International Institute of Information Technology

A Project Report
on

“Implementation Posit FPU on FPGA”.

Under Guidance
of
Prof. Subir K Roy

Submitted by

Apoorva J R & Shalini Singh
MS2019003 MS2019015

Index

The Posit Format

Here is the structure of an n-bit posit representation with *es* exponent bits:

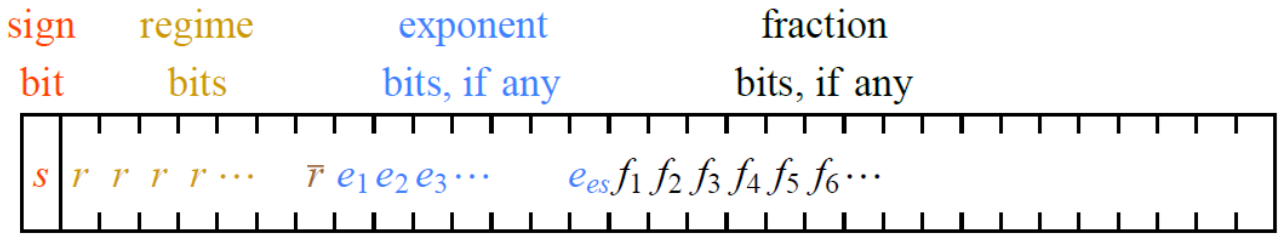
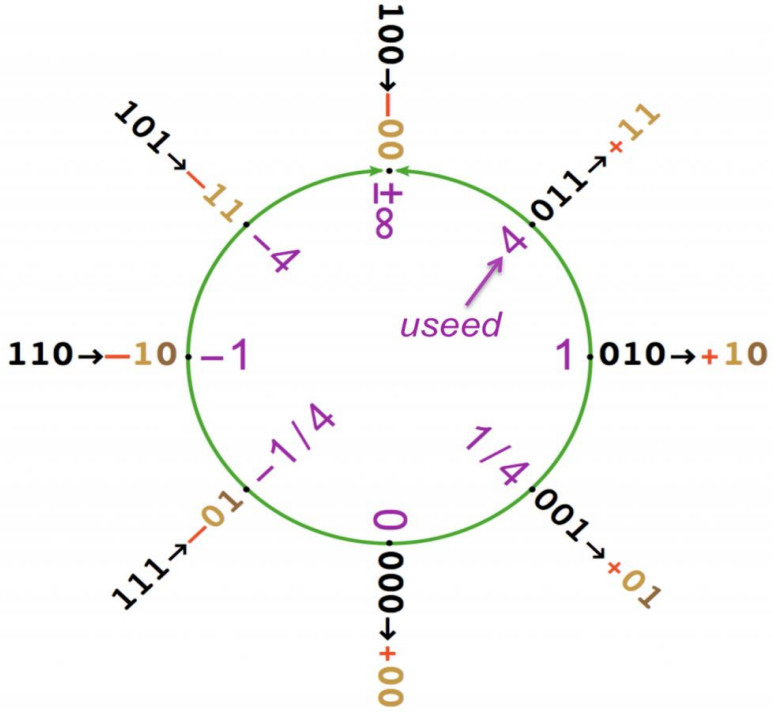


Figure 2. Generic posit format for finite, nonzero values

Where, **s** is sign bit, ***r, r,*** …. are regime bits, $\bar{r}$ is regime terminating bit, es is number of exponent bits, ***e1, e2,*** …. are exponent bits and ***f1, f2,*** …. are fraction bits.

Projective Reals



Posit precision increases by appending bits, and values remain where they are on the circle when a **0** bit is appended. Appending a **1** bit creates a new value between two posits on the circle. Let *maxpos* be the largest positive value and *minpos* be the smallest positive value on the ring defined with a bit string, *maxpos* is *useed* and *minpos* is *1/useed*.

As an example, fig. 4 shows a build up from a 3-bit to a 5-bit posit with $es = 2$, so $useed = 16$:

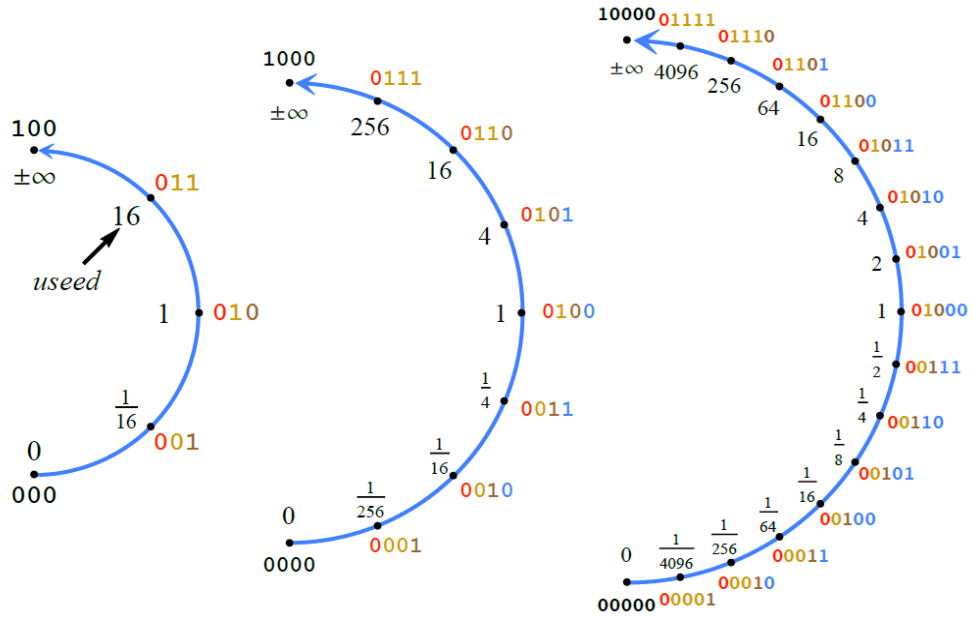


Figure 4. Posit construction with two exponent bits, $es = 2$, $useed = 2^{2^{es}} = 16$

PROBLEMS WITH IEEE 754

IEEE 754 is a prominent standard established in 1985 for representing real-valued numbers in a floating-point format. Despite all its benefits, this number system suffers from several weaknesses:

1. **Large size for small numbers**
2. **Limited precision:** IEEE 754 reserves several bits to represent NaNs, denormals, positive/negative zero and infinity.
3. **Exceptional bit representations**
4. **Breaking algebraic rules:**The expression (x+y)+z results in 1, where the floating-point values are x = 1e30, y = -1e30 and z = 1 is 1. Using the same values, x+(y+z) results in 0.
5. **Producing inconsistent results**
6. **Complex design and verification**

How Posit is Better Than IEEE 754?

- A posit number includes the exponent and fraction only if necessary.
- Tapered precision (small exponents are represented more accurately than the numbers with large magnitude exponents).
- Larger dynamic range and Holding Algebraic Rules Across Formats
- Exception Handling. While there are 14 representations of Nans in IEEE 754, there is no "NaN" in posits. Moreover, posit has single representations for 0 and ∞.
- less power use and high speed.
- Produces repetitive identical results.

Posit Representation

If the set of fraction bits is $\{f_1\ f_2\ldots..\ f_{fs}\}$, possibly the empty set, let f be the value represented by $1.f1\ f2\ldots f_{fs}$. Then p represents:

$$x = \begin{cases} 0, & p = 0, \\ \pm\infty, & p = -2^{n-1}, \\ sign(p) \times useed^k \times 2^e \times f, & \text{all other } p. \end{cases}$$

Example:

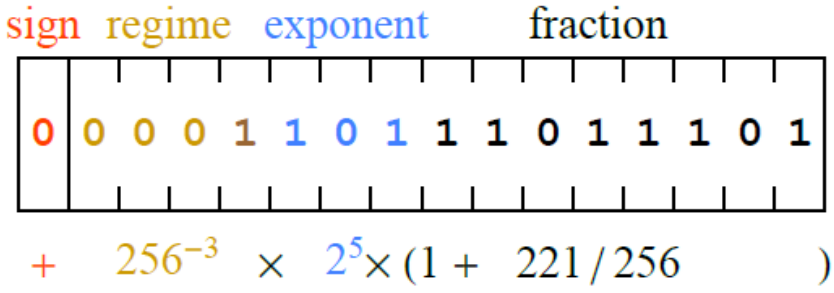
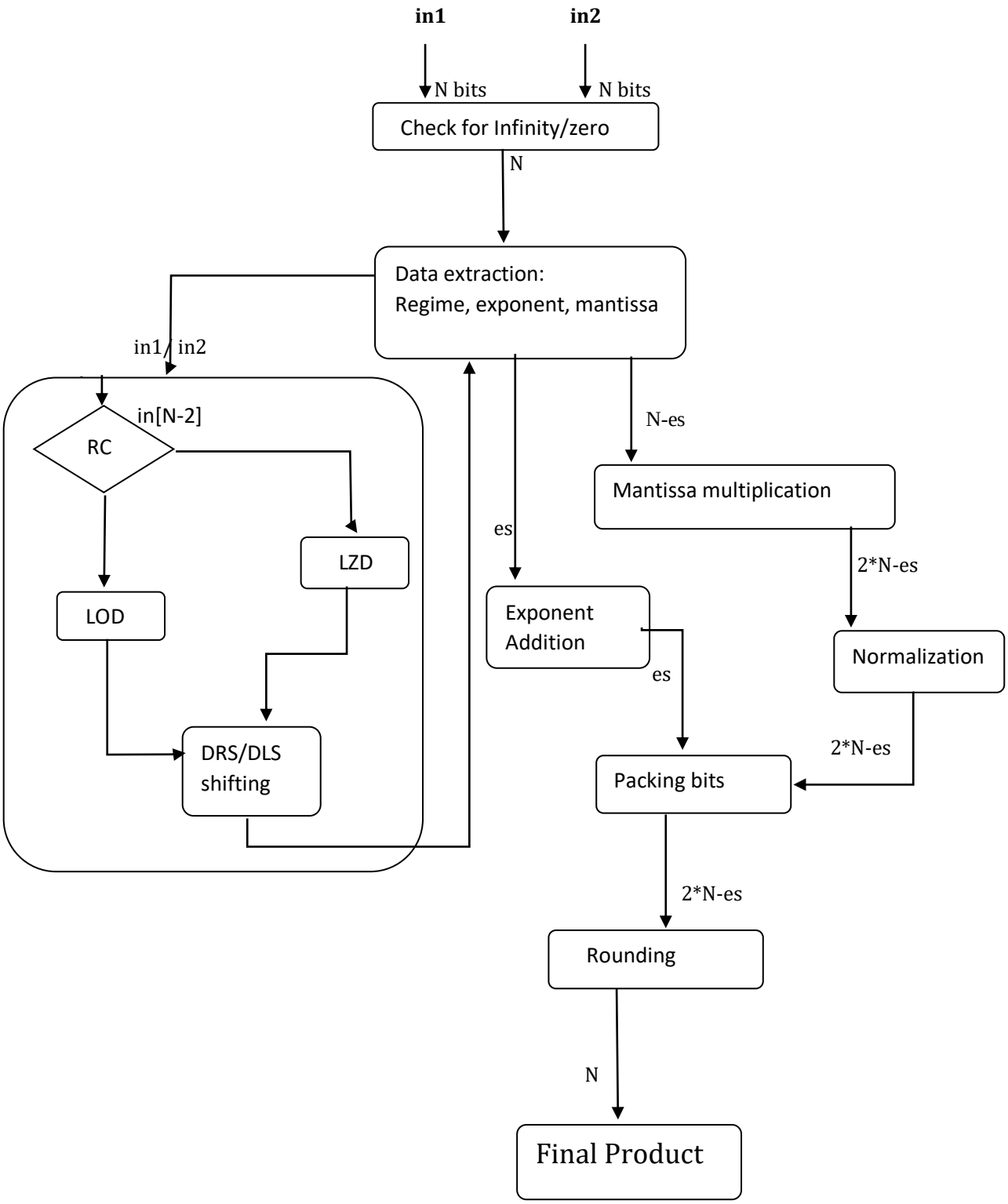


Figure 5. Example of a posit bit string and its mathematical meaning

Flow Chart of Posit multiplier



Algorithms:

(i) Posit_Adder

Algorithm 4 Proposed Posit Adder Computational Flow

GIVEN:
  N: Posit Word Size
  ES: Posit Exponent Field Size
  RS: $log_2\ (N)$ (Posit Regime Value Store Space Bit Size)
Input Operands: $IN1,\ IN2$
Posit Data Extraction → Effective Operand (XIN), Sign (S), Regime Check (RC), Regime (R), Exponent (E), Mantissa (M), Infinity (Inf), Zero (Z):Performed similar to data extraction of posit in Algorithm-3
  Extraction from IN1 → XIN1, S1, R1, E1, M1, Inf1, Z1
  Extraction from IN2 → XIN2, S2, R2, E2, M2, Inf2, Z2
  $Z \leftarrow Z1\&Z2 \qquad Inf \leftarrow Inf1|Inf2$
Core Adder Arithmetic Processing:
  Effective Operation: OP ← S1 xor S2
  Large Operand: IN1_gt_IN2 ←XIN1[N-2:0]>XIN2[N-2:0] ? 1 : 0
    Large (L) Component: LS, LRC, LR, LE, and LM
    Small (S) Component: SS, SRC, SR, SE, and SM
MANTISSA ADDITION:
Effective Exponent Difference (Ediff):
$Ediff \leftarrow ((LRC?LR-(SRC?SR:-SR):SR-LR)<<ES)+LE-SE$
$SM_T \leftarrow$ Dynamic Right Shift SM by Ediff
Add LM and SM_T:
  $Add_M \leftarrow OP\ ?\ LM+SM_T\ :\ LM-SM_T$
  Mantissa Overflowed: $Movf \leftarrow Add_M[MSB]$
  $Add_M \leftarrow Movf\ ?\ Add_M\ :\ Add_M << 1$
Normalization of Add_M:
  Nshift ← LOD of Add_M
  $Add_M \leftarrow Add_M << Nshift$, (Dynamic Left Shifting by Nshift)
Final EXPONENT (E_O) and REGIME (R_O) Computation:
  $Exp \leftarrow \{(LRC\ ?\ LR\ :\ -LR),LE\}+Movf-Nshift$
  E_O and R_O: Computed as Line 18-26 of Algorithm-1
Data Composition and Post-Processing:
  Regime, Exp & Mantissa Packing (REM): Computed similar to Line 27-29 of Algorithm-1 while using Exp, R_O, Add_M.
  Rounding: Round-to-zero (Truncation)
Final Output: Combine LS with LSB (N-1) bit of rounded REM, while considering Exceptions

(ii) Posit_multiplier

Algorithm 5 Proposed Posit Multiplier Computational Flow

GIVEN: N, ES, RS (Similar to the Adder Algorithm-4 definition)
Input Operands: $IN1$, $IN2$
Posit Data Extraction: → Similar to data extraction in Algorithm-3,4
  IN1 → XIN1, S1, R1, E1, M1, Inf1, Z1
  IN2 → XIN2, S2, R2, E2, M2, Inf2, Z2
  $Z \leftarrow Z1\&Z2$ $Inf \leftarrow Inf1|Inf2$
  $RG1 \leftarrow RC1\ ?\ R1\ :\ -R1$, $RG2 \leftarrow RC2\ ?\ R2\ :\ -R2$
Mantissa Multiplier Arithmetic Processing:
  $M \leftarrow M1 * M2$
  $Movf \leftarrow M[MSB]$
  $M \leftarrow Movf\ ?\ M\ :\ M << 1$
Final EXPONENT (E_O) and REGIME (R_O) Computation:
  $Exp \leftarrow \{RG1, E1\} + \{RG2, E2\} + Movf$
  E_O and R_O: Similar to Algorithm-1,4
Data Composition, Rounding and Final Output: Similar to Algorithm-1,4

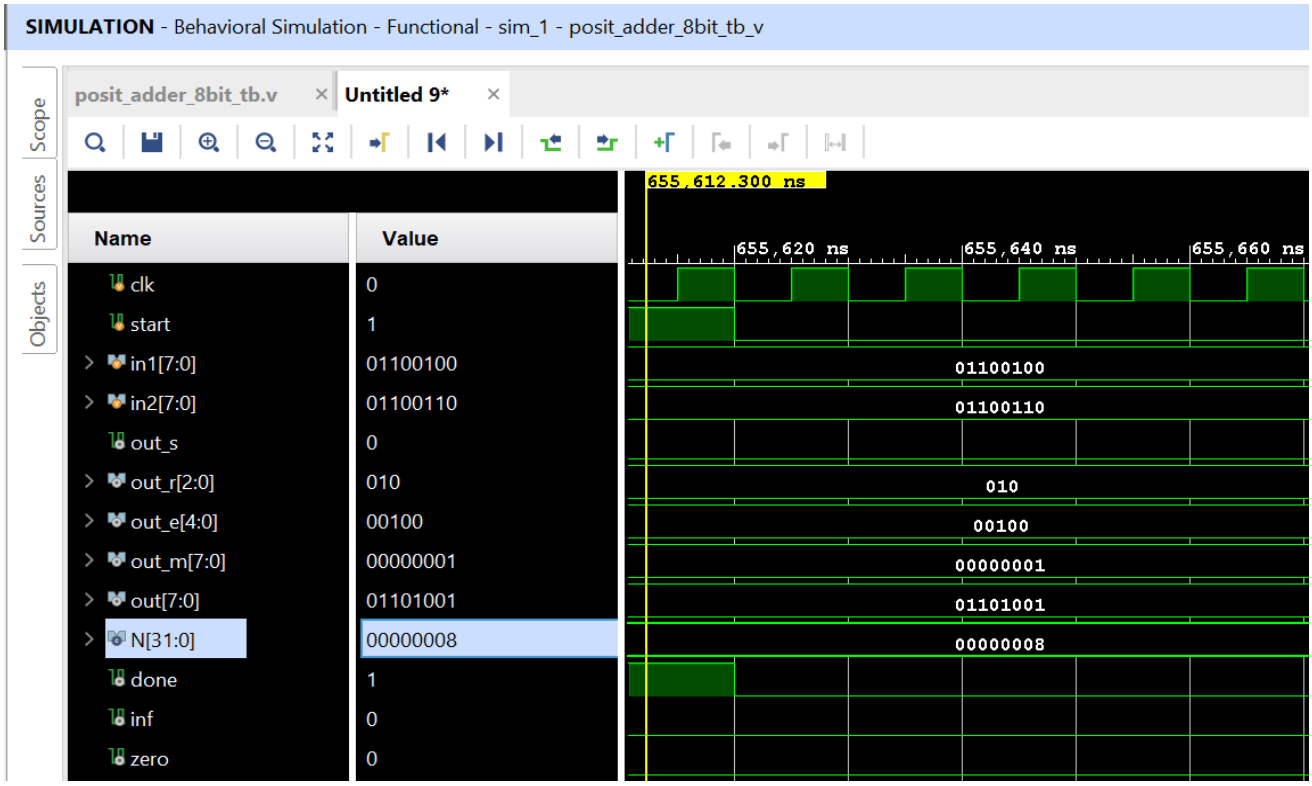
(iii) LOD/LZD and DLS

Algorithm 2 Parameterized Generation of LOD/LZD and DLS

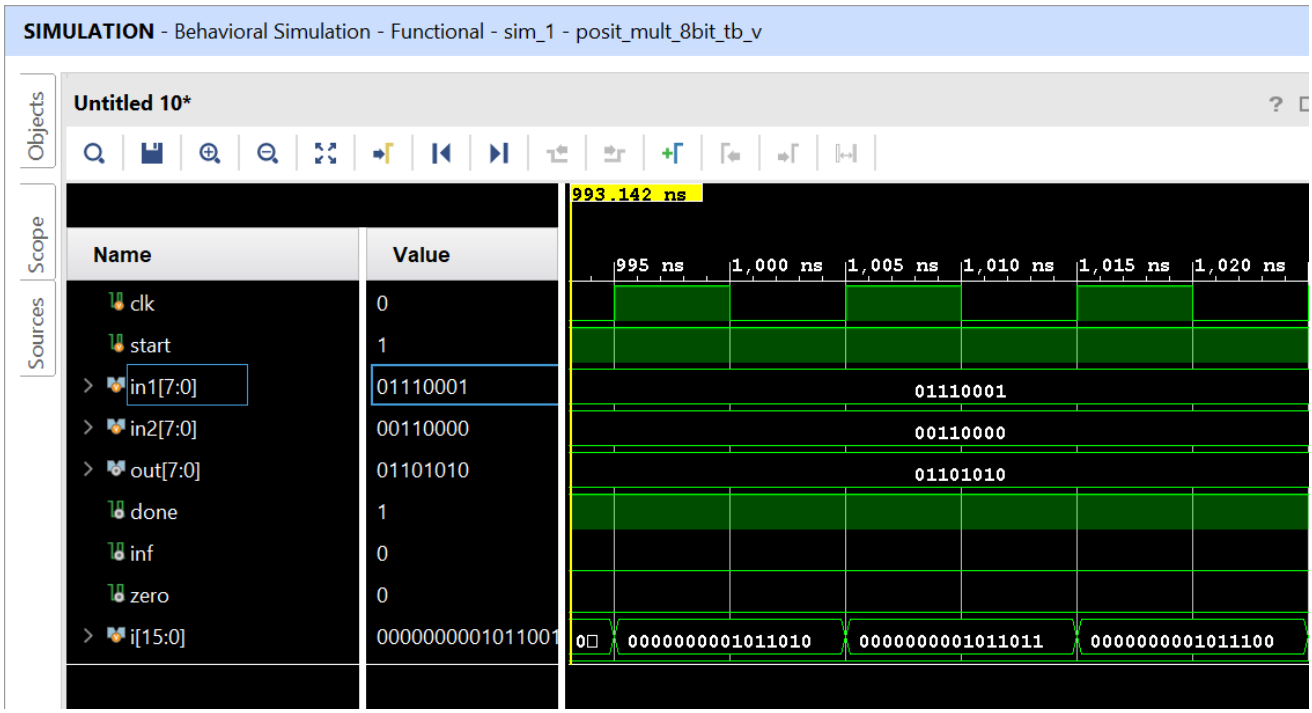
```
LOD/LZD #(N) (in[N-1:0], K[S-1:0], vld):
  N: Word Size,   S: Log2(N)
    GENERATE
      IF (N == 2)
        For LOD:   vld = |in,    K = (!in[1]) & in[0]
        For LZD:   vld = !(&in),   K = in[1] & (!in[0])
      ELSIF (N & (N-1))
        LOD/LZD #(1«S) (1«S 1'b0 | in, K, vld)
      ELSE
        K_L[S-2:0], K_H[S-2:0], vld_L, vld_H
        LOD/LZD #(N»1) (in[(N»1)-1:0], K_L, vld_L)
        LOD/LZD #(N»1) (in[N-1:N»1], K_H, vld_H)
        vld = vld_L | vld_L
        K = vld_H ? {1'b0,K_H} : {vld_L,K_L}
    ENDGENERATE

DLS #(N) (in[N-1:0], b[S-1:0], OUT):
  N: Word Size,   S: Log2(N),   TMP[S-1:0][N-1:0]
    TMP[0] = b[0] ? in « 1 : in;
    GENVAR i
    GENERATE
      for (i=1; i<S; i=i+1)
        TMP[i] = b[i] ? (TMP[i-1] « 2**i) : TMP[i-1]
      end
    ENDGENERATE
  OUT = TMP[S-1]
```

Simulation results

(i) 8-bit Posit Adder



(ii) 8-bit Posit Multiplier

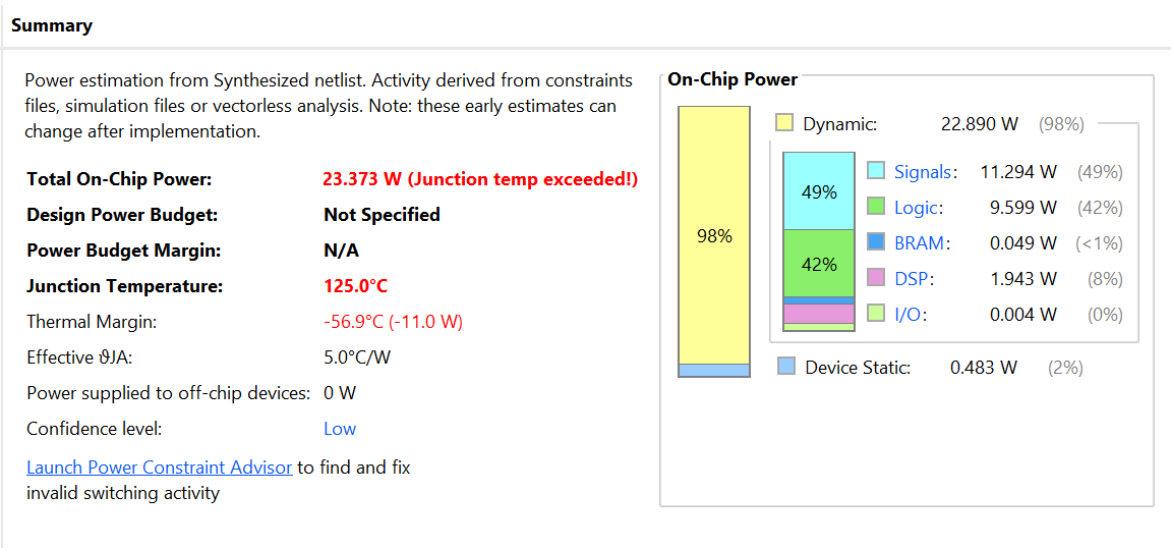


Xilinx Vivado Project Flow

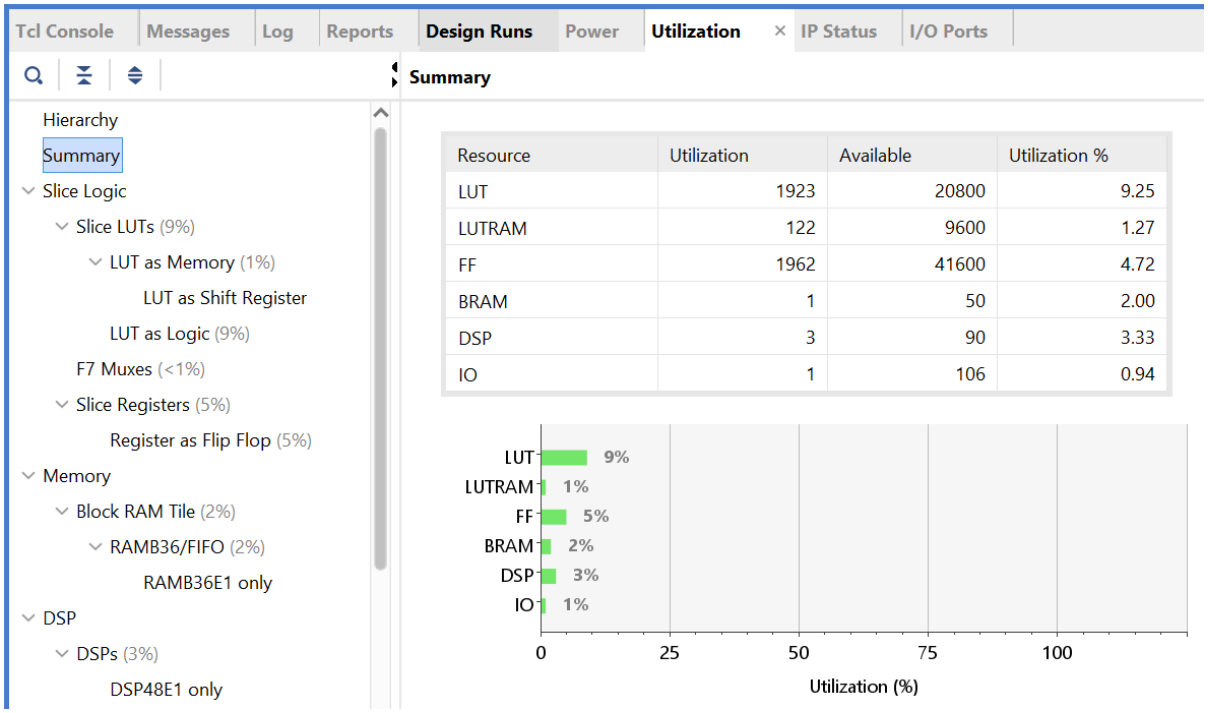
We followed the following steps on the Xilinx Synthesis Platform to carry out the synthesis and implementation of your multiplier design:

1. **Design and Simulate** using Testbench (posit_mult_tb.v)
2. **Verification** of the test cases by converting the posit value to decimal.
3. **RTL Analysis** where the schematic of the design is made.
4. **Synthesis** where power, timing, and utilization reports are generated
5. **Constraint** file: I/O ports constraints for adder and 8-bit multiplier designs. The inputs/outputs are directly given to the FPGA board pins.
6. **Implementation** where reports and schematic are generated based on constraints
8. **Bitstream generation** where the .bit file is generated to programming the hardware (file **main.bit** for 32 x 32bit multiplier)
9. **Hardware** implementation on Basys3 (using ILA and VIO for 32bit multiplier)
10. **Testing** for various inputs using ILA and VIO

Power Report



Design complexity and Resource utilization:



Resource	Utilization	Available	Utilization %
LUT	1923	20800	9.25
LUTRAM	122	9600	1.27
FF	1962	41600	4.72
BRAM	1	50	2.00
DSP	3	90	3.33
IO	1	106	0.94

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Block RAM Tile (50)	DSPs (90)	Bonded IOB (106)
main	1923	1962	3	1	3	1
dbg_hub (dbg_hub_CV)	0	0	0	0	0	0
I1 (ila_0)	792	1346	3	1	0	0
p1 (posit_mult)	850	0	0	0	3	0
vio1 (vio_0)	281	616	0	0	0	0

Observations / changes required in RTL design

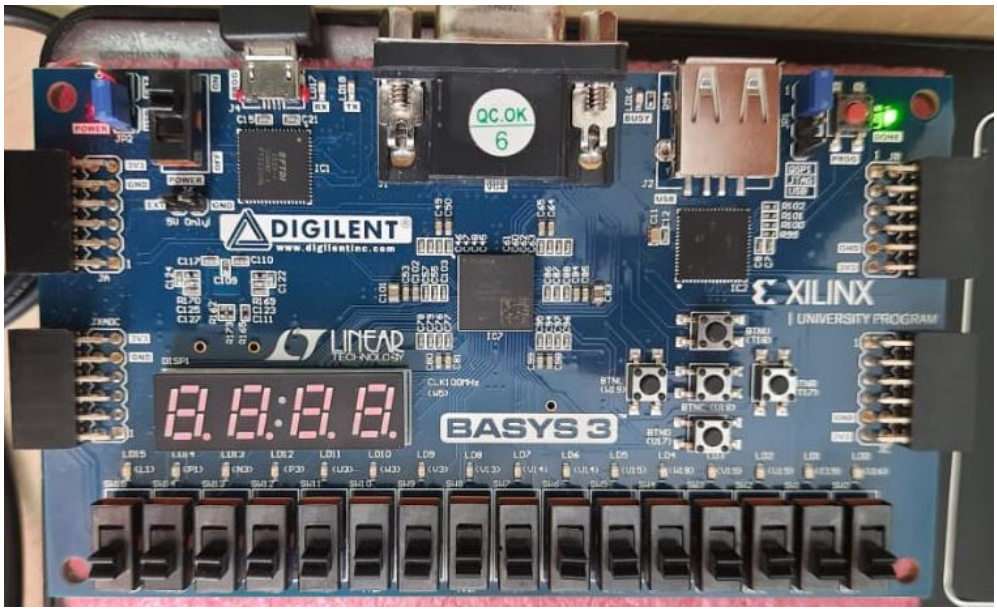
Altering the values of es (no. of bits for exponent value) according to the value of N (Posit size).

For instance, when N=8, and es=4, the output of the 8bit posit multiplier(posit_8mult.v) is far from true value. For test cases whose operands have huge difference among them, the output was either the larger number or very close to it. This is because large value of es gives very less or no room for mantissa bits. When we change es=2 for N=8, we get right results. It should be noted that N and es are proportional for any number.

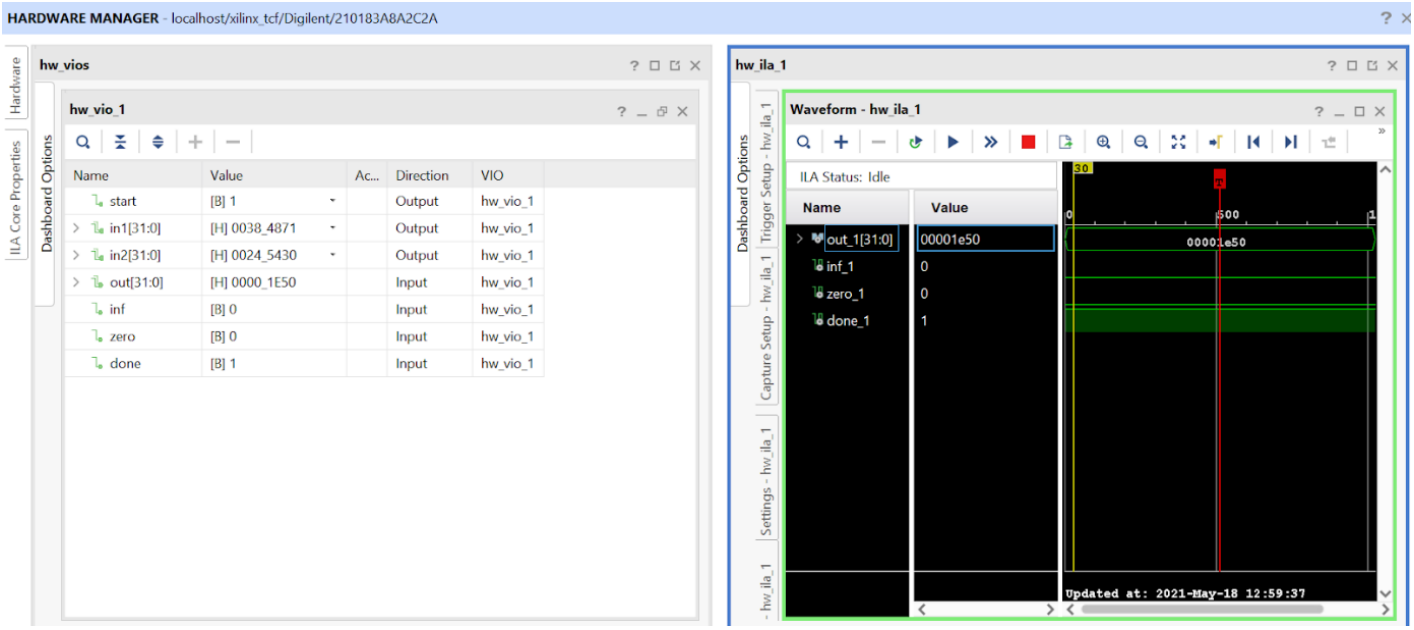
FPGA Implementation:

The 32 x 32-bit multiplier design is implemented on Basys3 FPGA board. The test inputs and outputs are seen on VIO (Virtual Input/Output) and output waveform is seen on ILA (Integrated Logic Analyzer).

Basys3 FPGA board programmed for 32x32bit multiplier:



VIO and ILA windows for Output



Result Validation:

For 32 x 32-bit multiplier: **N =32 & es = 8**

Operand 1**:** **in1** = 32'h00384871 = 1.422905485 **x 10^{15}**

in1 = 0000_0000_0011_1000_0100_1000_0111_0001

s1 =0 **k1** = -9 **ussed1** = $_2 2^8$ **e1** = $(11000010)_b$ = 194

m1 = $(1.010001110001)_b$ = 1.2637939453125

Operand 2**:** **in2** = 32'h00245430 = **1.256364029 x 10^{-33}**

in2 = 0000_0000_0010_0100_0101_0100_0011_0000

s2 = 0 **k2** = -9 **useed2** = $_2 2^8$ **e2** = $(00100010)_b$ = 34

m2 = $(1.1010000110000)_b$ = 1.630859375

ILA output **Out** = 32'h00001e50 = **1.734723476 x 10^{-18}**

Out = 0000_0000_0000_0000_0001_1110_0101_0000

s = 0 **k** = -18 **useed2** = $_2 2^8$ **e** = $(11100000)_b$ = 229 **m** = 0

in1 x in2 = **1.422905485 x 10^{15}** **x** **1.256364029 x 10^{-33} = 1.787687268 x 10^{-18}**

Challenges:

1. Limited resources.
2. Verifying test cases by manually converting the input operands and output to decimal values is time consuming.
3. Figuring out the relationship between N and es values. The multiplier doesn't give the right result for a very large value of es, as fewer bits remain for mantissa.
4. Configuration of VIO (Virtual Input/Output) and ILA (Integrated Logic Analyzer) for 32-bit posit multiplier. They were not necessary to implement the posit Adder and 8bit x 8bit posit multiplier as the number of pins allocated for input on Basys3 board were sufficient.

Conclusion

Posit number system is most efficient to represent real numbers for computers, and a better alternative to the standard IEEE floating point formats. We used this representation to build 8-bit Adder, 8-bit multiplier. Finally, we extended it to 32x32-bit multiplier and implemented it on FPGA Basys3 board. We configured VIO and ILA to observe the results for various test cases.

Bibliography

- Gustafson, John L. and Yonemoto, Isaac, "Beating Floating Point at its Own Game: Posit Arithmetic," pp. 1–16. [Online]. Available: http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf
- Universal Number Posit Arithmetic Generator on FPGA Manish Kumar Jaiswal, and Hayden K.-H So Dept. of EEE, The University of Hong Kong, Hong Kong.
- Posit: A Potential Replacement for IEEE 754