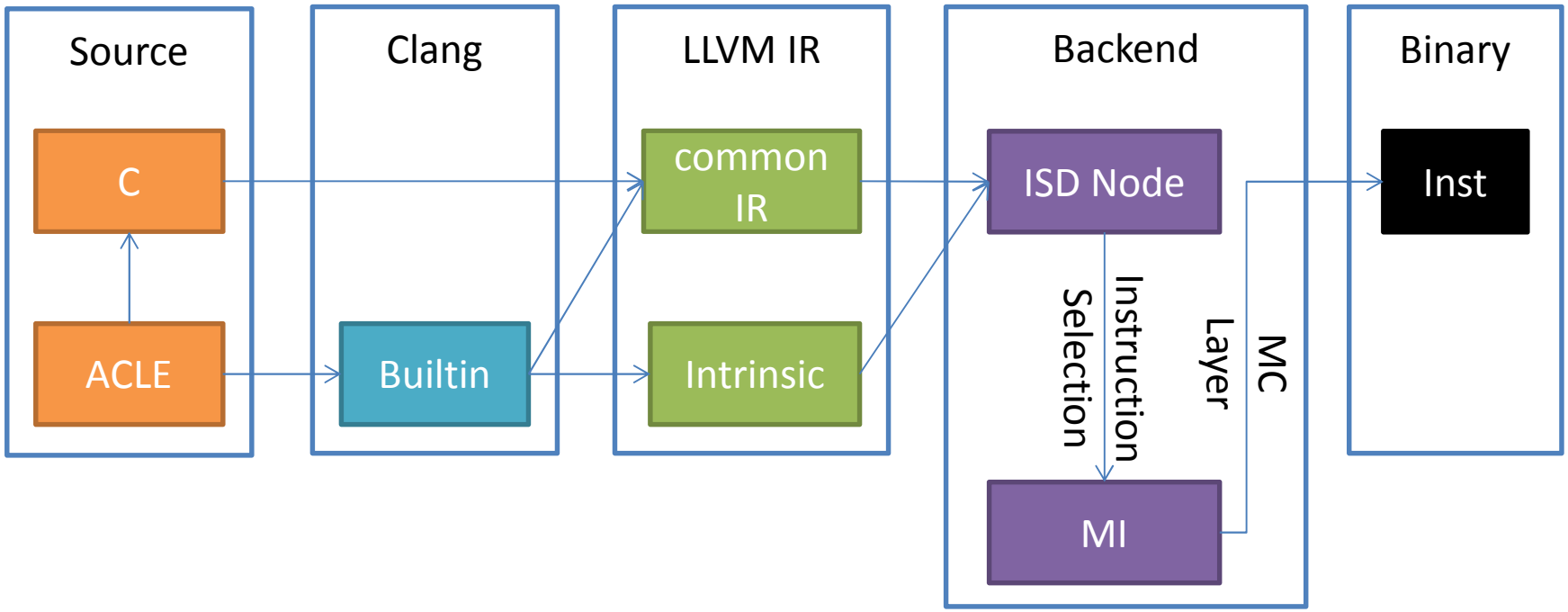
Implementation Goals

- Support all instructions in AArch64 NEON™
- Support all ACLE intrinsics
- Provide ARM 32-bit backward compatibility
- Integrated assembler on by default
  - MC Layer completeness: instruction printing, asm parsing, assembling, disassembling.

AArch64 Neon™ Implementation Status

Instruction Class	100% complete	Complete except community code review
AdvSIMD (imm)	14	
AdvSIMD (3 same)	80	
AdvSISD (3 same)	32	
AdvSIMD (lselem)	13	1
AdvSIMD (lsone)		36
AdvSIMD (3 diff)	26	
AdvSIMD (misc)	62	
AdvSIMD (across)	23	
AdvSIMD (insdup)	24	2
AdvSIMD (by element)	18	
AdvSIMD (shift)	28	
AdvSIMD (table)	8	
AdvSIMD (perm)	6	
AdvSIMD (extract)	2	
AdvSISD (3 diff)	3	
AdvSISD (misc)	36	
AdvSISD (pairwise)	6	
AdvSISD (copy)	3	1
AdvSISD (by element)	10	
AdvSISD (shift)	24	
AdvSIMD (lselem-post)	14	
AdvSIMD (lsone-post)		36
AdvSIMD Crypto (aes)	4	
AdvSIMD Crypto (3 sha)	7	
AdvSIMD Crypto (sha)	3	
	446	76
	85.44%	100.00%

Implement AArch64 Neon™



Implementation Details

- Minimize LLVM IR intrinsics
  - Reusing ARM definitions when possible
- Shared arm_neon.h between ARM and AArch64
- MI–based scheduler turned on by default
  - Overloaded memory cost model in Selection DAG-based scheduler causes issues
- Using RegisterOperand instead of Register class when defining Neon™ registers.
  - Avoid redundant register transformation between GPR and VPR.

Design Level Challenges

- Determine where to do lowering
  - front-end
  - back-end
- Determine how to do instruction selection
  - Write good and clean tablegen patterns
  - Write CPP code in IselDagToDag
- Determine scalar data type representation
  - One element vector
  - Pure scalar

Effective Table Gen

```
class NeonI_2VElem<bit q, bit u, bits<2> size, bits<4> opcode,
                   dag outs, dag ins, string asmstr,
                   list<dag> patterns, InstrItinClass itin>
  : A64InstRdnm<outs, ins, asmstr, patterns, itin> {
  let Inst{31} = 0b0;
  let Inst{30} = q;
  let Inst{29} = u;
  let Inst{28-24} = 0b01111;
  let Inst{23-22} = size;
  // l in Inst{21}
  // m in Inst{20}
  // Inherit Rm in 19-16
  let Inst{15-12} = opcode;
  // h in Inst{11}
  let Inst{10} = 0b0;
  // Inherit Rn in 9-5
  // Inherit Rd in 4-0
}
class NI_2VE<bit q, bit u, bits<2> size, bits<4> opcode,
             string asmop, string ResS, string OpS, string EleOpS,
             Operand OpImm, RegisterOperand ResVPR,
             RegisterOperand OpVPR, RegisterOperand EleOpVPR>
  : NeonI_2VElem<q, u, size, opcode,
    (outs ResVPR:$Rd),
    (ins ResVPR:$src, OpVPR:$Rn, EleOpVPR:$Re, OpImm:$Index),
     asmop # "\t$Rd." # ResS # ", $Rn." # OpS #  ", $Re." # EleOpS
# "[$Index]",
     [],
     NoItinerary> {
  bits<3> Index;
  bits<5> Re;
  let Constraints = "$src = $Rd";
}
```

```
multiclass NI_2VE_v2<bit u, bits<4> opcode, string asmop> {
  // vector register class for element is
  // always 128-bit to cover the max index
  def _4s4s : NI_2VE<0b1, u, 0b10, opcode, asmop, "4s", "4s", "s",
                     neon_uimm2_bare, VPR128, VPR128, VPR128> {
    let Inst{11} = {Index{1}};
    let Inst{21} = {Index{0}};
    let Inst{20-16} = Re;
  }
  ...
}
defm FMLAvve : NI_2VE_v2<0b0, 0b0001, "fmla">;
// Pattern for lane in 64-bit vector
class NI_2VEswap_lane<Instruction INST,
                      Operand OpImm, SDPatternOperator op,
                      RegisterOperand ResVPR, RegisterOperand OpVPR,
                      ValueType ResTy, ValueType OpTy,
                      SDPatternOperator coreop>
  : Pat<(ResTy (op (ResTy (coreop (OpTy OpVPR:$Re), (i64
OpImm:$Index))),  (ResTy ResVPR:$Rn), (ResTy ResVPR:$src))),
    (INST ResVPR:$src, ResVPR:$Rn,
      (SUBREG_TO_REG (i64 0), OpVPR:$Re, sub_64),
      OpImm:$Index)>;
multiclass NI_2VE_fma_v2_pat<
                    string subop, SDPatternOperator op> {
  def : NI_2VEswap_lane<
        !cast<Instruction>(subop # "_4s4s"),
        neon_uimm1_bare, op, VPR128, VPR64, v4f32, v2f32,
        BinOpFrag<(Neon_vduplane
                    (Neon_combine_4f node:$LHS, undef),
                    node:$RHS)>>;   ...
}
defm FMLA_lane_v2_s : NI_2VE_fma_v2_pat<"FMLAvve", fma>;
```

Testing

- LLVM Regression tests
  - Assembling, disassembling, codegen (.ll, .c, .txt)
- LLVM auto-generated ACLE tests
  - arm_neon_test.h, arm_neon_sema.h
- ARM's MC Hammer test
  - MC layer test against golden implementation
- ARM's Emperor tests
  - Randomly generated ACLE composition test
- Other vector workloads to test pattern matching

Future Work

- Complete NEON™ implementation by end of Nov. 2013
- Performance tuning
  - Investigate regressions compared to the most recent GCC
    - Use ARM's foundation model available at http://www.linaro.org/engineering/engineering-projects/armv8
  - Add micro-architecture description to MI-based scheduler
    - Based on ARM's code generation guidelines
  - Explore sub-word level parallelism (SLP)
    - Improve pattern matching quality for vectorizers.

Clang/LLVM vs. GCC

	GCC	Clang/LLVM
1	Limited C++ code	C++ code base, more modulized
2	GPL License	UIUC License (FreeBSD Style)
3	A compiler tool chain only	A building block for compiler technology
4	Can only support one ISA for a single compiler build	Can support multiple different ISAs simultaneously with a single build
5	IR can be dumped for reading purpose only	IR itself is a language, and can feed middle-end passes directly
6	LISP-like .md description for back-end code generation	tablegen domain specific language for retargetable code description
7	Use Tcl syntax in deja GNU framework	Use lit and FileCheck tools only
8	Must generate assembly code first	Can generate binary through MC layer directly
9	front-end and back-end are tightly combined	"LLVM" can be built independently without clang
10	No JIT support	Naturally support JIT/MCJIT building block

Acknowledgments

- Ana Pazos (QuIC Inc.)
- Hao Liu (ARM Ltd. Shanghai)
- Kevin Qin (ARM Ltd. Shanghai)
- Chad Rosier (QuIC Inc.)
- Tim Northover (Apple)
- Clang and LLVM community reviewers