ARM64 ISA (Instruction Set Architecture)

An Initial Encounter

20 November 2012

Sverre Jarp, CERN openlab

64-bit ISA

- What is an "ISA" anyway ?
- ISA describes "only"
  - what can be done
    - "multiply two floating-point registers"
      - scalar or vector mode
- μ-architecture adds:
  - how is it done and what will it cost
    - "One single execution unit can handle this instruction"
    - "The Flp multiply itself takes 5 cycles"
- The μ-arch is not yet published
  - and, will probably come in several flavours

Some “tidbits” about μ-arch

- Initial info (Cortex A57 specs from ARM):
  - 1 – 4x SMP on chip
  - Increased peak instruction throughput via duplication of execution resources.
  - 3-wide decode bandwidth
  - High-capacity register renaming provides 3-wide, large-instruction rename bandwidth.
  - Support for 8 issue slots and up to 128 instructions in flight
  - 64KB page support

Applied Micro:
X-Gene processor at 3 GHz with 4-wide decode (?)

Some AArch32 Cortex-A9 DP latencies

- From FPU Technical Reference Manual:

Instructions	Latencies	Throughput
FADD, FSUB	4	1
FMUL	6	2
FMAC, FNMAC	9	2
FCPY, FABS, FNEG	1	1
FDIV	25	20
FSQRT	32	28
FCMP	1	1

Not “fused” but “chained”

AArch64 (A64) – a high-level view

- Not backwards compatible
  - But, an enhanced AArch32 (A32) execution stage will co-exist (see later)
- Almost all features are now integrated:
  - With AArch32: FPU, SIMD, Neon, Cryptography were add-ons
- Load/Store architecture
  - Multiple addressing modes
- Registers:
  - 32 64-bit integer
  - 32 128-bit packed vector
- Fixed instruction length
  - 32 bits:
    - opcode (10-bits!) + dest + src3 + src2 + src1
    - <name>{subtype>} <containers>
- Instructions are typically “ternary”

Still optional

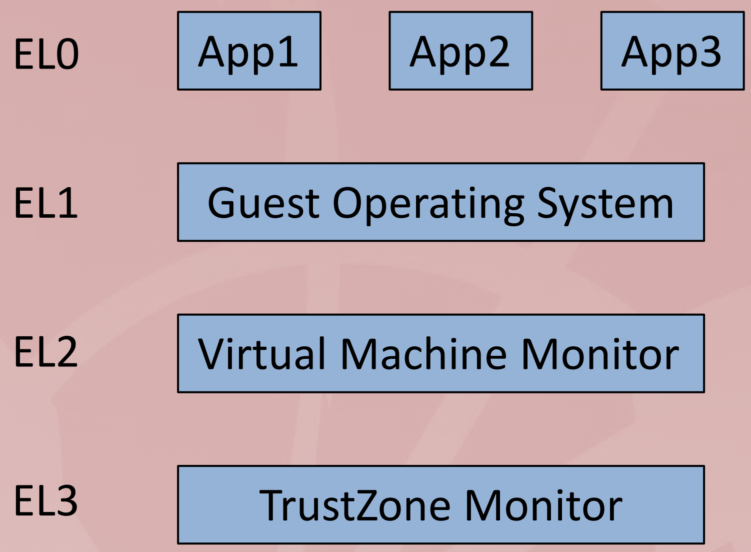
This would imply 2^{10} instructions (100 pages with 10 each) !!!

Changes from AArch32

- Floating-point entirely integrated
  - Based on IEEE-754 (2008)
- Many new instructions
  - Also, several modified instructions
  - Instructions with more than one destination split in two:
    - XXX1 and XXX2 (Vector permutations, for instance)
- Different register sets
  - Bigger integer set (now: 32)
  - Larger vector registers (size: 128b)
- No predication of instructions
  - But, some conditional moves, etc.

Exception Layers

- Four levels, in total:



AArch64 will support AArch32 at a lower privilege level.

The AArch64 ABI

- Register conventions:

Register	Special	Role
SP		Stack Pointer
R30	LR	Link Register
R29	FP	Frame Pointer
R28-R19		Callee-saved
R18		Platform Register
R17	IP1	Intra-procedural
R16	IP0	Intra-procedural
R15-R9		Temporary
R8		Indirect result location
R7-R0		Parameters/Results

Register	Special	Role
V31-V16		Temporary
V15-V8		Callee-saved
V7-V0		Parameters/Results

Operand containers

- Usually the register type:
  - Integer:

	Width
W	32-bit integer
X	64-bit integer

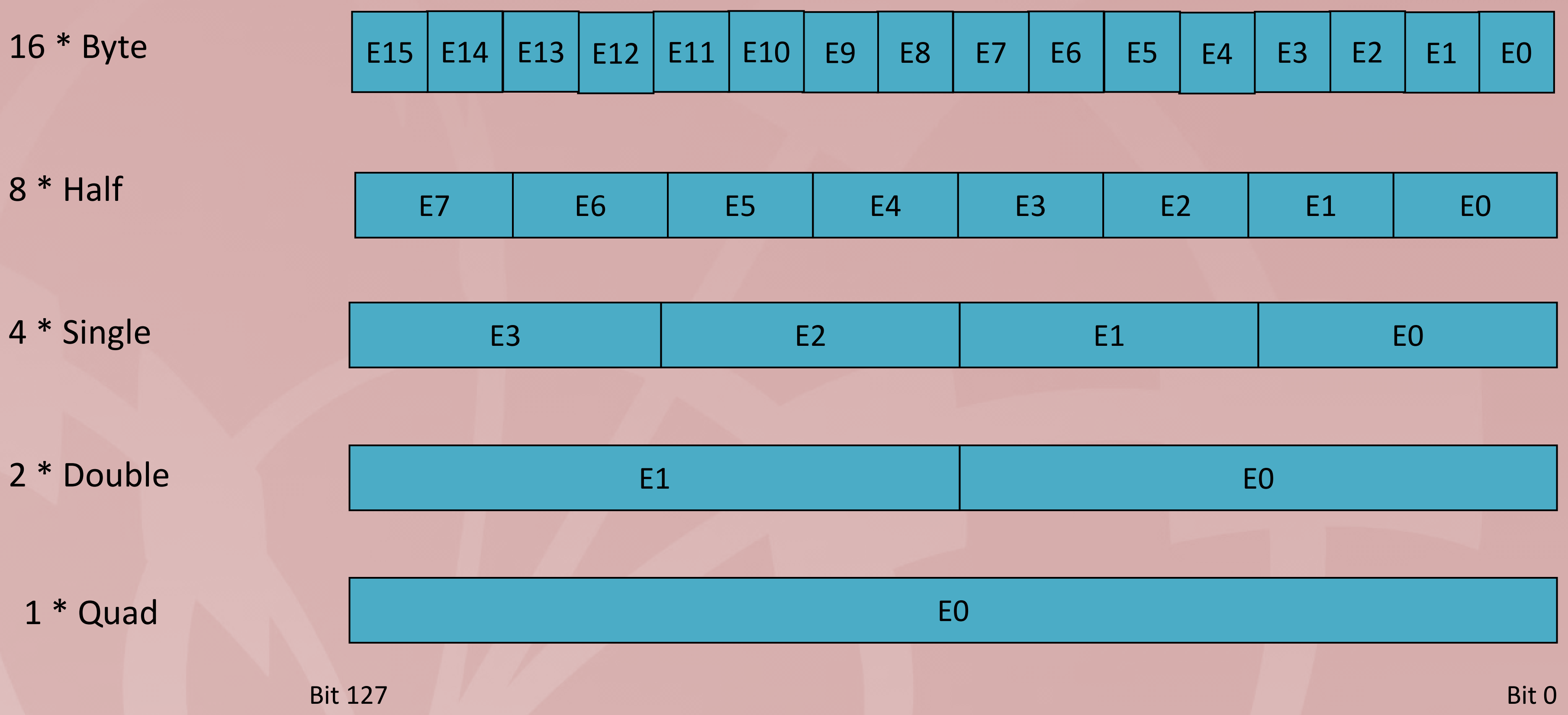
  - Packed:

	Width
B	8-bit: scalar
H	16-bit: scalar or float (HP)
S	32-bit: scalar or float (SP)
D	64-bit: scalar or float (DP)
Q	128-bit: scalar

Note the different notation (compared to x86) of “packed” and “scalar” !

Packed operands

- Similar to SSE on x86:



Subtypes

- Instruction suffix:
  - Load-Store
  - Sign/Zero Extend

	Subtype
B	Byte
SB	Signed Byte
H	Halfword
SH	Signed Halfword
W	Word
SW	Signed Word

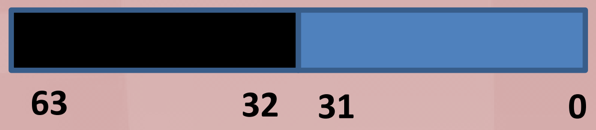
  - Register Width Changes:

	Subtype
H	High (dst gets top half)
N	Narrow (dst < src)
L	Long (dst > src)
W	Wide (dst == src1, src1 > src2)

Not discussed any further →

Working with integer registers

- Straight forward
  - Example:
    - "ADD X16, X14, X15"
    - "ADD X16, X14, W15, SXT" // sign-extend
    - "ADD X16, X15, #42" // immediate
  - 32-bit mode:
    - "ADD W16, W14, W15"
    - Upper 32 bits:
      - Ignored from a source
      - Zeroed in a destination
      - Right shifts/rotates inject at bit 31



  - Register 31:
    - Stack Pointer (SP)
      - When used as a load/store base register: WSP|SP
    - Zero otherwise:
      - WZR|XZR

FP/SIMD registers

- Holds both scalar floating-point and vector operands:
  - SIMD scalar registers: Bn, Hn, Sn, Dn, Qn
  - SIMD vector registers: Vn.16B, Vn.8H, Vn.4S, Vn.2D
  - Or, half the size: Vn.8B, Vn.4H, Vn.2S, Vn.1D
  - SIMD vector element: Vn.B[i], Vn.H[i], Vn.S[i], Vn.D[i]
  - SIMD register list: {V4.4S – V7.4S}
  - Or ,element list: {V4.4S – V7.4S}[3]

Addressing modes

- Multiple variants:
  - Base register (no offset)
  - Base plus offset
    - Immediate offset (scaled 12-bit unsigned, unscaled 9-bit-signed)
    - Base plus 32-/64-bit register (optionally scaled)
  - Pre-indexed
    - Unscaled 9-bit signed
  - Post-indexed
    - Unscaled 9-bit signed
  - Literal (PC-relative), for loads of 32-bits or larger

- Not all modes available for a given LD/ST type:
  - Exclusive/ Acquire/Release
  - Register
  - Register pair

Control Flow Instructions

- Conditional Branch:
  - B.cond label
  - CB(N)Z Wn|Xn, label
  - TB(N)Z Wn|Xn, #uimm6, label

- Unconditional Branch (and Link):
  - B label
  - BL label
  - BR Xm
  - BLR Xm
  - RET {Xm}   X30 (LR) as default

LD/ST single register

- Multiple subtypes:

Mnemonic	Operands
LDR	Wt|Xt, addr
LDR(B|H)	Wt, addr
LDRS(B|H)	Wt|Xt, addr
LDRSW	Xt, addr

- Fewer stores:

No change of direction !

Mnemonic	Operands
STR	Wt|Xt, addr
STR(B|H)	Wt, addr

LD/ST register pair

(also: non-temporal)

- Limited subtypes:

Mnemonic	Operands
LD(N)P	Wt1|Xt1, Wt2|Xt2, addr
n.a.	
n.a.	
LDPSW	Xt1, Xt2, addr

- Addr:
  - Base plus s-7-s offset
  - Pre-indexed; not for N?
  - Post-indexed; not for N?

Mnemonic	Operands
ST(N)P	Wt1|Xt1, Wt2|Xt2, addr
n.a.	

Integer/Logical (immediate)

- ADD(S) Xd, Xn, #aimm // S sets the condition flag
  - Variants with:
    - SUB, replacing ADD
    - W registers
- Aliases for providing: CMP and MOV

- AND(S) Xd, Xn, #bimm64
  - Variant with W register and bimm32
    - bimm is a repetitive pattern
- Also EOR and ORR
- TST (bitwise test) is aliased to ANDS

FLP/SIMD Scalar Memory Access

- LD/ST address
- Unscaled versions
- LD/ST Pair
- Non-temporal versions

Mnemonic	Operands
LDR	Bt|Ht|St|Dt|Qt, addr
LDUR	Bt|Ht|St|Dt|Qt, [base,#simm9]
LDP	St1|Dt1|Qt1, St2|Dt2|Qt2, addr
LDNP	St1|Dt1|Qt1, St2|Dt2|Qt2, [base, #imm]

Mnemonic	Operands
STR	Bt|Ht|St|Dt|Qt, addr
STUR	Bt|Ht|St|Dt|Qt, [base,#simm9]
STP	St1|Dt1|Qt1, St2|Dt2|Qt2, addr
STNP	St1|Dt1|Qt1, St2|Dt2|Qt2, [base, #imm]

Flp Move Register; Move Immediate; Convert; Round

- FMOV (Register – register):
  - 32-bits: Sd ← Sn, Wd ← Sn, Sd ← Wn
  - 64 bits: Dd ← Dn, Xd ← Dn, Dd ←Xn
  - High-order 64-bits: Xd ← Vn.D[1], Vn.D[1] ← Xn
- Immediate:
  - FMOV Sd, #fpimm8, but FMOV Sd, WZR
  - FMOV Dd, #fpimm8, but FMOV Dd, XZR
- Convert precision:
  - All combinations between Hn, Sn, and Dn
- Round to Integral:
  - FRINTr Sd, Sn or FRINTr Dd, Dn

Rounding modes (r):
- N (nearest, ties to even)
- A (nearest, ties away from zero)
- P (towards +inf.)
- M (towards –inf.)
- Z (towards zero)
- I (Use FPCR rounding)
- X (use FPCR with exactness check)

Flp Convert to/from Int

- Convert Single/Double to Signed/Unsigned:
  - FCVTr(S|U) Wd|Xd, Sn|Dn
  - Rounding modes (r):
    - NAPMZ
- Convert Signed/Unsigned to Single/Double:
  - (S|U)CVTF Sd|Dd, Wn|Xn
    - Using FPCR rounding mode

Also: Convert to/from Fixed-point

FLP Arithmetic

- One source:
  - FABS, FNEG, FSQRT   (Single or Double)
- Two sources:
  - FADD, FDIV, FMUL, FNMUL, FSUB
- Min/Max:
  - FMAX(NM), FMIN(NM)
- Multiply-Add (three sources):
  - FMADD, FMSUB, FNMADD, FNMSUB

FLP Arithmetic (cont'd)

- Flag-setting compare:
  - FCMP, FCMPE Sn, Sm|#0.0 (same for D)
  - FCMPP, FCMPPE Sn, Sm, #uimm4, cond
- Select:
  - FCSEL Sd, Sn, Sm, cond (same for D)

Vector Data Movement

- Duplicate (DUP) vector element
- Insert (INS) vector element
- Unsigned move (UMOV)
- Signed move (SMOV)

- Other moves (MOV) are mapped back to one of the above.

Vector Arithmetic

- Absolute difference
- Add, Sub (int or flp)
- Saturating Add (signed, unsigned)
- And, Exclusive Or, Or, Or Not, (8B or 16B)
- Bitwise operations
- Compare (sint, uint or flp)
- Absolute Compare (flp)
- Divide (flp)
- Halving Add, Subtract (int)
- Max and Min (sint, uint, flp)

Vector Arithmetic

- Multiply & Add/Subtract (int or flp)
- Multiply (int or flp)
- Polynomial Multiply (bytes)
- Reciprocal divide/sqrt step (flp)
- Saturating Double Multiply High Half (int)
- (Saturating) (Rounding) Shift Left (int)
- Saturating Subtract (int)
- Rounding Halving Add (int)

Let's start to wind down

(otherwise we'll be here until tomorrow)

- Hundreds of instructions still to go:
  - Scalar Arithmetic
  - Vector/Scalar Widening/Narrowing Arithmetic
  - Vector/Scalar Unary Arithmetic
  - Vector/Scalar-by-Element Arithmetic
  - Vector Permute
  - Vector Immediate
  - Vector/Scalar shifts
  - Vector/Scalar FLP/INT convert
  - Vector/Scalar Reduce
  - Vector Pairwise Arithmetic
  - Vector Table Lookup
  - Vector LD/ST Single/Multiple Structures
  - Optional Crypto Extensions
  - System Instructions

Vector Unary Arithmetic

(only one source)

- Absolute Value (int, sint, flp)
- Negate (int, sint saturating, flp)
- Count Bits (sign, zero, non-zero)
- Bitwise Invert
- Add Long Pair (signed, unsigned)
- FLP Convert (H → S or S → D; and back)
- Integer Narrow (whole family)
- Recipical estimate (Int, Flp)
- Reciprocal SQRT estimate (Int, Flp)
- SQRT
- Reverse: RBIT, REV16, REV32, REV64

Vector Permute

- From two input vector registers:
  - Bitwise Extract (via immediate index)
    - EXT
  - Vector Element Transpose
    - TRN1/TRN2
  - Vector Element Unzip
    - UZP1/UZP 2
  - Vector Element Zip
    - ZIP1/ZIP2

Two destination registers need two instructions (unlike AArch32)

Conclusion

- (Very) large instruction set
- Multiple sub-sets:
  - Integer or floating-point
  - Individual scalar or vector
  - Compute or manipulate
  - "Special function"
- Need to wait for μ-arch to understand "how", rather than "what"
  - Latencies
  - Superscalar design
- Expect more information to be made available over time

Seems unlikely that compilers can "master" the whole set! But, expect both gcc and armcc to provide intrinsics (as they do for AArch32)