VIXL

A runtime assembler C++ API

Agenda

- What is VIXL?
- What is it used for?
- What's new in VIXL?
- Conclusion & Q&A

What is VIXL

- A Runtime code-generation library
  - for ARM AArch64 now, AArch32 (Arm and Thumb-2) coming CY2016
  - A program driven assembler, macro assembler, disassembler, simulator and debugger
- Developed as the code generator for the Google V8 JavaScript AArch64 port
- Available through GitHub
  - https://github.com/armvixl/vixl
- Supports most of the instruction set
  - At EL1 (usertime)
  - With full NEON support
- Used by a number of projects
  - Android's ART, QEMU, HipHop, Mozilla, and more!

VIXL's Assembler

- Example of a loop:

```
        mov x0, 10
loop:
        subs x0, x0, 1
        bne loop
```

- VIXL

```
Label loop;
mov(x0, 10);
Bind(&loop);
subs(x0, x0, 1);
b(&loop, ne);
```

NEON support in the assembler

- Example of generating NEON code:

```
Ld1(v0.V8B, v1.V8B(), MemOperand(x0));
Saddl(v0.V8H(), v0.V8B(), v1.V8B());
Addv(h0, v0.V8H());
```

- Will produce this code:

```
ld1 {v0.8b, v1.8b}, [x0]
saddl v0.8h, v0.8b, v1.8b
addv h0, v0.8h
```

- Register methods (here, for example V8H()) determine
  - the number of lanes (8)
  - and size of the register (D or Q) (8*16bit = 128bits = Q)

VIXL's MacroAssembler

- While VIXL's Assembler will generate one instruction, the MacroAssembler gives you a lot more flexibility
  - AArch64 uses modified immediates
    - mov r0, 0x10001 is permitted
    - mov r0, 0x10002 is not!
  - MacroAssembler::Mov(r0, 0x10002) will do the job for you, using a pair of movz/movk or whatever combination of instructions necessary
  - The MacroAssembler ultimately calls the Assembler
- Simplifies your code without a deep knowledge of ARM AArch64
- Sometimes it might optimize your code
  - But it's still an assembler, not a compiler!

VIXL's MacroAssembler (2)

- Will perform simple tricks to shorten your generated code
  - For example:

```
Add(w0, w0, 0x10002);
```

  - Will generate:

```
mov w16, #0x8001
add x0, x0, w16, lsl #1
```

  - Instead of the usual movz/movk for 0x10002

What to do with the generated code

- In a JIT environment, simply run it
  - Natively,
  - Or Simulated
- Save it for future reuse
- Or generate an ELF file using elfio and link it to your application
  - Create a symbol for the generated functions using elfio

```
Elf_Word add_symbol(string_section_accessor& pStrWriter,
                    const char* str, Elf64_Addr value,
                    Elf_Xword size, unsigned char info,
                    unsigned char other, Elf_Half shndx );
symbols.add_symbol( strings, fn_name, fn_addr, fn_size, STB_GLOBAL,
                    STB_FUNC, 0, text_sec->get_index);
```

  - ELFio is available here: http://elfio.sourceforge.net

A quick example

- The headers

```
#include <stdio.h>
#include "vixl/a64/assembler-a64.h"
#include "vixl/a64/disasm-a64.h"

using namespace vixl;
```

A quick example (cont …)

```
int main(int argc, char ** argv) {
  const size_t bufSize = 1024;
  uint8_t* buffer = new uint8_t[bufSize];
  Assembler asm_(buffer, bufSize);

  asm_.orr(w0, wzr, 0x10001);
  asm_.mov(x1, x0);
  asm_.FinalizeCode();

  BufferDisassembler disasm_(buffer,
      asm_.CursorOffset());
}
```

A quick example (cont …)

```
class BufferDisassembler {
  Decoder decoder_;
  PrintDisassembler disasm_;
  public:
    BufferDisassembler(
        uint8_t* buffer, size_t size)
        : disasm_(stdout) {
      decoder_.AppendVisitor(&disasm_);
      uint32_t* instr =
          reinterpret_cast<uint32_t*>(buffer);
      for (; size > 0; size -= 4, instr++) {
        decoder_.Decode(
            reinterpret_cast<Instruction*>(instr));
      }
    }
};
```


ARM®

A quick example (cont …)

- Will output

```
0x00000000013ad010   320083e0   mov w0, #0x10001
0x00000000013ad014   aa0003e1   mov x1, x0
```

Another one using the macro assembler

```
asm_.Mov(w0, 0x10002);
asm_.Mov(x1, x0);
```

- Will output:

```
0x0000000000f3f010  52800040          mov w0, #0x2
0x0000000000f3f014  72a00020          movk w0, #0x1, lsl #16
0x0000000000f3f018  aa0003e1          mov x1, x0
```

- A lot more examples (and surely more complex) are available in the example directory

VIXL's Simulator

```
0x0000000002ff90b4  4c960a40            st4 {v0.4s, v1.4s, v2.4s, v3.4s}, [x18], x22
#    v0: 0x0f0e0d0c0b0a09080706050403020100 (7.00365e-30, 2.65846e-32, 1.00825e-34, 3.82047e-37) <- 0x00007ffd447edf15
#    v1: 0x1f1e1d1c1b1a19181716151413121110 (3.34819e-20, 1.27467e-22, 4.84942e-25, 1.84362e-27) <- 0x00007ffd447edf15
#    v2: 0x2f2e2d2c2b2a29282726252423222120 (1.58413e-10, 6.04532e-13, 2.30573e-15, 8.78905e-18) <- 0x00007ffd447edf15
#    v3: 0x3f3e3d3c3b3a39383736353433323130 (0.743122, 0.00284155, 1.08604e-05, 4.14886e-08) <- 0x00007ffd447edf15
#   x18: 0x00007ffd447edf1a
0x0000000002ff90b8  4c000e40            st4 {v0.2d, v1.2d, v2.2d, v3.2d}, [x18]
#    v0: 0x0f0e0d0c0b0a09080706050403020100 (3.69192e-236, 7.94993e-275) <- 0x00007ffd447edf1a
#    v1: 0x1f1e1d1c1b1a19181716151413121110 (8.56775e-159, 1.84632e-197) <- 0x00007ffd447edf1a
#    v2: 0x2f2e2d2c2b2a29282726252423222120 (1.98829e-81, 4.28794e-120) <- 0x00007ffd447edf1a
#    v3: 0x3f3e3d3c3b3a39383736353433323130 (0.000461414, 9.95833e-43) <- 0x00007ffd447edf1a
0x0000000002ff90bc  d29bda13            mov x19, #0xded0
#   x19: 0x000000000000ded0
0x0000000002ff90c0  f2a88fd3            movk x19, #0x447e, lsl #16
#   x19: 0x00000000447eded0
0x0000000002ff90c4  f2cfffb3            movk x19, #0x7ffd, lsl #32
#   x19: 0x00007ffd447eded0
0x0000000002ff90c8  3cc10660            ldr q0, [x19], #16
#    v0: 0x12030231302120111001000130201000 <- 0x00007ffd447eded0
#   x19: 0x00007ffd447edee0
0x0000000002ff90cc  3cc10661            ldr q1, [x19], #16
#    v1: 0x16070635342524151405043332232213 <- 0x00007ffd447edee0
#   x19: 0x00007ffd447edef0
0x0000000002ff90d0  3cc10662            ldr q2, [x19], #16
#    v2: 0x1a0b0a393829281918090837362726 17 <- 0x00007ffd447edef0
#   x19: 0x00007ffd447edf00
```

- Able to simulate whatever instruction VIXL can assemble
- Execution and register trace output available
- Extensive back-to-back testing against hardware
- Printf!

ARM®

VIXL Simulator (cont)

- Because everything is simulated
  - Every instruction executed is traced
  - Every register update is traced
  - Every memory load/store, with address information, is checked

- Can be seen as slow
  - No, it's not! if run on many cores, at let's say 3 or 4Ghz, it will be fast!

VIXL's Debugger

- gdb with fewer commands
  - p for print, x for examine, si to execute the current instruction

```
MacroAssembler masm(assm_buf, BUF_SIZE);
Decoder decoder;
Debugger debugger(&decoder);
…
// Generate the code for the example function.
Label start;
masm.Bind(&start);
masm.Brk();
masm.Nop();
. . .
debugger.RunFrom(
    masm.GetLabelAddress<Instruction*>(&start));
```

```
Hit breakpoint at pc=0x7ffe93ce2a40.
Next: 0x00007ffe93ce2a40  d4200000  brk #0x0
vixl> si
Next: 0x00007ffe93ce2a48  14000002  b #+0x8 (addr
0x7ffe93ce2a50)
vixl>
Next: 0x00007ffe93ce2a50  d2800f61  mov x1, #0x7b
vixl>
Next: 0x00007ffe93ce2a54  d2803902  mov x2, #0x1c8
vixl>
Next: 0x00007ffe93ce2a58  8b020020  add x0, x1, x2
vixl>
Next: 0x00007ffe93ce2a5c  d65f03c0  ret
vixl> p x1
x1 = 000000000000007b
vixl> p x2
x2 = 00000000000001c8
vixl> p x0
x0 = 0000000000000243
vixl> c
```


ARM®

Decoders

- The disassembler and simulator take action based on a common instruction decoder
- They use a visitor pattern
  - For each instruction, call the visitor methods of all registered classes
  - VisitAddSubImmediate()
  - VisitDaraProcessing3Source()
- Classes associated with instruction decoding implement this visitor interface
  - Simulator
  - Disassembler
  - Instrumentation
- Add your own visitors to get reports on what instructions have been encountered

Hacking with VIXL

- Most projects only use part of VIXL
  - Android's ART uses the assembler and the disassembler
  - QEMU uses the disassembler
- Google V8 and Mozilla use everything
  - The simulator is used to run tests on many fast cores
- The components are useful in isolation
  - We have already produced small command line tools for convenience, like a64disasm

VIXL as a programmable assembler

- Use VIXL to generate static code from C++, produce an object, and link it into the final executable
  - Plug VIXL into elfio to make a programmable assembler
  - Like an inline assembler, but more flexible
  - Like intrinsics, but for all instructions
- Example of an inner loop of an image processing operation:

```
Bind(&loop);
Ldrb(value, MemOperand(data));
Mul(value, value, coeff);
Lsr(value, value, 8);
Strb(value, MemOperand(data, 1, PostIndex));
Sub(length, length, 1);
Cbnz(length, &loop);
```

VIXL as a programmable assembler (2)

- Generate the code for different values of coeff
- Same code sequence can produce special cases for each value
- Create a symbol for each of the generated functions using elfio

```
symbols.add_symbol(strings, fn_name, fn_addr, fn_size, STB_GLOBAL,
                   STB_FUNC, 0, text_sec->get_index);
```

- Build the programmable assembler
- Run it to produce objects that can be linked into the rest of your project
- Like GAS, but more flexible
- Or wait from inside your app, generate the function upon request, and call it
  - You have created a dynamic C++ template instantiation!

```
template <int8_t coeff> mullcoeff(int8_t *data);
```

Dynamic coding example

```
Function mul(MacroAssembler& asm_, uint8_t coeff) {
  ptrdiff_t fn_start= asm_.CursorOffset();
  Label mul;
  asm_.Bind(&mul);
  Label loop;
  asm_.Bind(&loop);
  asm_.Ldrb(w11, MemOperand(x0));
  asm_.Mul(w11, w11, w1);
  asm_.Lsr(w11, w11, 8);
  asm_.Strb(w11, MemOperand(x0, 1, PostIndex));
  asm_.Ret();
  return Function(mul.location(), asm_.CursorOffset() - fn_start);
}
```

Dynamic coding example (cont…)

- The obvious cases

```
if (coeff != 1) {
  Label loop;
  asm_.Bind(&loop);
  if (coeff == 0) {
    asm_.Strb(wzr, MemOperand(x0, 1, PostIndex));
    asm_.Sub(x1, x1, 1);
    asm_.Cbnz(x1, &loop);
  } else {
    …
  }
}
asm_.Ret();
```

Dynamic coding example (cont…)

```
mul_0:
0x1fd2010  3800141f      strb wzr, [x0], #1
0x1fd2014  d1000421      sub x1, x1, #0x1 (1)
0x1fd2018  b5ffffc1      cbnz x1, #-0x8 (addr 0x1fd2010)
0x1fd201c  d65f03c0      ret
mul_1:
0x1fd2020  d65f03c0      ret
```

Dynamic coding example (cont…)

- What if coeff is a power of 2?

```
Label loop;
asm_.Bind(&loop);
asm_.Ldrb(w11, MemOperand(x0));
if (IsPowerOf2(coeff)) {
  unsigned pow2= CountTrailingZeros(coeff);
  asm_.Ubfx(w11, w11, 8 - pow2, pow2);
} else {
  …
}
```

Dynamic coding example (cont…)

```
mul_2:
0x1fd2024   3940000b        ldrb w11, [x0]
0x1fd2028   53077d6b        ubfx w11, w11, #7, #1
0x1fd202c   3800140b        strb w11, [x0], #1
0x1fd2030   d1000421        sub x1, x1, #0x1 (1)
0x1fd2034   b5ffff81        cbnz x1, #-0x10 (addr 0x1fd2024)
0x1fd2038   d65f03c0        ret
```

Dynamic coding example (cont…)

```
int main(int argc, char ** argv) {
  Function jumptable[255];
  const size_t bufSize = 1024;
  uint8_t* buffer = new uint8_t[bufSize];
  MacroAssembler asm_(buffer, bufSize);

  jumptable[0] = mul(asm_, 0);
  jumptable[1] = mul(asm_, 1);
  jumptable[2] = mul(asm_, 2);
  jumptable[11] = mul(asm_, 11);

  asm_.FinalizeCode();

}
```

- jumpTable can also be an std::set and upper_bound can be used
  - Because jumpTable[11] is the same as jumpTable[12]

Random instruction set testing

- To test your CPU by brute force
  - Execute a constrained random instruction stream, and see what breaks
- Use VIXL to generate this on the target under test rather than ahead of time
- Generate random 32bit values, and use VIXL to decide if the instruction is valid before executing natively
- Compare observable state after an instruction sequence between simulator and native, and detect deviation
  - The simulator can be compiled for AArch64, so simulated and native instructions can be run in the same environment
- Use simulator feedback to inform decisions about what to generate
  - Check address register is legal for generating memory access ops
  - Verify loops can exit in a reasonable amount of time

Where to use VIXL

- JITs: JavaScript, Java, Python, other scripting languages
- Dynamic code generation of optimized routine
- Testing
  - Random Instruction Stream (RIS) testing
  - Toolchain testing
- ISA experimentation: try out features of ARM's AArch64
  - Simple, fast, tested API
  - Integrated suite, ready to use on a new JIT project
- Teaching tool using a parsing expression grammar tool like PEG.js
  - Turn this JSON representation

    {"mnemonic":"mov","destination":{"type":"x","index":1}, ...
  - Into a Mov(x1, …) can be done in a matter of hours

What's new in VIXL

- VIXL1.8 introduced a few more things:
  - Full NEON support
  - Support for long branches
  - Improved handling of literal pools.
  - Support some `ic` and `dc` cache op instructions.
  - Support CRC32 instructions.
  - Support half-precision floating point instructions.
  - MacroAssembler support for `bfm`, `ubfm` and `sbfm`.
  - Other small bug fixes and improvements.
- We are now at 1.10 (July 2015)
  - improved literal handling,
  - a better build and test infrastructure,
  - And bug fixes

Conclusion

- Vixl is a toolkit
  - VIXL does the boring bits for you
- VIXL is available on github
  - https://github.com/armvixl/vixl
  - The license is a permissive free software license equivalent to the BSD 3-Clause license
- VIXL is maintained and developed
  - We are now (since July 2015) at 1.10
  - Supported by ARM
- … and extended
  - VIXL (= LXIV, ie 64 in roman numerals) for Aarch32 is coming CY2016
- A good reading start:
  - https://github.com/armvixl/vixl/blob/master/README.md