What’s new in VIXL?

TatWai Chong
Agenda

● What is VIXL?
● What is new?
● What is coming?
What is VIXL?

● Runtime code generation library
  ○ Assembler
  ○ MacroAssembler
  ○ Disassembler
  ○ Simulator (AArch64 only)

● Instruction coverage.
  ○ Core AArch32/64 v8.x
  ○ FP/Neon – fp16/32/64
  ○ SVE (on-going)
  ○ EL0 only
Assembler

- **Programmatic** C++ API ARMv8 assembly.
  - One-to-one mapping from API to ISA.
  - Input of API such as register, memory and constant are parameterized.
  - Assembler has a low run-time overhead.
  - `asm->mov(x1, 123);`
  - `asm->add(x0, x1, Operand(x2, LSL, 2));`

- **Debug features (debug mode only):**
  - Check immediate validity
  - Check sized of generated code
Macro-Assembler

● Abstract away the ISA limitations.
  ○ assm->mov(r0, 0x10002) // isn’t permitted, must be a valid constant.
  ○ masm->Mov(r0, 0x10002) // generate movz/movk automatically.

● Abstraction for
  ○ Stack pointer & Zero register access
  ○ Immediate and offset
  ○ Literal pools

● For flexibility, convenience and, especially safety.
  ○ Register scope utility allows scratch registers to be managed safely.

● Select the proper assembly instructions for each macro instruction independently.
  ○ e.g. analyze an immediate input and try to encode it in the most appropriate assembly instruction.
Disassembler

Supports all instructions emitted by VIXL assembler.

580036e2  ldr x2, pc+1756 (addr 0x7fdd4ba79720)
18003703  ldr w3, pc+1760 (addr 0x7fdd4ba79728)
...
92800002  mov x2, #0xffffffffffffffff
d2f00003  mov x3, #0x8000000000000000
...
91048c0a  add x10, x0, #0x123 (291)
9144882b  add x11, x1, #0x122000 (1187840)
...
d1000414  sub x20, x0, #0x1 (1)
d1044435  sub x21, x1, #0x111 (273)
Simulator

- Support all instructions emitted by the VIXL assembler.
  - Isn't cycle-accurate.
- Use-cases:
  - Testing
  - Debugging
    - Single-step execution
  - Stack analysis (spviewer)
  - Algorithm development
Simulator

- Provides execution traces along with options and API.
  - `__DoLog()`, `__DoPrintf()`
  - `--trace-sim`

- Traces can be used to build debuggers that can step backwards.
  - Using an internal prototype (`spviewer`).

```assembly
0x00007fbdc899a064  52800036            mov w22, #0x1
#    x22: 0x0000000000000001
0x00007fbdc899a068  72aff016            movk w22, #0x7f80, lsl #16
#    x22: 0x0000000007f80001
0x00007fbdc899a06c  1e2702d6            fmov s22, w22
#    v22: 0x000000000000000007f80001
0x00007fbdc899a070  f100029f            cmp x20, #0x0 (0)
# NZCV: N:0 Z:1 C:1 V:0
0x00007fbdc899a074  1e300600            fccmp s16, s16, #nzcv, eq
# NZCV: N:0 Z:1 C:1 V:0
```
What’s new?

● Semantic Versioning
  ○ Starting at 3.0.0
  ○ Updating VIXL in projects should be more manageable.
● CPU feature management and detection
● Armv8.x support.
  ○ PAuth
  ○ BTI
  ○ FP16
  ○ Dot product
  ○ RCpc
  ○ LSE
  ○ JSCVT
  ○ Complex operations
  ○ …
CPU feature management and detection

- Feature management
  - For whose who want to control which features are available in the code, independently of the current platform.
  - VIXL provides feature set manipulation (class `CPUFeatures`) and audit (class `CPUFeaturesAuditor`).

- Feature detection
  - For whose who want to enable everything (include optional features) the current platform supports.
  - The proper way to implement feature detection is not obvious.
  - VIXL provides `CPUFeatures::InferFromOS()` to detect features.
Pointer Authentication (PAuth)

- Used to ensure functions return to the location expected by the program.
- For software stack protection and control flow integrity (CFI)
- PAC* and AUT*

Example

```c
__Paciasp();  // pacia lr, sp
__Autiasp();  // autia lr, sp
```
Branch Target Indicators (BTI)

- Mitigate the ability of attacker to run arbitrary code.
- Mark valid targets of indirect branches, so program flow can’t reach anywhere.
- BTI <Target>
  - BTI C – Can be targeted by call (function entry).
  - BTI J – Can be targeted by jump (jump table).
- Example
  ```
  __ Bind(&Label, BTI_j);  or  assm->bind(&Label);
  assm->bti(BTI_j);
  ```
Half precision extension (FP16)

• Float16
  ○ 16-bit IEEE 754 floating point
  ○ Supported in NEON and SVE extension.

• Example
  __ Fmov(v0.V4H(), RawbitsToFloat16(0x7c2f));
  __ Fmov(v1.V4H(), Float16(0.5));
  __ Fadd(v8.V4H(), v1.V4H(), v0.V4H());
Dot product

- SDOT & UDOT
- Example

```c
__ Udot(v18.V4S(), v0.V16B(), v1.V16B());
```
Release Consistency processor consistency (RCpc)

- *In addition to RCsc model.* ARMv8.3-RCpc provides Load-AcquirePC instructions.
  - LDAPR*, LDAPUR*

- Better support for RCpc as a more relaxed memory model than RCsc.
  - Better fit with C++11 memory_order_acquire/release semantics.

- Example
  ```
  __ Stlr(x0, MemOperand(x19, i)); // Thread a
  ...__
  __ Ldapr(x1, MemOperand(x19, i)); // Thread b
  ```
Large System Extension (LSE)

- atomics - CAS/SWP/LD[op]
- Reduce runtime overhead in big system, especially at high thread counts.
- Example
  
  __ Cas(w1, w0, MemOperand(x21));
  __ Swpb(w0, w1, MemOperand(x20));
JavaScript conversion (JSCVT)

● FJCVTZS
  ○ Converts JavaScript-style double-precision floating-point to 32-bit integer.
  ○ Checks if the number converted was an integer.

● Example
  __ Fmov(d0, 1.5);
  __ Fjcvtzs(w0, d0);
Complex number support

- Complex numbers \((a+bi)\) are packed in a vector register.
  - FCMLA - Complex multiply-accumulate
  - FCADD - Complex addition.

- Example
  ```
  __ Movi(v1.V2D(), 0x40A0000040400000, 0x40A0000040400000); // (5i, 3)
  __ Movi(v2.V2D(), 0x4040000040A00000, 0x4040000040A00000); // (3i, 5)
  __ Fcmla(v31.V4S(), v1.V4S(), v2.S(), 0, 90);
  ```
Scalable vector extension (SVE)

- Scalable vector length – 128 to 2048 bits
- Vector length agnostic (VLA)
- No need to recompile compiler intrinsics or rewrite SVE assembly.

Per-lane predication
SVE is coming from VIXL

- Currently on a public feature branch.
  - [https://git.linaro.org/arm/vixl.git/log/?h=sve](https://git.linaro.org/arm/vixl.git/log/?h=sve)
- Major useful functionality should be available by Q4.
- VIXL’s Simulator supports configurable vector length.
  - Keep VLA semantic in VIXL.

### Daxpy kernel code

<table>
<thead>
<tr>
<th>Neon code</th>
<th>VIXL-style SVE code</th>
</tr>
</thead>
<tbody>
<tr>
<td>.loop:</td>
<td>__ Ld1d(z1.VnD()), p0.Zeroing(), SVEMemOperand(x0, x4, LSL, 3));</td>
</tr>
<tr>
<td>1dr d1, [x0, x4, lsl #3] // d1=x[i]</td>
<td>__ Ld1d(z2.VnD()), p0.Zeroing(), SVEMemOperand(x1, x4, LSL, 3));</td>
</tr>
<tr>
<td>1dr d2, [x1, x4, lsl #3] // d2=y[i]</td>
<td>__ Fmla(z2.VnD(), p0.Merging(), z1.VnD(), z0.VnD())</td>
</tr>
<tr>
<td>fmadd d2, d1, d0, d2 // d2+=x[i]*a</td>
<td>__ St1d(z2.VnD()), p0, SVEMemOperand(x1, x4, LSL, 3));</td>
</tr>
<tr>
<td>str d2, [x1, x4, lsl #3] // y[i]=d2</td>
<td>__ Incp(x4, ALL);</td>
</tr>
<tr>
<td>add x4, x4, #1 // i+=1</td>
<td></td>
</tr>
</tbody>
</table>
SVE is coming from VIXL

- Support register (Z/P) and memory tracing.

<table>
<thead>
<tr>
<th>Address (of result)</th>
<th>Expected</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000000429c80 (result + 0):</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>0x000000000429c88 (result + 8):</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x000000000429ec78 (result + 24568):</td>
<td>0x0000000000000000</td>
<td>0x0000000000000000</td>
</tr>
<tr>
<td>0x000000000429ec80 (result + 24576):</td>
<td>0x1714110e0b080502 0x1613100d0a070401</td>
<td>0x1613100d0a070401</td>
</tr>
<tr>
<td>0x000000000429ec88 (result + 24584):</td>
<td>0x2f2c292623201d1a 0x2e2b2825221f1c19</td>
<td>0x2e2b2825221f1c19</td>
</tr>
<tr>
<td>0x000000000429ec90 (result + 24592):</td>
<td>0x4744413e3b383532 0x4643403d3a373431</td>
<td>0x4643403d3a373431</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- z1<255:128>: 0x 03 02 -> 0x00007fff00000008
- z1<383:256>: 0x 05 04 -> 0x00007fff00000018
- z2<255:128>: 0x f003 f002 -> 0x00007fff00000008
- z2<383:256>: 0x f005 f004 -> 0x00007fff00000018
- z3<255:128>: 0x 7f80f003 7f80f002 -> 0x00007fff00000008
- z3<383:256>: 0x 7f80f005 7f80f004 -> 0x00007fff00000018
- z4<255:128>: 0x7ff0f0017f80f0037ff0f0017f80f002 -> 0x00007fff00000008
- z4<383:256>: 0x7ff0f0017f80f0057ff0f0017f80f004 -> 0x00007fff00000018
What else is coming?

- Remaining bits of Armv8.x
  - AMTE (memory coloring)
- Better API documentation
- C++14
- Hardening
  - Static analysis
  - Code coverage
- Trace tools (spviewer)
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org