H 3
Computer Arithmetic

ELECTA01 Computer architecture

ELECTA01
Hogeschool Rotterdam
Opleiding Elektrotechniek
Minor Embedded Systems
J.W. Peltenburg / J.Z.M. Broeders (brojz@hr.nl)

ARITHMETIC FOR COMPUTERS

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Addition and Subtraction

>>

- Right to left addition, bit by bit.
- Carries passed to the next digit to the left.
- Behavior as expected, because we humans do the same.
- Subtraction is the same as addition: the appropriate operand is simply negated before being added.

Integer Addition

- Example: 7 + 6

		(0)		(0)		(1)		(1)		(0)		(Carries)
. . .		0		0		0		1		1		1
. . .		0		0		0		1		1		0
. . .	(0)	0	(0)	0	(0)	1	(1)	1	(1)	0	(0)	1

- Overflow if result out of range
  - Adding +ve and –ve operands, no overflow
  - Adding two +ve operands
    - Overflow if result sign is 1
  - Adding two –ve operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

+7:	0000 0000 … 0000 0111
–6:	1111 1111 … 1111 1010
+1:	0000 0000 … 0000 0001

- Overflow if result out of range
  - Subtracting two +ve or two –ve operands, no overflow
  - Subtracting +ve from –ve operand
    - Overflow if result sign is 0
  - Subtracting –ve from +ve operand
    - Overflow if result sign is 1

Overflows

- Can we have an overflow by adding if both the operands have different signs?

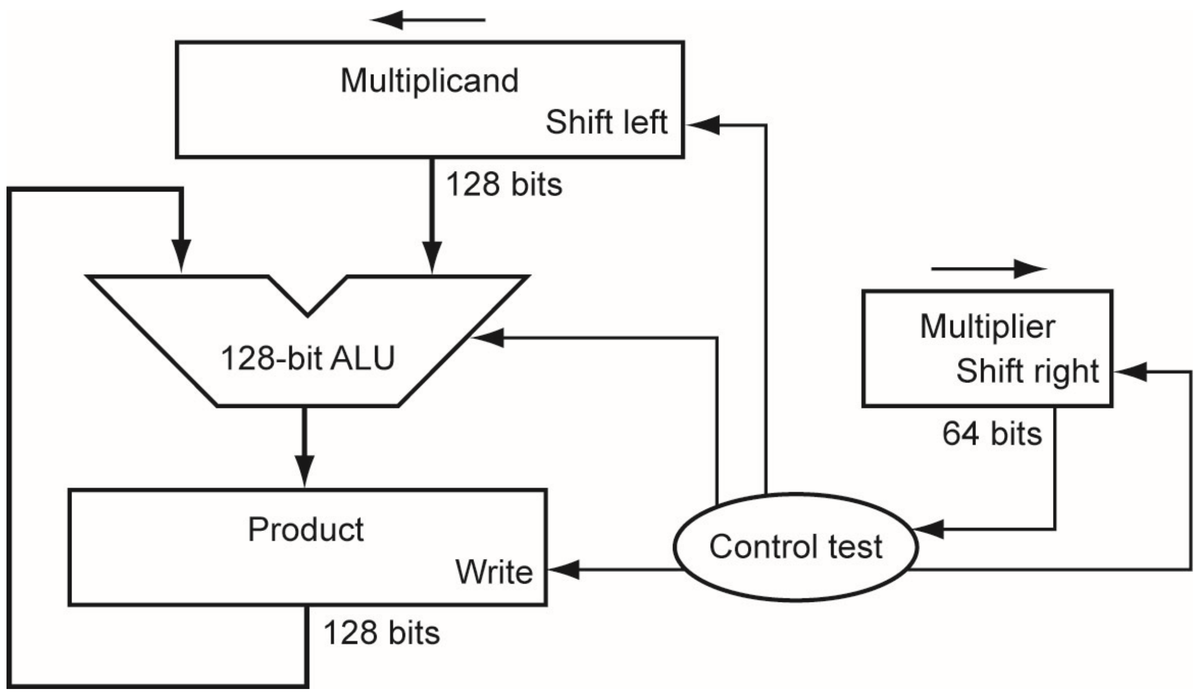
Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A – B	≥ 0	< 0	< 0
A – B	< 0	≥ 0	≥ 0

Multiplication

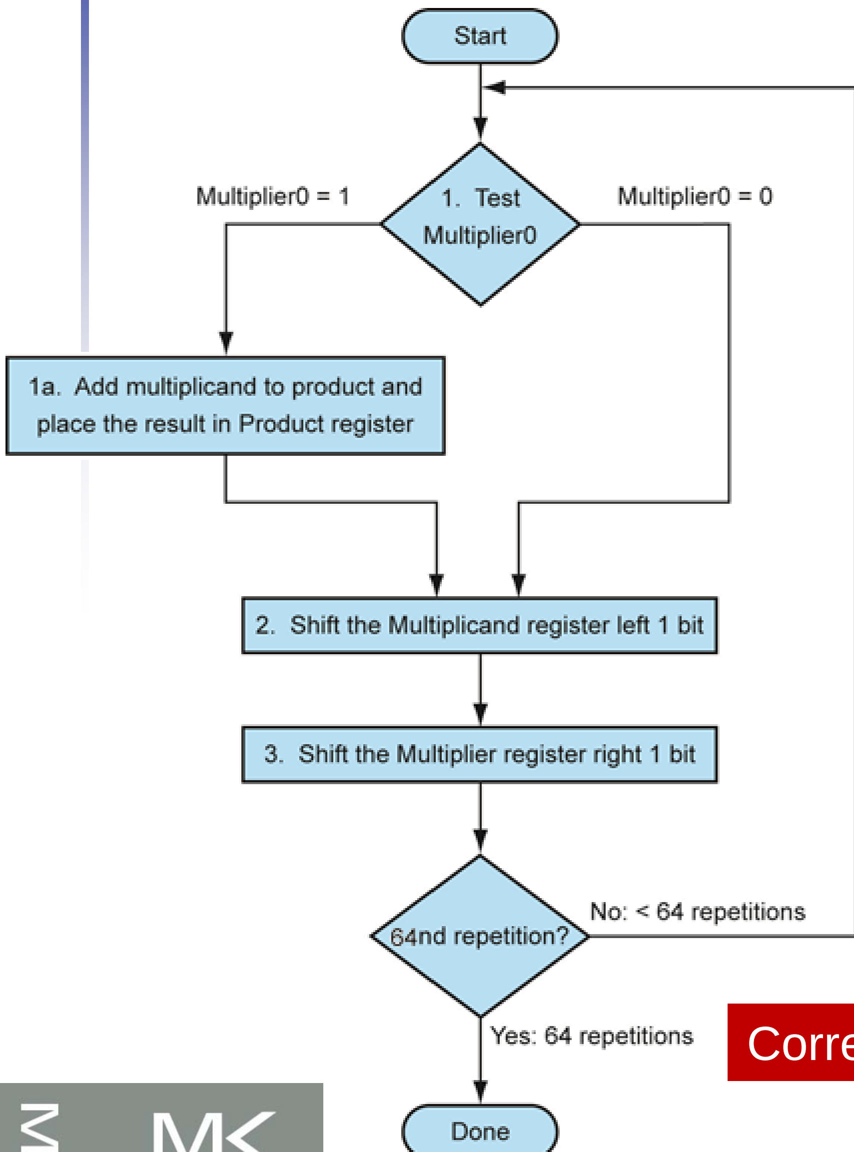
- Start with long-multiplication approach

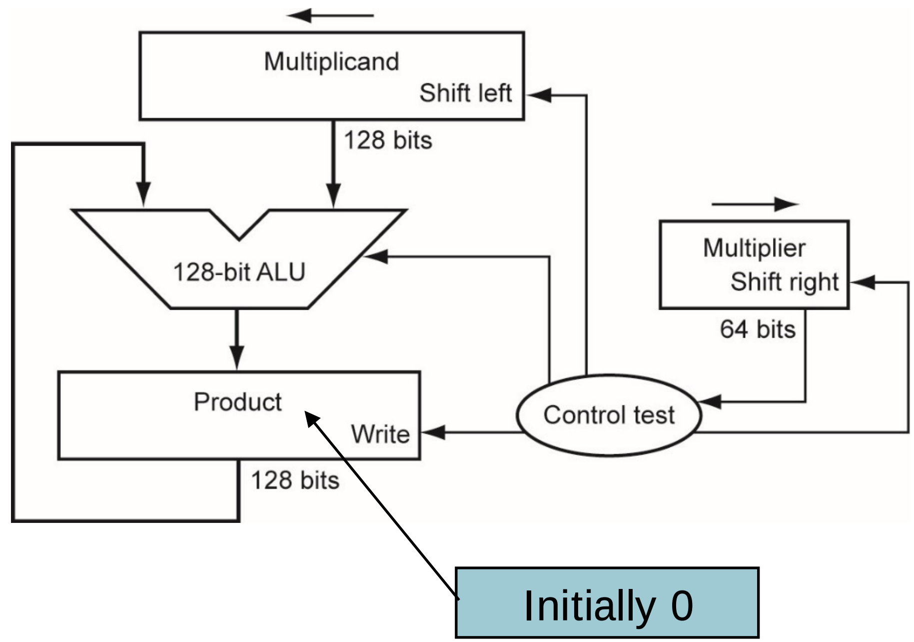
```
multiplicand        1000
multiplier     ×    1001
               ---------
                    1000
                   0000
                  0000
                 1000
               ---------
product          1001000
```

Length of product is the sum of operand lengths



Multiplication Hardware





Corrected version of Figure 3.4 page 194.

Exercise

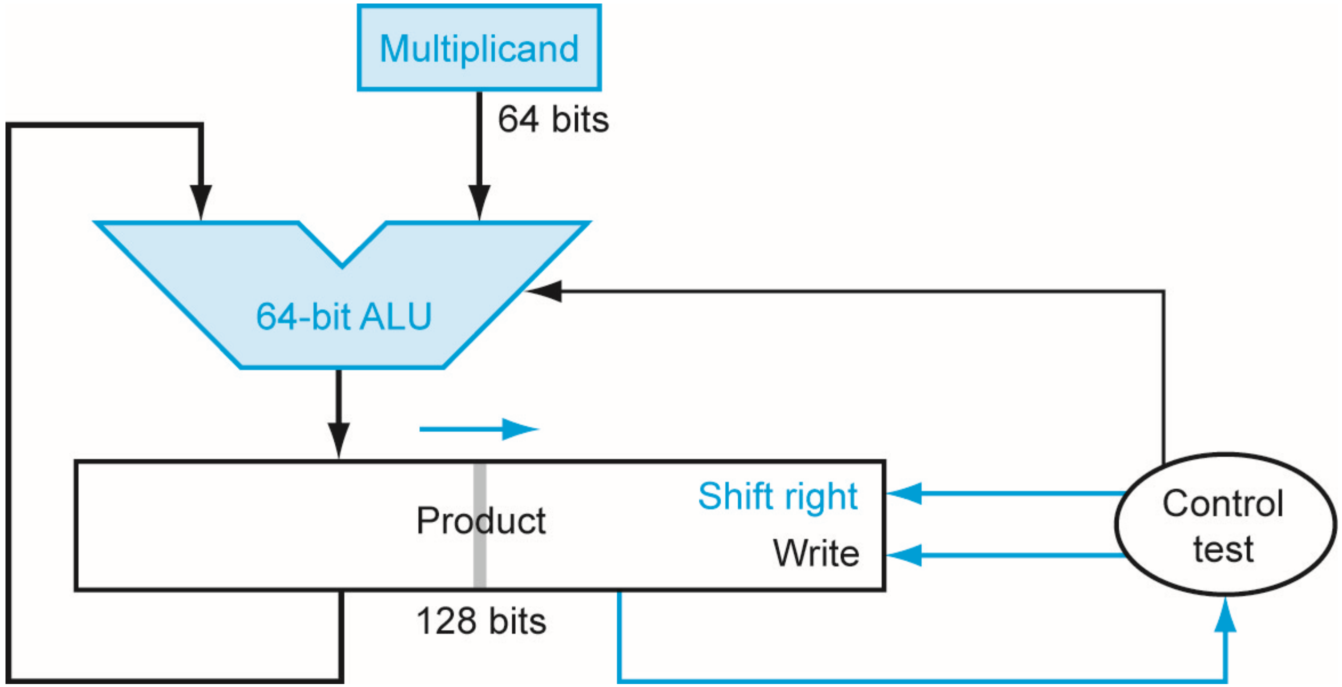
- Multiply 10 by 9
- Use 4 bits multiplicand and 4 bits multiplier (8 bits product).

Exercise solution

Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	1001	0000 1010	0000 0000
1	multiplier(0)=1; add multiplicand sll multiplicand slr multiplier	1001 1001 0100	0000 1010 0001 0100 0001 0100	0000 1010 0000 1010 0000 1010
2	multiplier(0)=0; **no** add multiplicand sll multiplicand slr multiplier	0100 0100 0010	0001 0100 0010 1000 0010 1000	0000 1010 0000 1010 0000 1010
3	multiplier(0)=0; **no** add multiplicand sll multiplicand slr multiplier	0010 0010 0001	0010 1000 0101 0000 0101 0000	0000 1010 0000 1010 0000 1010
4	multiplier(0)=1; add multiplicand sll multiplicand slr multiplier	0001 0001 0000	0101 0000 1010 0000 1010 0000	0101 1010 0101 1010 0101 1010

Optimized Multiplier

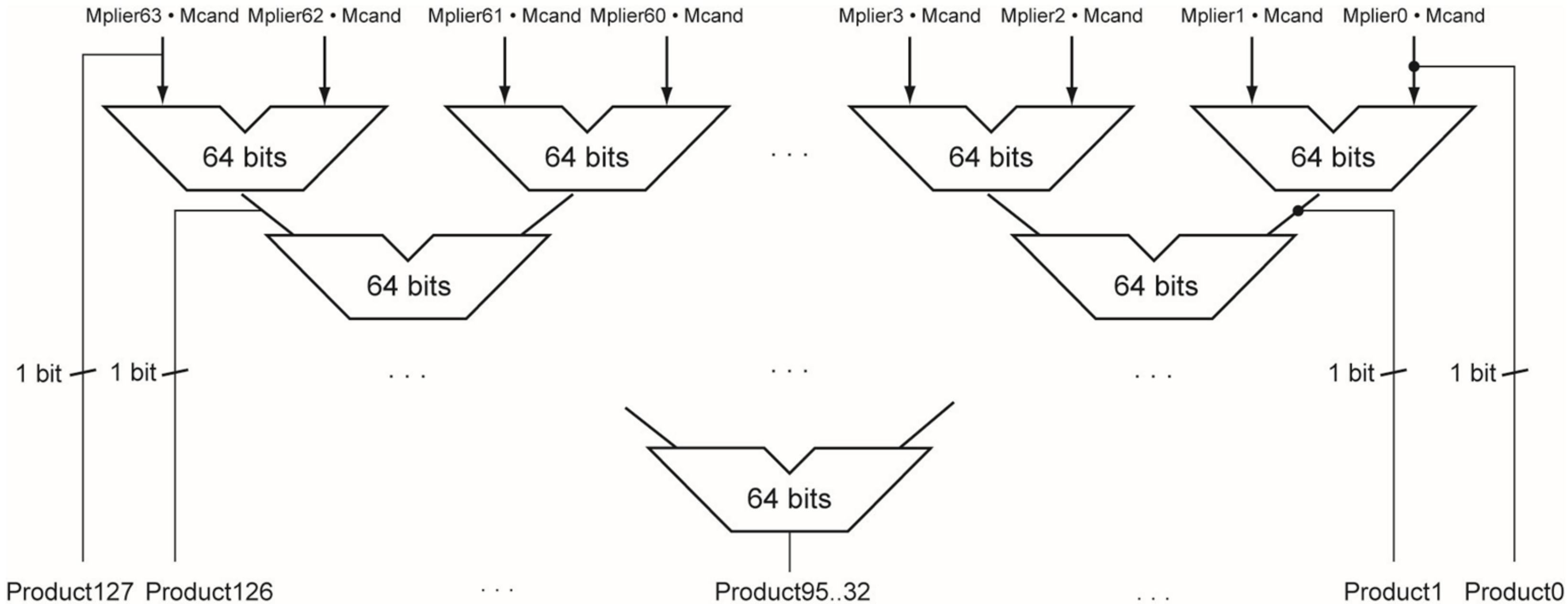
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff



- Can be pipelined
  - Several multiplication performed in parallel

LEGv8 Multiplication

- Three multiply instructions:
  - MUL: multiply
    - Gives the lower 64 bits of the product
  - SMULH: signed multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - UMULH: unsigned multiply high
    - Gives the upper 64 bits of the product, assuming the operands are unsigned

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
  - latest version IEEE 754-2008
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Several representations, most importantly:
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize 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 = 1023

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 $\approx$ 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
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Implicit 1 is significant
  - Single: approx 2^{-24}
    - Equivalent to $24 \times \log_{10}2 \approx 24 \times 0.3 \approx$ 7 decimal digits of precision
  - Double: approx 2^{-53}
    - Equivalent to $53 \times \log_{10}2 \approx 53 \times 0.3 \approx$ 16 decimal digits of precision

Floating-Point Example

- Represent –0.75 in IEEE 754 single and double precision
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00 `0xBF400000`
- Double: 1011111111101000…00
  - `0xBFE8000000000000`

Floating-Point Example

- What number is represented by the single precision float (IEEE 754) `0xC0A00000`

  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1.01_2) \times 2^{(129-127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal 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!

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

FP Instructions in LEGv8

- Separate FP registers
  - 32 x 32-bits single-precision: S0, …, S31
  - 32 x 64-bits double-precision: D0, …, D31
  - Sn stored in the lower 32 bits of Dn
- 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`

FP Instructions in LEGv8

- Single-precision arithmetic
  - `FADDS`, `FSUBS`, `FMULS`, `FDIVS`
    - e.g., `FADDS S2, S4, S6`
- Double-precision arithmetic
  - `FADDD`, `FSUBD`, `FMULD`, `FDIVD`
    - e.g., `FADDD D2, D4, D6`
- Single- and double-precision comparison
  - `FCMPS, FCMPD`
  - Sets or clears FP condition-code bits
- Branch on FP condition code true or false
  - `B.cond`

FP Example: °F to °C

- C code:

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

  - `fahr` in S0, result in S0, literals in global memory space with offset from X27 (const5, const9, const32).

- Compiled LEGv8 code:

```
f2c:
  LDURS S16, [X27, const5]
  LDURS S17, [X27, const9]
  FDIVS S18, S16, S17
  LDURS S19, [X27, const32]
  FSUBS S20, S0, S19
  FMULS S0,  S18, S20
  BR    LR
```

ARM Call Convention:
D0-D7: Arguments / Results
D8-D15: Saved by callee
D16-D31: Temporaries

FP Example: Matrix Multiplication

- X = X + Y × Z
  - All 32 × 32 matrices, 64-bit double-precision elements
- C code:

```
void mm(double x[][],
        double y[][], double z[][]) {
  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)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in X0, X1, X2, and i, j, k in X9, X10, X11

FP Example: Matrix Multiplication

- LEGv8 code:

```
mm:   ADDI  X12, XZR, #32
      MOV   X9, XZR
L1:   MOV   X10, XZR
L2:   MOV   X11, XZR
      LSL   X13, X9, #5
      ADD   X14, X13, X10
      LSL   X15, X14, #3
      ADD   X16, X0, X15
      LDURD D8, [X16, #0]
L3:   LSL   X13, X9, #5
      ADD   X14, X13, X11
      LSL   X15, X14, #3
      ADD   X17, X1, X15
      LDURD D9, [X17, #0]
      LSL   X13, X11, #5
      ADD   X14, X13, X11
      LSL   X15, X14, #3
      ADD   X17, X2, X15
      LDURD D10, [X17, #0]
      FMULD D11, D9, D10
      FADDD D8, D8, D11
      ADDI  X11, X11, #1
      CMP   X11, X12
      B.LT  L3
      STURD D8, [X16, #0]
      ADDI  X10, X10, #1
      CMP   X10, X12
      B.LT  L2
      ADDI  X9, X9, #1
      CMP   X9, X12
      B.LT  L1
      BR    LR
```

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow