AArch64: ARM’s 64-bit architecture

Tim Northover

November 8, 2012
Outline

AArch64 Architecture
AArch64 Backend
Testing the Backend
Interesting Curiosities
  Load-store Patterns
  Templated Operands
  Conditional Compare
Creating the Backend
Future Ideas
So what is AArch64 then?

ARM’s new 64-bit architecture.

RISC-like; fixed 32-bit instruction width.

31 general purpose registers, x0-x30 with 32-bit subregisters w0-w30 (+PC, +SP, +ZR)

Always an FPU; 32 registers, each 128-bits wide.

About as nice as a compiler could hope for.
AArch64 Architecture

So what is AArch64 then?

- ARM’s new 64-bit architecture.
AArch64 Architecture

So what is AArch64 then?

- ARM’s new 64-bit architecture.
- RISC-like; fixed 32-bit instruction width.
AArch64 Architecture

So what is AArch64 then?

- ARM’s new 64-bit architecture.
- RISC-like; fixed 32-bit instruction width.
- 31 general purpose registers, x0-x30 with 32-bit subregisters w0-w30 (+PC, +SP, +ZR)
So what is AArch64 then?

- ARM’s new 64-bit architecture.
- RISC-like; fixed 32-bit instruction width.
- 31 general purpose registers, x0-x30 with 32-bit subregisters w0-w30 (+PC, +SP, +ZR)
- Always an FPU; 32 registers, each 128-bits wide.
AArch64 Architecture

So what is AArch64 then?

- ARM’s new 64-bit architecture.
- RISC-like; fixed 32-bit instruction width.
- 31 general purpose registers, x0-x30 with 32-bit subregisters w0-w30 (+PC, +SP, +ZR)
- Always an FPU; 32 registers, each 128-bits wide.
- About as nice as a compiler could hope for.
Tiny Example

```c
int foo(int val) {
    int newval = bar(val);
    return val + newval;
}
```
Tiny Example

```c
int foo(int val) {
    int newval = bar(val);
    return val + newval;
}
```

Could compile to

```assembly
foo:
    sub    sp, sp, #16
    stp    x19, x30, [sp]
    mov    w19, w0
    bl     bar
    add    w0, w0, w19
    ldp    x19, x30, [sp]
    add    sp, sp, #16
    ret
```
Tiny Example

```c
int foo(int val) {
    int newval = bar(val);
    return val + newval;
}
```

Could compile to

```
foo:
    sub    sp, sp, #16
    stp    x19, x30, [sp]
    mov    w19, w0
    bl     bar
    add    w0, w0, w19
    ldp    x19, x30, [sp]
    add    sp, sp, #16
    ret

foo:
    sub    sp, sp, #8
    strd   r4, r14, [sp]
    mov    r4, r0
    bl     bar
    add    r0, r0, r4
    ldrd   r4, r14, [sp]
    add    sp, sp, #8
    bx     lr
```
Outline

AArch64 Architecture

AArch64 Backend

Testing the Backend

Interesting Curiosities
  Load-store Patterns
  Templated Operands
  Conditional Compare

Creating the Backend

Future Ideas
AArch64 Backend: Goals

What we wanted:

- LLVM backend targeting ELF output on Linux.
- Integrated assembler on by default.
- Using up to date LLVM APIs and style.
- Strong testing.
- Compiling standard-compliant C and C++.

What we didn't want:

- Optimisation less important (for now!).
- Features unused by C and C++ were lower priority.
AArch64 Backend: Goals

What we wanted:
- LLVM backend targeting ELF output on Linux.
- Integrated assembler on by default.
What we wanted:

- LLVM backend targeting ELF output on Linux.
- Integrated assembler on by default.
- Using up to date LLVM APIs and style.
AArch64 Backend: Goals

What we wanted:

- LLVM backend targeting ELF output on Linux.
- Integrated assembler on by default.
- Using up to date LLVM APIs and style.
- Strong testing.
- Compiling standard-compliant C and C++.
AArch64 Backend: Goals

What we wanted:
- LLVM backend targeting ELF output on Linux.
- Integrated assembler on by default.
- Using up to date LLVM APIs and style.
- Strong testing.
- Compiling standard-compliant C and C++.

What we didn’t want:
- Optimisation less important (for now!).
- Features unused by C and C++ were lower priority.
AArch64 Backend: Tests Passed

- C++98 and C99 well supported.
- SPEC2000 and SPEC2006 run successfully (e.g. gcc, perl).
- Self-built clang and LLVM pass the regression testsuite, both as shared libraries and static (takes 12 hours to run on a model).
- NEON work ongoing, but not ready for use.
- LLVM testsuite has about 10 failures.
- MC Hammer passes on scalar instructions (see later).
Getting Started

There’s a model and basic Linux filesystem available at http://www.linaro.org/engineering/armv8/

- Model of a fixed, reasonably complete system.
- Linux filesystem (OpenEmbedded) to boot it.
- Toolchain for headers, linkers, ...

Used for our internal tests currently.

Try to compile your favourite program! See what breaks it!
Outline

AArch64 Architecture
AArch64 Backend

Testing the Backend
Interesting Curiosities
Load-store Patterns
Templated Operands
Conditional Compare

Creating the Backend
Future Ideas
Lower Level Testing: MC Hammer

- Implemented by Richard Barton and presented at Euro-LLVM.
- Idea: automatically test all 32-bit bitpatterns against another (independent) implementation.
- Ensures InstPrinter, AsmParser, Disassembler and MCCodeEmitter are consistent and correct.
- Covers all bitpatterns, but only checks valid assembly.
MC Hammer on AArch64

How did it help us?

- Executed on all builds for all scalar instructions.
- Directed us towards the useful regression tests.
- Still need good regression tests to save time and (hopefully) prevent any bad commit.
Testing the Hard Parts: Relocations

- Do the numbers match? Are they filtered through the umpteen layers of indirection properly? E.g.
  - MO_LO12 → VK_AARCH64_LO12
  - fixup_a64_add_lo12
  - R_AARCH64_ADD_ABS_LO12_NC
  - 0x115

- I think so, but...

- Have to run both llvm-objdump (check names) and elf-dump (check numerics) to test everything.
Testing the Hard Parts: CodeGen

- Can never be quite sure about all the edge cases.
- Regression tests for each pattern, of course.
- No revolutionary new solution here.
- Ultimately, running real code is the only way.
Testing the Hard Parts: Misc

1 Exceptions

- In principle, straightforward DWARF style on AArch64.
- But, small model: code and data in single 4GB space.
- Implies relocations need 64-bit (even PC-relative need +4GB and -4GB).
- Took a couple of tries, mixed in with link-time failures.
Testing the Hard Parts: Misc

1 Exceptions

- In principle, straightforward DWARF style on AArch64.
- But, small model: code and data in single 4GB space.
- Implies relocations need 64-bit (even PC-relative need +4GB and -4GB).
- Took a couple of tries, mixed in with link-time failures.

2 Debugging information

- Another one that can look OK but be wrong.
- Even harder to test beyond “Looks ok to me. Maybe.”
Outline

AArch64 Architecture
AArch64 Backend
Testing the Backend
Interesting Curiosities
  Load-store Patterns
  Templated Operands
  Conditional Compare
Creating the Backend
Future Ideas
Load-store Patterns: the Problem

```python
def addr_op : Operand<i64>,
    ComplexPattern<i64, 2, "SelectAddress"> {
        let MIOperandInfo = (ops GPR64:$base, imm:$offset);
        ...
    }

    // ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd), (ins addr_op:$addr),
        "ldr $Rd, $addr",
        [(set GPR64:$Rd, (load addr_op:$addr))]>;
```

- Needs custom AsmParser
Load-store Patterns: the Problem

\[
\text{def addr_op : Operand}\langle\text{i64}\rangle,
\quad \text{ComplexPattern}\langle\text{i64}, 2, \text{"SelectAddress"}\rangle \{ \\
\quad \text{let MIOperandInfo = (ops GPR64:$base, imm:$offset)}; \\
\quad \ldots \\
\}\]

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd), (ins addr_op:$addr), \\
\quad \text{\"ldr \$Rd, \$addr\"}, \\
\quad [(set GPR64:$Rd, (load addr_op:$addr)])]>;

- Needs custom AsmParser, InstPrinter
Load-store Patterns: the Problem

```python
def addr_op : Operand<i64>,
    ComplexPattern<i64, 2, "SelectAddress"> {
    let MIOperandInfo = (ops GPR64:$base, imm:$offset);
    ...}

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd), (ins addr_op:$addr),
    "ldr $Rd, $addr",
    [(set GPR64:$Rd, (load addr_op:$addr))]>;
```

- Needs custom AsmParser, InstPrinter, Disassembler
Load-store Patterns: the Problem

```python
def addr_op : Operand<i64>,
    ComplexPattern<i64, 2, "SelectAddress"> {
        let MIOperandInfo = (ops GPR64:$base, imm:$offset);
        ...
}

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd), (ins addr_op:$addr),
    "ldr $Rd, $addr",
    [(set GPR64:$Rd, (load addr_op:$addr))]>;
```

- Needs custom AsmParser, InstPrinter, Disassembler and Encoder.
Load-store Patterns: the Problem

```python
def addr_op : Operand<i64>,
    ComplexPattern<i64, 2, "SelectAddress"> {
        let MIOperandInfo = (ops GPR64:$base, imm:$offset);
    ...
}

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd), (ins addr_op:$addr),
    "ldr $Rd, $addr",
    [(set GPR64:$Rd, (load addr_op:$addr))]>;
```

- Needs custom AsmParser, InstPrinter, Disassembler and Encoder.
- Complex, duplicated C++ selection code (ldr x0, [x3, w5, sxtw #3]).
Load-store Patterns: the Solution

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd),
    (ins GPR64:$Rn, uimm12:$offset),
    "ldr $Rd, [$Rn, $offset]", [???]>

- All the MC components become much simpler: a normal instruction.
- Patterns not simpler.
Load-store Patterns: the Solution

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd),
    (ins GPR64:$Rn, uimm12:$offset),
    "ldr $Rd, [$Rn, $offset]", [???]>

- All the MC components become much simpler: a normal instruction.
- Patterns not simpler.
  - Need to construct patterns with varying shapes (e.g. shift/no shift). Aha! Inner multiclass should do this.
Load-store Patterns: the Solution

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd),
               (ins GPR64:$Rn, uimm12:$offset),
               "ldr $Rd, [$Rn, $offset]", [???]>;

- All the MC components become much simpler: a normal instruction.
- Patterns not simpler.
  1 Need to construct patterns with varying shapes (e.g. shift/no shift). Aha! Inner multiclass should do this.
  2 Need the contents of those DAGs to vary by instruction. Aha! Inner multiclass should do this
Load-store Patterns: the Solution

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd),
    (ins GPR64:$Rn, uimm12:$offset),
    "ldr $Rd, [$Rn, $offset]", [???]>

- All the MC components become much simpler: a normal instruction.
- Patterns not simpler.
  1. Need to construct patterns with varying shapes (e.g. shift/no shift). Aha! Inner multiclass should do this.
  2. Need the contents of those DAGs to vary by instruction. Aha! Inner multiclass should do this
Load-store Patterns: the Solution

// ldr x0, [sp, #16]
def LOAD : Inst<(outs GPR64:$Rd),
    (ins GPR64:$Rn, uimm12:$offset),
    "ldr $Rd, [$Rn, $offset]", [???]>

- All the MC components become much simpler: a normal instruction.
- Patterns not simpler.
  1. Need to construct patterns with varying shapes (e.g. shift/no shift). Aha! Inner multiclass should do this.
  2. Need the contents of those DAGs to vary by instruction. Aha! Inner multiclass should do this.
  3. Oh dear.
Load-store Patterns: Worthwhile?

The big question is, was it worth it?

- TableGen was horribly ugly: foreach, subst
- Could be improved hugely by improving TableGen.
- Reduces C++ complexity; increases TableGen complexity.
- Initial patch: +834 lines, -1288 lines.
The big question is, was it worth it?

- TableGen was horribly ugly: foreach, subst
- Could be improved hugely by improving TableGen.
- Reduces C++ complexity; increases TableGen complexity.
- Initial patch: +834 lines, -1288 lines.
- Undecided.
Problem: groups of similar operands. Mostly similar handling but details slightly different.

```cpp
defualt operand: AsmOperandClass {
  let PredicateMethod = "isUImm<6>";
  ...
}
```

Requires certain accommodation in what TableGen does with the strings.
Templating Operands: A Useful Trick

- Problem: groups of similar operands. Mostly similar handling but details slightly different.
- Solution: C++ templates.

```cpp
def uimm6_asmoperand : AsmOperandClass {
    let PredicateMethod = "isUImm<6>";
    ...
}
```

- Requires certain accommodation in what TableGen does with the strings.
Conditional Compare

ccmp x0, x1, #12, ge
Conditional Compare

ccmp x0, x1, #12, ge

Check NZCV flags for \( \geq \) (signed).
Conditional Compare

\texttt{ccmp x0, x1, \#12, ge}

- Check NZCV flags for $\geq$ (signed).
- If previous comparison passed, do this one and set NZCV.
Conditional Compare

```plaintext
ccmp x0, x1, #12, ge
```

- Check NZCV flags for $\geq$ (signed).
- If previous comparison passed, do this one and set NZCV.
- Otherwise, set NZCV to 12 (N=1, Z=1, C=0, V=0)
Before CCMP

\[ r_0 \geq r_1 \land r_2 \geq r_3 \]

Reasonably simple optimisation on ARM:

```
cmp r0, r1
cmp ge r2, r3
bge good
```

Generalisations:
- Any number of \( \geq \) comparisons.
- Or with \(<\) instead of And with \(\geq\).
- Certain compatible comparisons.

But there are limitations.
Before CCMP

\[ r_0 \geq r_1 \land r_2 \geq r_3 \]

Reasonably simple optimisation on ARM:

\[
\begin{align*}
\text{cmp} & \ r_0, \ r_1 \\
\text{cmpge} & \ r_2, \ r_3 \\
\text{bge} & \ \text{good}
\end{align*}
\]

Generalisations:

Any number of \( \geq \) comparisons.

Or with \(<\) instead of \(\land\) with \(\geq\).

Certain compatible comparisons.

But there are limitations.
Before CCMP

\[ r_0 \geq r_1 \land r_2 \geq r_3 \]

Reasonably simple optimisation on ARM:

\[
\text{cmp } r_0, r_1 \\
\text{cmpge } r_2, r_3 \\
\text{bge } \text{good}
\]

Generalisations:
- Any number of \( \geq \) comparisons.
Before CCMP

\[ r_0 \geq r_1 \land r_2 \geq r_3 \]

Reasonably simple optimisation on ARM:

\[
\begin{align*}
\text{cmp} & \ r_0, \ r_1 \\
\text{cmpge} & \ r_2, \ r_3 \\
\text{bge} & \ \text{good}
\end{align*}
\]

Generalisations:

- Any number of \( \geq \) comparisons.
- Or with \(<\) instead of \(\geq\).
Before CCMP

\[ r_0 \geq r_1 \quad \&\& \quad r_2 \geq r_3 \]

Reasonably simple optimisation on ARM:

\[
\begin{align*}
&\text{cmp } r_0, \text{ } r_1 \\
&\text{cmpge } r_2, \text{ } r_3 \\
&\text{bge good}
\end{align*}
\]

Generalisations:

- Any number of ≥ comparisons.
- Or with < instead of And with ≥.
- Certain compatible comparisons.
Before CCMP

\[ r_0 \geq r_1 \ \&\& \ r_2 \geq r_3 \]

Reasonably simple optimisation on ARM:

\[
\begin{align*}
\text{cmp} & \ r_0, \ r_1 \\
\text{cmpge} & \ r_2, \ r_3 \\
\text{bge} & \ \text{good}
\end{align*}
\]

Generalisations:

- Any number of \(\geq\) comparisons.
- Or with \(<\) instead of And with \(\geq\).
- Certain compatible comparisons.

But there are limitations.
With CCMP

\[ x_0 \geq x_1 \land x_2 = x_3 \]

First try:

\begin{align*}
\text{cmp} & \quad r_0, \ r_1 \\
\text{cmpge} & \quad r_2, \ r_3 \\
bXX & \quad \text{good}
\end{align*}
With CCMP

\[ x_0 \geq x_1 \land x_2 = x_3 \]

First try:

\[
\begin{align*}
\text{cmp} & \quad r0, \ r1 \\
\text{cmpge} & \quad r2, \ r3 \\
\text{bXX} & \quad \text{good}
\end{align*}
\]

But with CCMP:

\[
\begin{align*}
\text{cmp} & \quad x0, \ x1 \\
\text{ccmp} & \quad
\end{align*}
\]
With CCMP

\[ x_0 \geq x_1 \land x_2 = x_3 \]

First try:

\[
\begin{align*}
\text{cmp} & \quad r0, \quad r1 \\
\text{cmpge} & \quad r2, \quad r3 \\
bXX & \quad \text{good}
\end{align*}
\]

But with CCMP:

\[
\begin{align*}
\text{cmp} & \quad x0, \quad x1 \\
\text{ccmp} & \quad \text{ge}
\end{align*}
\]
With CCMP

\[ x_0 \geq x_1 \&\& x_2 = x_3 \]

First try:

\[
\begin{align*}
\text{cmp} &\quad r_0, r_1 \\
\text{cmpge} &\quad r_2, r_3 \\
\text{bXX} &\quad \text{good}
\end{align*}
\]

But with CCMP:

\[
\begin{align*}
\text{cmp} &\quad x_0, x_1 \\
\text{ccmp} &\quad x_2, x_3, \text{ge} \\
\text{b.eq} &\quad \text{good}
\end{align*}
\]
With CCMP

\[ x_0 \geq x_1 \land x_2 = x_3 \]

First try:

\[
\begin{align*}
\text{cmp} & \quad r_0, \ r_1 \\
\text{cmpge} & \quad r_2, \ r_3 \\
\text{bXX} & \quad \text{good}
\end{align*}
\]

But with CCMP:

\[
\begin{align*}
\text{cmp} & \quad x_0, \ x_1 \\
\text{ccmp} & \quad x_2, \ x_3, \ <\text{ne}>, \ \text{ge} \\
\text{b.eq} & \quad \text{good}
\end{align*}
\]
With CCMP

\[ x_0 \geq x_1 \land x_2 = x_3 \]

First try:

\[
\begin{align*}
\text{cmp} & \quad r0, \ r1 \\
\text{cmpge} & \quad r2, \ r3 \\
bXX & \quad \text{good}
\end{align*}
\]

But with CCMP:

\[
\begin{align*}
\text{cmp} & \quad x0, \ x1 \\
\text{ccmp} & \quad x2, \ x3, \ #0, \ \text{ge} \\
b.eq & \quad \text{good}
\end{align*}
\]
Outline

AArch64 Architecture
AArch64 Backend
Testing the Backend
Interesting Curiosities
  Load-store Patterns
  Templated Operands
  Conditional Compare
Creating the Backend
Future Ideas
Summary of Effort

What did it take to create the backend?

- 1.5 months on basic layout.
- Then 4 months implementing instructions systematically.
- Then 4 months on integration (ABI, bugs, PIC, TLS, ...).

Time was increased by desire for full MC layer support for all instructions.
Phase 1: Create a Solid Base

1. Compile *anything*:
   
   ```
   define void @foo() { ret void }
   ```

2. Create some way of creating a live value: global variables for us, could be function parameters.
   
   ```
   @src = global i32 0
   @dst = global i32 0
   define void @foo() {
       %val = load i32* @src
       store i32 %val, i32* @dst
       ret void
   }
   ```

3. Implement ELF (relocations); asm parsing; related instructions.
Phase 2: Implement the ISA

1. Systematically implement all scalar instructions, a slice at a time.
2. Make sure assembly/encoding/... perfect.
3. Instruction selection for obvious patterns.
4. Hope was that by the end most DAG structures covered by default.
5. Implement features occasionally when necessary instructions present: function calls, stack objects, ...
Phase 2 approach was mostly successful: compiled and ran “hello world” immediately. zlib after a small patch.
Phase 3: Make it Work

1. Phase 2 approach was mostly successful: compiled and ran “hello world” immediately. zlib after a small patch.

2. Failed on odd corners not corresponding neatly to a single instruction.

3. E.g. jump tables, stranger SELECT_CC variants, external symbols . . .

4. Finally implemented other known large-scale features: DWARF; exception-handling; TLS . . .
Outline

AArch64 Architecture
AArch64 Backend
Testing the Backend
Interesting Curiosities
  Load-store Patterns
  Templated Operands
  Conditional Compare
Creating the Backend
Future Ideas
Unimplemented Features

- MCJIT
- FastISel
- Other memory models.
- NEON support is ongoing.
- Production-quality assembler (GNU as directives...).
Refactoring

1. ConstantIslands pass
   - Bulk is identical to ARM.
   - Changes to target-specific details (addressing limits etc).
   - Problem: intermixed with Thumb narrowing and more.
   - Second problem: very difficult to test, needs massive functions.
Refactoring

1. ConstantIslands pass
   - Bulk is identical to ARM.
   - Changes to target-specific details (addressing limits etc).
   - Problem: intermixed with Thumb narrowing and more.
   - Second problem: very difficult to test, needs massive functions.

2. 128-bit float legalisation
   - Duplication from LegalizeTypes.
   - It’s *almost* completely illegal.
Infrastructure

What can we do to make AArch64 a fully supported target?

- There will only be simulators for a while yet.
- Build bots?
- Daily tests?
- LLVM testsuite??