UNIVERSITÀ DEGLI STUDI DI TRIESTE
Dipartimento di Ingegneria e Architettura

# Arithmetic in LEGv8

## A. Carini – Digital System Architectures

# Multiply in LEGv8

- To produce a properly signed or unsigned 128-bit product, LEGv8 has three instructions:
  - multiply (**MUL**),
  - signed multiply high (**SMULH**) and
  - unsigned multiply high (**UMULH**).
- To get the integer 64-bit product, the programmer use MUL.
- To get the upper 64 bits of the 128-bit product, the programmer uses either SMULH or UMULH, depending on the types of multiplier and multiplicand.

- LEGv8 multiply instructions do not set the overflow condition code, so it is up to the software to check to see if the product is too big to fit in 64 bits.
  - There is no overflow if the upper 64 bits is 0 for UMULH or the replicated sign of the lower 64 bits for SMULH.

# Divide in LEGv8

- To handle both signed integers and unsigned integers, LEGv8 has two instructions:
  - **signed divide (SDIV)** and
  - **unsigned divide (UDIV)**.
- The common hardware support for multiply and divide allows LEGv8 to provide a single pair of 64-bit registers that are used both for multiply and divide.

- LEGv8 divide instructions ignore overflow: software must determine whether the quotient is too large.
- In addition to overflow, division can also result in an improper calculation: division by 0.
- LEGv8 software must check the divisor to discover division by 0 as well as overflow.

# Multiply and Divide in LEGv8

| | | | |
|---|---|---|---|
| multiply | MUL X1, X2, X3 | X1 = X2 × X3 | Lower 64-bits of 128-bit product |
| signed multiply high | SMULH X1, X2, X3 | X1 = X2 × X3 | Upper 64-bits of 128-bit signed product |
| unsigned multiply high | UMULH X1, X2, X3 | X1 = X2 × X3 | Upper 64-bits of 128-bit unsigned product |
| signed divide | SDIV X1, X2, X3 | X1 = X2 / X3 | Divide, treating operands as signed |
| unsigned divide | UDIV X1, X2, X3 | X1 = X2 / X3 | Divide, treating operands as unsigned |

# Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - –2.34 × 1056 ← normalized
  - +0.002 × 10–4 ← Not normalized
  - +987.02 × 109 ← Not normalized

- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types float and double in C

# Floating Point Standard IEEE Std 754-1985

- Two representations: Single precision (32-bit) and Double precision (64-bit)

| | single: 8 bits<br>double: 11 bits | single: 23 bits<br>double: 52 bits |
|---|---|---|
| S | Exponent | Fraction |

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalized significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

# Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

# Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

# Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

# Denormalized Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
  - Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

# IEEE Std 754-1985 Summary

| Single precision | | Double precision | | Object represented |
|---|---|---|---|---|
| Exponent | Fraction | Exponent | Fraction | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | Nonzero | 0 | Nonzero | ± denormalized number |
| 1–254 | Anything | 1–2046 | Anything | ± floating-point number |
| 255 | 0 | 2047 | 0 | ± infinity |
| 255 | Nonzero | 2047 | Nonzero | NaN (Not a Number) |

# Overflow and underflow

- As for integer operations, floating-point arithmetic operation can originate *overflows.*
- ***overflow*** here means that
    - the exponent is too large to be represented in the exponent field.

- Floating point offers a new kind of exceptional event as well: the nonzero fraction we are calculating could become so small that it cannot be represented.
- We call this event ***underflow***:
    - it occurs when the negative exponent is too large to fit in the exponent field.

# Managing Overflows and underflows

- What should happen on an overflow or underflow to let the user know that a problem occurred?
- LEGv8 can raise an **exception**, also called an **interrupt** on many computers.
- An exception or interrupt is essentially an *unscheduled procedure call*.
    - The address of the instruction that overflowed is saved in a register, and
    - the computer jumps to a predefined address to invoke the appropriate routine for that exception.
    - In some situations the program can continue after corrective code is executed.

# Floating-Point Instructions in LEGv8

- LEGv8 supports the IEEE 754 single-precision and double-precision formats with these instructions:
  - Floating-point addition, single (**FADDS**) and addition, double (**FADDD**)
  - Floating-point subtraction, single (**FSUBS**) and subtraction, double (**FSUBD**)
  - Floating-point multiplication, single (**FMULS**) and multiplication, double (**FMULD**)
  - Floating-point division, single (**FDIVS**) and division, double (**FDIVD**)
  - Floating-point comparison, single (**FCMPS**) and comparison, double (**FCMPD**)
- Separate floating-point registers:
  - called **S0**, **S1**, **S2**, … for single precision and **D0**, **D1**, **D2**, . . . for double precision.
  - Single precision registers are just the lower half of double-precision registers.
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - **LDURS**, **LDURD**
  - **STURS**, **STURD**

# LEGv8 floating-point assembly language

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | FP add single | FADDS S2, S4, S6 | S2 = S4 + S6 | FP add (single precision) |
| | FP subtract single | FSUBS S2, S4, S6 | S2 = S4 - S6 | FP sub (single precision) |
| | FP multiply single | FMULS S2, S4, S6 | S2 = S4 × S6 | FP multiply (single precision) |
| | FP divide single | FDIVS S2, S4, S6 | S2 = S4 / S6 | FP divide (single precision) |
| | FP add double | FADDD D2, D4, D6 | D2 = D4 + D6 | FP add (double precision) |
| | FP subtract double | FSUBD D2, D4, D6 | D2 = D4 - D6 | FP sub (double precision) |
| | FP multiply double | FMULD D2, D4, D6 | D2 = D4 × D6 | FP multiply (double precision) |
| | FP divide double | FDIVD D2, D4, D6 | D2 = D4 / D6 | FP divide (double precision) |
| Conditional branch | FP compare single | FCMPS S4, S6 | Test S4 vs. S6 | FP compare single precision |
| | FP compare double | FCMPD D4, D6 | Test D4 vs. D6 | FP compare double precision |
| Data transfer | Load single FP | LDURS S1, [X23,100] | S1 = Memory[X23 + 100] | 32-bit data to FP register |
| | Load double FP | LDURD D1, [X23,100] | D1 = Memory[X23 + 100] | 64-bit data to FP register |
| | Store single FP | STURS S1, [X23,100] | Memory[X23 + 100] = S1 | 32-bit data to memory |
| | Store double FP | STURD D1, [X23,100] | Memory[X23 + 100] = D1 | 64-bit data to memory |

# LEGv8 floating-point machine language

| Name | Format | Example | | | | | Comments |
|---|---|---|---|---|---|---|---|
| FADDS | R | 241 | 6 | 10 | 4 | 2 | FADDS S2, S4, S6 |
| FSUBS | R | 241 | 6 | 14 | 4 | 2 | FSUBS S2, S4, S6 |
| FMULS | R | 241 | 6 | 2 | 4 | 2 | FMULS S2, S4, S6 |
| FDIVS | R | 241 | 6 | 6 | 4 | 2 | FDIVS S2, S4, S6 |
| FADDD | R | 243 | 6 | 10 | 4 | 2 | FADDD D2, D4, D6 |
| FSUBD | R | 243 | 6 | 14 | 4 | 2 | FSUBD D2, D4, D6 |
| FMULD | R | 243 | 6 | 2 | 4 | 2 | FMULD D2, D4, D6 |
| FDIVD | R | 243 | 6 | 6 | 4 | 2 | FDIVD D2, D4, D6 |
| FCMPS | R | 241 | 6 | 8 | 4 | 0 | FCMPS S4, S6 |
| FCMPD | R | 243 | 6 | 8 | 4 | 0 | FCMPD D4, D6 |
| LDURS | D | 1506 | 100 | 0 | 4 | 2 | LDURS S2, [X23,100] |
| LDURD | D | 2018 | 100 | 0 | 4 | 2 | LDURD S2, [X23,100] |
| STURS | D | 1504 | 100 | 0 | 4 | 2 | STURS D2, [X23,100] |
| STURD | D | 2016 | 100 | 0 | 4 | 2 | STURD D2, [X23,100] |
| Field size | | 11 bits | 5 or 9 bits | 6 or 2 bits | 5 bits | 5 bits | All LEGv8 instructions 32 bits |

# Example

- The LEGv8 code to load two single precision numbers from memory, add them, and then store the sum might look like this:

```
LDURS  S4, [X28,c]  //  Load 32-bit F.P. number into S4
LDURS  S6, [X28,a]  //  Load 32-bit F.P. number into S6
FADDS  S2, S4, S6  //  S2 = S4 + S6 single precision
STURS  S2, [X28,b]  //  Store 32-bit F.P. number from S2
```

# Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

- fahr in S12, result in S0, literals in global memory space
- Compiled LEGv8 code:

```
f2c:
    LDURS S16, [X27,const5]   // S16 = 5.0 (5.0 in memory)
    LDURS S18, [X27,const9]   // S18 = 9.0 (9.0 in memory)
    FDIVS S16, S16, S18       // S16 = 5.0 / 9.0
    LDURS S18, [X27,const32]  // S18 = 32.0
    FSUBS S18, S12, S18       // S18 = fahr – 32.0
    FMULS S0, S16, S18        // S0 = (5/9)*(fahr – 32.0)
    BR LR                     // return
```

# Example: Array Multiplication

- C = C + A × B
  - All 32 × 32 matrices, 64-bit double-precision elements

- C code:

```
void mm (double c[][], double a[][], double b[][]) {
  int i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1)
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}
```

- Addresses of x, y, z in X0, X1, X2, and i, j, k in X19, X20, X21

# Example: Array Multiplication

- LEGv8 code:

```
mm:...
    LDI X10, 32              // X10 = 32 (row size/loop end)
    LDI X19, 0               // i = 0; initialize 1st for loop
L1: LDI X20, 0               // j = 0; restart 2nd for loop
L2: LDI X21, 0               // k = 0; restart 3rd for loop
    LSL X11, X19, 5          // X11 = i * 2 5 (size of row of c)
    ADD X11, X11, X20// X11 = i * size(row) + j
    LSL X11, X11, 3          // X11 = byte offset of [i][j]
    ADD X11, X0, X11 // X11 = byte address of c[i][j]
    LDURD D4, [X11,#0]       // D4 = 8 bytes of c[i][j]
L3: LSL X9, X21, 5           // X9 = k * 2 5 (size of row of b)
    ADD X9, X9, X20          // X9 = k * size(row) + j
    LSL X9, X9, 3            // X9 = byte offset of [k][j]
    ADD X9, X2, X9           // X9 = byte address of b[k][j]
    LDURD D16, [X9,#0]       // D16 = 8 bytes of b[k][j]
```

# Example: Array Multiplication

```
LSL X9, X19, 5              // X9 = i * 2 5 (size of row of a)
ADD X9, X9, X21             // X9 = i * size(row) + k
LSL X9, X9, 3               // X9 = byte offset of [i][k]
ADD X9, X1, X9              // X9 = byte address of a[i][k]
LDURD D18, [X9,#0]          // D18 = 8 bytes of a[i][k]
FMULD D16, D18, D16         // D16 = a[i][k] * b[k][j]
FADDD D4, D4, D16           // f4 = c[i][j] + a[i][k] * b[k][j]
ADDI X21, X21, 1            // $k = k + 1
CMP X21, X10                // test k vs. 32
B.LT L3                     // if (k < 32) go to L3
STURD D4, [X11,0]           // c[i][j] = D4
ADDI X20, X20, #1           // $j = j + 1
CMP X20, X10                // test j vs. 32
B.LT L2                     // if (j < 32) go to L2
ADDI X19, X19, #1           // $i = i + 1
CMP X19, X10                // test i vs. 32
B.LT L1                     // if (i < 32) go to L1
```

# Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
    - **guard** and **round** The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.
    - **sticky bit** A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults

# The BIG Picture

- Bit patterns have no inherent meaning.
- They may represent signed integers, unsigned integers, floating-point numbers, instructions, character strings, and so on.
- What is represented depends on the instruction that operates on the bits in the word.

- The major difference between computer numbers and numbers in the real world is that computer numbers have limited size and hence limited precision;
- it’s possible to calculate a number too big or too small to be represented in a computer word.
- Programmers must remember these limits and write programs accordingly.

# Subword Parallellism

- Many graphics systems uses 8 bits to represent each of the three primary colors.
- Audio samples are often represented with 16 bits.
- Architects recognized that many graphics and audio applications would perform the same operation on vectors of these data.
- Thus, graphics and audio applications can take advantage of performing simultaneous operations on short vectors.
- By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on shorter vectors:
  - Sixteen 8-bit adds
  - Eight 16-bit adds
  - Four 32-bit adds
- **Subword Parallelism** is also called **data-level parallelism**, **vector parallelism**, or Single Instruction, Multiple Data (**SIMD**).

# ARMv8 SIMD

- ARMv8 added **32 128-bit registers** (V0, V1, ..., V31) and more than **500 machine-language instructions** to support subword parallelism.
- It supports all the subword data types you can imagine:
  - 8-bit, 16-bit, 32-bit, 64-bit, and 128-bit signed and unsigned integers
  - 32-bit and 64-bit floating point numbers

- ARMv8 assembler uses different suffixes for the SIMD registers to represent different widths.
- The suffixes are **B** (byte) for 8-bit operands, **H** (half) for 16-bit operands, **S** (single) for 32-bit operands, **D** (double) for 64-bit operands, and **Q** (quad) for 128-bit operands.
- The programmer also specifies the **number of subword operations** for that data width with a number before the register name.
- *Examples:*
  - 16 8-bit integer adds:

```
ADD V1.16B, V2.16B, V3.16B
```

  - 4 32-bit FP adds:

```
FADD V1.4S, V2.4S, V3.4S
```

# SIMD example on x86: DGEMM

```c
void dgemm (size_t n, double* A, double* B, double* C)
{
   for (size_t i = 0; i < n; ++i)
      for (size_t j = 0; j < n; ++j)
      {
         double cij = C[i+j*n]; /* cij = C[i][j] */
         for(size_t k = 0; k < n; k++ )
            cij += A[i+k*n] * B[k+j*n]; /*cij+=A[i][k]*B[k][j]*/
         C[i+j*n] = cij; /* C[i][j] = cij */
      }
}
```

- Notice that in reality it computes $C^T = B^T * A^T$

# SIMD example on x86: DGEMM

```
//include <x86intrin.h>
void dgemm (size_t n, double* A, double* B, double* C)
{
  for ( size_t i = 0; i < n; i+=4 )
    for ( size_t j = 0; j < n; j++ ) {
       __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j] */
       for( size_t k = 0; k < n; k++ )
          c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
                 _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
                 _mm256_broadcast_sd(B+k+j*n)));
       _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
       }
}
```

- The Advanced Vector Extensions (AVX) version is 3.85 times as fast the unoptimized code on one core of a 2.6 GHz Intel Core i7.

# ARMv8 SIMD

| Type | Description | Name | Size (bits) | | | | | FP Precision | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 8 | 16 | 32 | 64 | 128 | SP | DP |
| Add/ Subtract | Integer add | ADD | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | FP add | FADD | | | | | | ✓ | ✓ |
| | Integer subtract | SUB | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | FP subtract | FSUB | | | | | | ✓ | ✓ |
| Multiply | Unsigned integer multiply | UMUL | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Signed integer multiply | SMUL | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | FP multiply | FMUL | | | | | | ✓ | ✓ |
| Compare | Integer compare equal | CMEQ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | FP compare equal | FCMEQ | | | | | | ✓ | ✓ |
| Min/Max | Unsigned integer minmum | UMIN | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Signed integer minmum | SMIN | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | FP minmum | FMIN | | | | | | ✓ | ✓ |
| | Unsigned integer maximum | UMAX | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Signed integer maximum | SMAX | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | FP maximum | FMAX | | | | | | ✓ | ✓ |
| Shift | Integer shift left | SHL | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Unsigned integer shift right | USHR | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Signed integer shift right | SSHR | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| Logical | Bitwise AND | AND | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Bitwise OR | ORR | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| | Bitwise exclusive OR | EOR | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| Data Transfer | Load register | LDR | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | Store register | STR | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

# Full ARMv8 Integer and Floating-point Arithmetic Instructions

| Type | Mnemonic | Instruction |
|---|---|---|
| Integer Multiply & Divide | *MUL* | Multiply |
| | SMULH | Signed multiply high |
| | UMULH | Unsigned multiply high |
| | SDIV | Signed divide |
| | UDIV | Unsigned divide |
| | *SMULL* | Signed multiply long |
| | *UMULL* | Unsigned multiply long |
| | *MNEG* | Multiply-negate |
| | *UMNEGL* | Unsigned multiply-negate long |
| | *SMNEGL* | Signed multiply-negate long |

| Type | Mnemonic | Instruction |
|---|---|---|
| FP two source operands | FADDS | Floating-point add single |
| | FSUBS | Floating-point subtract single |
| | FMULS | Floating-point multiply single |
| | FDIVS | Floating-point divide single |
| | FADDD | Floating-point add double |
| | FSUBD | Floating-point subtract double |
| | FNMUL | Floating-point scalar multiply-negate |
| | FMULD | Floating-point multiply double |
| | FDIVD | Floating-point divide double |
| | FCMPS | Floating-point compare single (quiet) |
| | FCMPD | Floating-point compare double (quiet) |
| | FCMPE | Floating-point signaling compare |
| | FCCMP | Floating-point conditional quiet compare |
| | FCCMPE | Floating-point conditional signaling compare |

# Full ARMv8 Integer and Floating-point Arithmetic Instructions

| Type | Mnemonic | Instruction |
|---|---|---|
| FP one operand | FABS | Floating-point scalar absolute value |
| | FNEG | Floating-point scalar negate |
| | FSQRT | Floating-point scalar square root |
| FP Min/Max | FMAX | Floating-point scalar maximum |
| | FMIN | Floating-point scalar minimum |
| | FMAXNM | Floating-point scalar maximum number (NaN = –Inf) |
| | FMINNM | Floating-point scalar minimum number (NaN = +Inf) |

| Type | Mnemonic | Instruction |
|---|---|---|
| Integer Mul-Add | MADD | Multiply-add |
| | MSUB | Multiply-subtract |
| | SMADDL | Signed multiply-add long |
| | SMSUBL | Signed multiply-subtract long |
| | UMADDL | Unsigned multiply-add long |
| | UMSUBL | Unsigned multiply-subtract long |
| FP Mul-Add | FMADD | Floating-point fused multiply-add |
| | FMSUB | Floating-point fused multiply-subtract |
| | FNMADD | Floating-point negated fused multiply-add |
| | FNMSUB | Floating-point negated fused multiply-subtract |
| FP move | FMOV | Floating-point move to/from integer or FP register |
| | FMOVI | Floating-point move immediate |
| FP sel | FCSEL | Floating-point conditional select |

# Full ARMv8 Integer and Floating-point Arithmetic Instructions

| Type | Mnemonic | Instruction |
|---|---|---|
| FP round | FRINTA | Floating-point round to nearest with ties to odd |
| | FRINTI | Floating-point round using current rounding mode |
| | FRINTM | Floating-point round toward -infinity |
| | FRINTN | Floating-point round to nearest with ties to even |
| | FRINTP | Floating-point round toward +infinity |
| | FRINTX | Floating-pointl exact using current rounding mode |
| | FRINTZ | Floating-point round toward 0 |
| FP convert | FCVTAS | FP convert to signed integer, rounding to nearest odd |
| | FCVTAU | FP convert to unsigned integer, rounding to nearest odd |
| | FCVTMS | FP convert to signed integer, rounding toward -infinity |
| | FCVTMU | FP convert to unsigned integer, rounding toward -infinity |
| | FCVTNS | FP convert to signed integer, rounding to nearest even |
| | FCVTNU | FP convert to unsigned integer, rounding to nearest even |
| | FCVTPS | FP convert to signed integer, rounding toward +infinity |
| | FCVTPU | FP convert to unsigned integer, rounding toward +infinity |
| | FCVTZS | FP convert to signed integer, rounding toward 0 |
| | FCVTZU | FP convert to unsigned integer, rounding toward 0 |
| | SCVTF | Signed integer convert to FP, current rounding mode |
| | UCVTF | Unsigned integer convert to FP, current rounding mode |

# LEGv8 core instructions

| LEGv8 core instructions | Name | Format |
|---|---|---|
| add | ADD | R |
| subtract | SUB | R |
| add immediate | ADDI | I |
| subtract immediate | SUBI | I |
| add and set flags | ADDS | R |
| subtract and set flags | SUBS | R |
| add immediate and set flags | ADDIS | I |
| subtract immediate and set flags | SUBIS | I |
| load register | LDUR | D |
| store register | STUR | D |
| load signed word | LDURSW | D |
| store word | STURW | D |
| load half | LDURH | D |
| store half | STURH | D |
| load byte | LDURB | D |
| store byte | STURB | D |
| load exclusive register | LDXR | D |
| store exclusive register | STXR | D |
| move wide with zero | MOVZ | IM |

| LEGv8 core instructions | Name | Format |
|---|---|---|
| move wide with keep | MOVK | IM |
| and | AND | R |
| inclusive or | ORR | R |
| exclusive or | EOR | R |
| and immediate | ANDI | I |
| inclusive or immediate | ORRI | I |
| exclusive or immediate | EORI | I |
| logical shift left | LSL | R |
| logical shift right | LSR | R |
| compare and branch on equal 0 | CBZ | CB |
| compare and branch on not equal 0 | CBNZ | CB |
| branch conditionally | B.cond | CB |
| branch | B | B |
| branch to register | BR | R |
| branch with link | BL | B |

| LEGv8 arithmetic core | Name | Format |
|---|---|---|
| multiply | MUL | R |
| signed multiply high | SMULH | R |
| unsigned multiply high | UMULH | R |
| signed divide | SDIV | R |
| unsigned divide | UDIV | R |
| floating-point add single | FADDS | R |
| floating-point subtract single | FSUBS | R |
| floating-point multiply single | FMULS | R |
| floating-point divide single | FDIVS | R |
| floating-point add double | FADDD | R |
| floating-point subtract double | FSUBD | R |
| floating-point multiply double | FMULD | R |
| floating-point divide double | FDIVD | R |
| floating-point compare single | FCMPS | R |
| floating-point compare double | FCMPD | R |
| load single floating-point | LDURS | D |
| load double floating-point | LDURD | D |
| store single floating-point | STURS | D |
| store double floating-point | STURD | D |

SPEC CPU2006 integer and floating point →

| Instruction subset | Integer | Fl. pt. |
|---|---|---|
| LEGv8 core | 98% | 31% |
| LEGv8 arithmetic core | 2% | 66% |
| Remaining ARMv8 | 0% | 3% |

# Fallacies and Pitfalls

*Fallacy: Just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2.*

- Right shift divides by $2^i$ only for unsigned integers
- For signed integers, e.g., –5 / 4
  - With logic shift:
    - $11111011_2$ >>> 2 = $00111110_2$ = +62
  - Arithmetic right shift: replicate the sign bit
    - $11111011_2$ >> 2 = $11111110_2$ = –2

# Fallacies and Pitfalls

*Pitfall: Floating-point addition is not associative.*

| | | (x+y)+z | x+(y+z) |
|---|---|---|---|
| x | -1.50E+38 | | -1.50E+38 |
| y | 1.50E+38 | 0.00E+00 | |
| z | 1.0 | 1.0 | 1.50E+38 |
| | | 1.00E+00 | 0.00E+00 |

*Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.*

- Parallel programs may interleave operations in unexpected orders
- Assumptions of associativity may fail
- Need to validate parallel programs under varying degrees of parallelism
- Programmers who write parallel code with floating-point numbers need to verify whether the results are credible, even if they don't give the exact same answer as the sequential code.

# References


- David A. Patterson and John L. Hennessy, “Computer organization and design ARM edition: the hardware software interface,” Morgan Kaufmann, 2016.
- Chapter (3.2, 3.3, 3.4 solo LEGv8), (3.5: formato floating point e LEGv8), 3.6, 3.8, 3.9, 3.10


Most of the text has been taken and adapted from “Computer Organization and Design ARM Edition: The Hardware Software Interface”.
If not differently indicated, all figures have been taken from the book or the material in the companion website of “Computer Organization and Design ARM Edition: The Hardware Software Interface”.