RF2P: A Lightweight RISC Processor Optimized for Rapid Migration from IEEE-754 to Posit

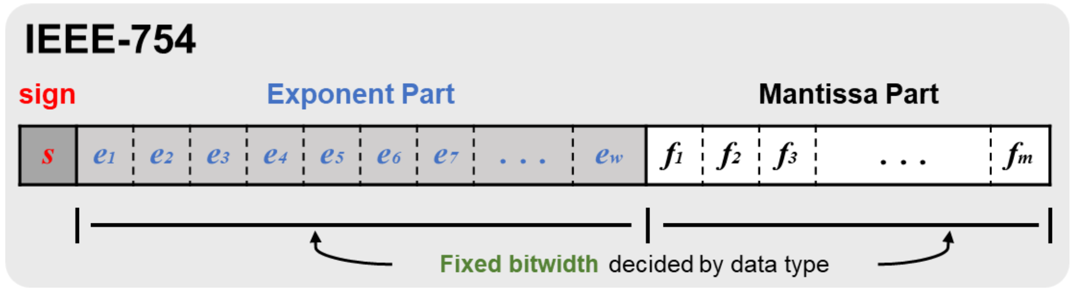
Hyun Woo Oh, Seongmo An, Won Sik Jeong, Seung Eun Lee

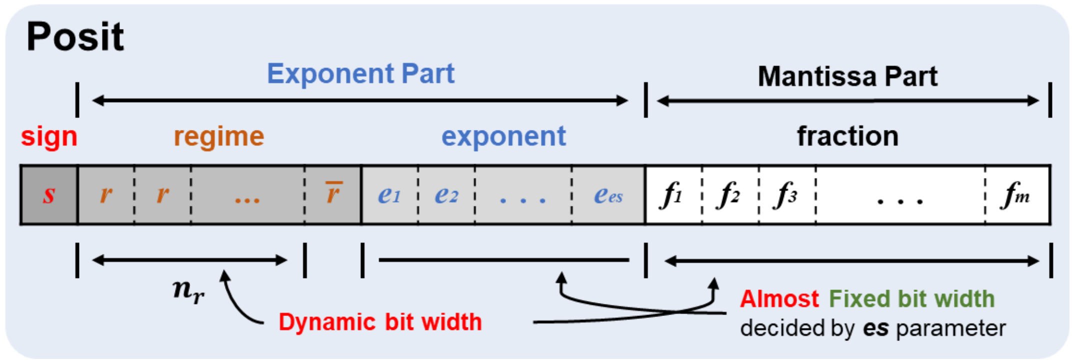
Seoul National University of Science and Technology

Seoul, Republic of Korea

Why posit?

- **Better dynamic range**
- **Higher precision for frequently-used numbers**


IEEE-754
sign
Exponent Part
Mantissa Part
s
e1 e2 e3 e4 e5 e6 e7 . . . ew
f1 f2 f3 . . . fm
Fixed bitwidth decided by data type

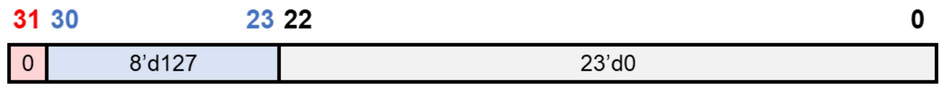


Why posit?

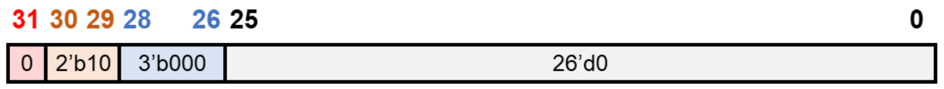
- **Examples**

Float32 | ***Posit(N=32, es=3)***

$$1 = (-1)^{0} \times 2^{0} \times 1.0000000\ldots_{(2)}$$



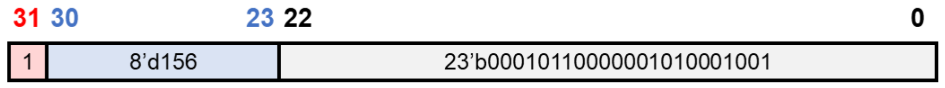
$exp = 127 - 127 = 0$



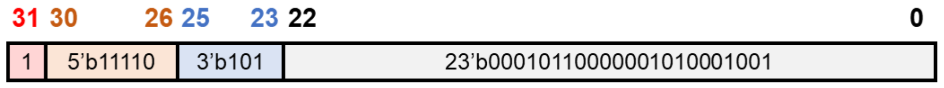
$n_r = 1$ $e = 0$

$exp = (1 - 1) \cdot 2^3 + 0 = 0$

$$-583{,}049{,}812 = (-1)^{1} \times 2^{29} \times 1.00010110000000101000100101 01\ldots_{(2)}$$



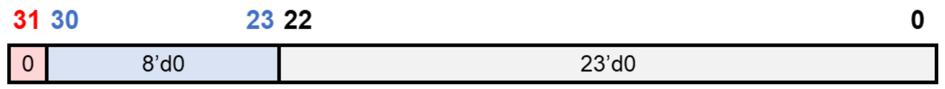
$exp = 156 - 127 = 29$



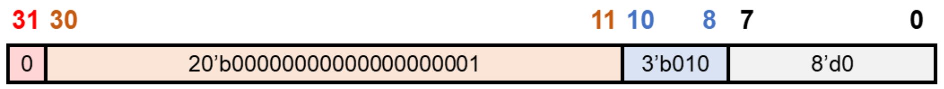
$n_r = 4$ $e = 5$

$exp = (4 - 1) \cdot 2^3 + 5 = 29$

$$2^{-150} = (-1)^{0} \times 2^{-150} \times 1.0000000\ldots_{(2)}$$



$exp = -\infty \rightarrow value = 0$



$n_r = 19$ $e = 2$

$exp = -19 \cdot 2^3 + 2 = -150$

Challenges in applying posit

- **Lack of HW acceleration**
  - Low throughput & energy efficiency
    - Extreme slow SW-only implementation
  - Requires more area for HW implementation

- **No compiler support**
  - Requires whole new code to migrate to posit

Related Works

- **PERCIVAL (TETC ’22)**
  - High throughput approximate solutions (MUL, DIV)
    - Innate errors (up to 11%)
  - Custom instruction set extension
    - Requires a whole new code
- **PERC: Posit Enhanced Rocket Chip (CARRV ’20)**
  - FP extension compatible
    - But no compiler support → Requires new code for variable initialization
    - Large PAU→ low operating frequency
- **PERI: Posit Enabled RISC-V Core (TACO ’21)**
  - FP extension compatible & Compiler support
  - Low throughput on intensive operations (DIV, SQRT)

Related Works

- **PERCIVAL (TETC ’22)**
  - High throughput approximate solutions (MUL, DIV)

1) Fixed PAU arch. → overheads appear by discrepancies (max frequency, area, …) → **Scalable architecture is required**

  - FP extension compatible

2) Integrated **solution for fast migration** is required. (compiler support, compatibility, testability)

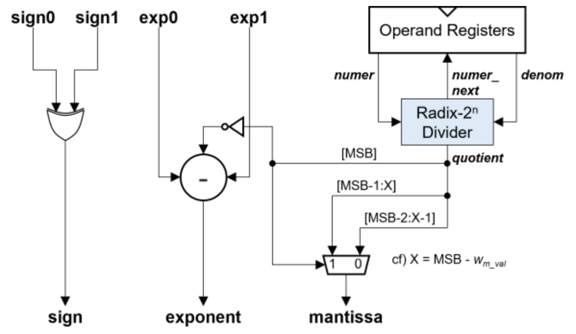
  - FP extension compatible & Compiler support
  - Low throughput on intensive operations (DIV, SQRT)

SEOULTECH

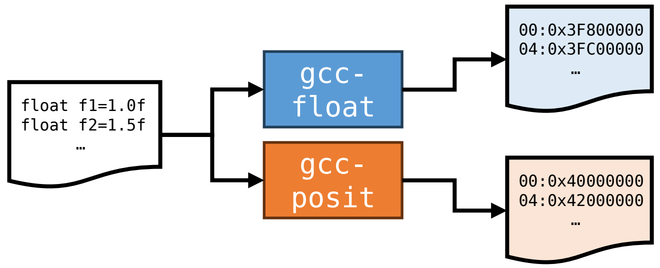
Computer Architecture Lab 컴퓨터 구조 연구실

Our Solutions

- **Scalable architecture for DIV & SQRT**
  - **Provide versatility** for numerous conditions
  - **Minimize the resource** requirements



- **Compiler optimization**
  - **Minimize workloads** for applying posit

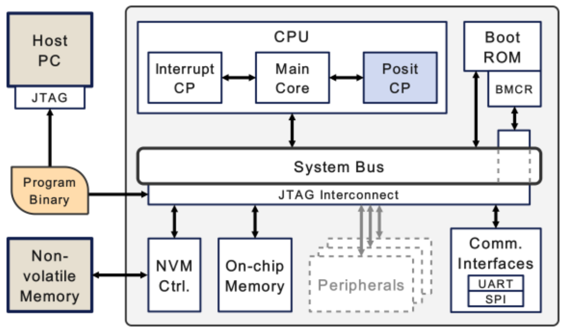


```
float f1=1.0f
float f2=1.5f
…
```

```
00:0x3F800000
04:0x3FC00000
…
```

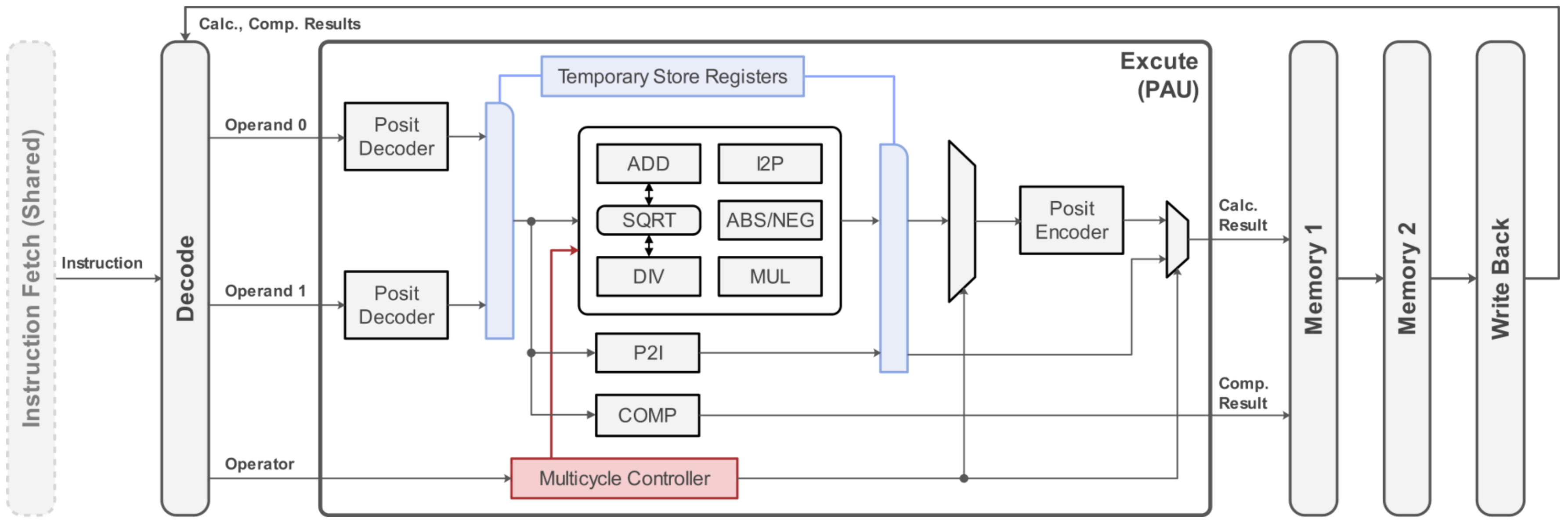
```
00:0x40000000
04:0x42000000
…
```

- **Evaluation platform**
  - Provide an **expedient evaluation method**



Coprocessor & PAU architecture

- **Tuned for 6-stage pipelined Processor**
  - Posit operations using FP instructions
  - Square root logic exploiting ADD and DIV

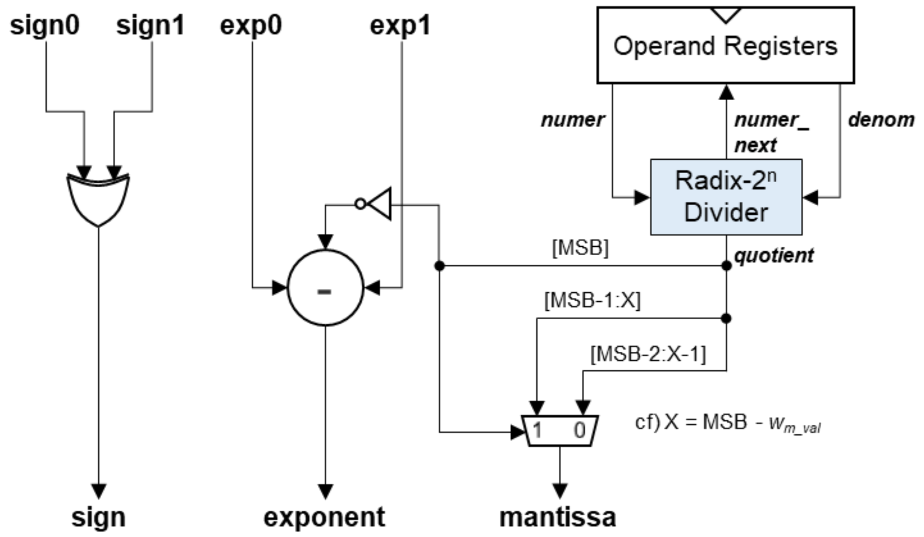


SEOULTECH

Computer Architecture Lab 컴퓨터 구조 연구실

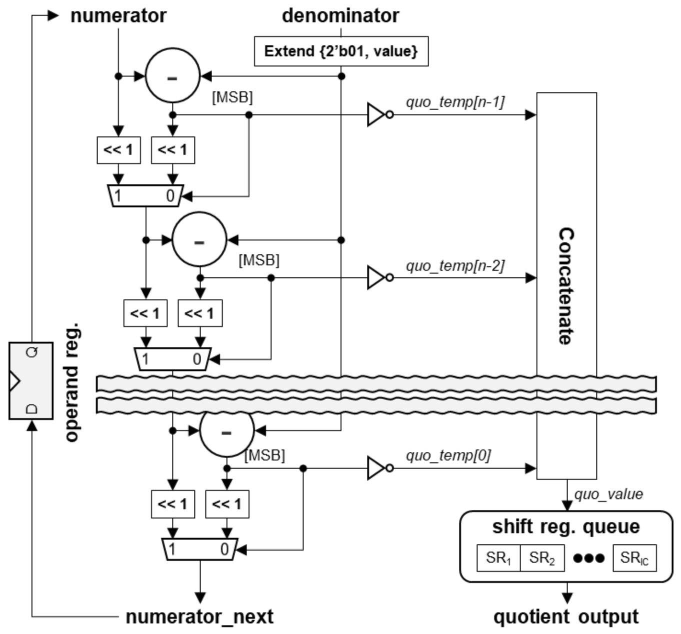
Scalable Divider

- **Posit n=32, es=3 settings**
  - 28-bit division is required to ensure the maximum precision



Scalable Divider

- **Radix-2^n divider generator**
  - Key factor (n): what number of digits is calculated in one cycle.
  - Flexibility for various conditions
    - Area $\propto n$
    - Power $\propto n$
    - Frequency $\propto 1/n$



Square Root Logic

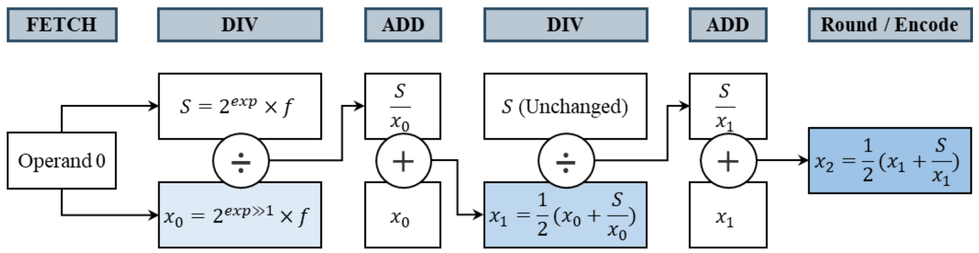
- **Babylonian Method**

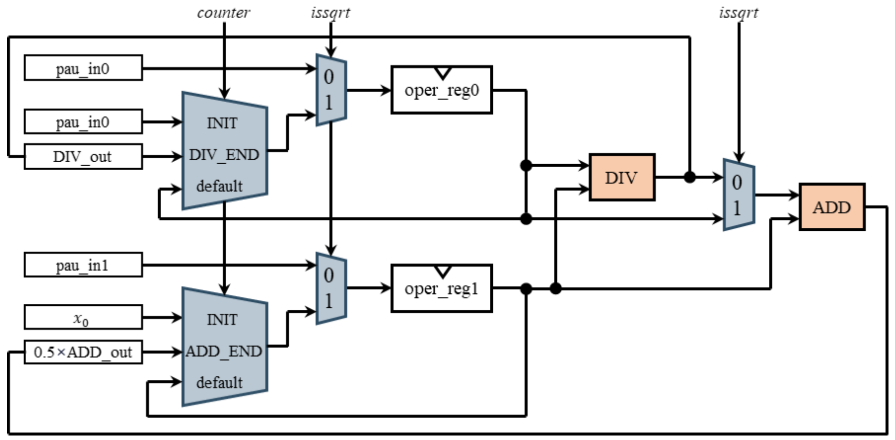
$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{S}{x_n}\right)$$

  - Requires division, addition, and multiplication
  - Based on convergence
    - Error (< 1) decreases by a square manner on each iteration.
    - Affected by the first estimated value (x_0)

Square Root Logic

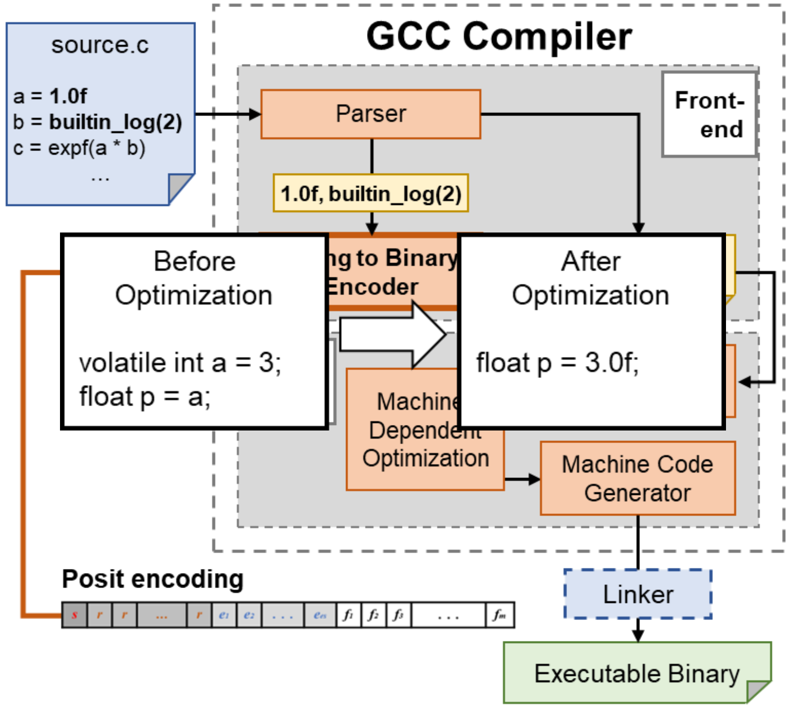
- **Implementation**
  - Designed with additional multiplexers and a low-precision subtractor.
    - Exploiting the existing adder and divider
    - Area-efficient
  - Estimate first x_0 as $2^{exp \gg 1} \times f$
    - Minimize the additional area overhead





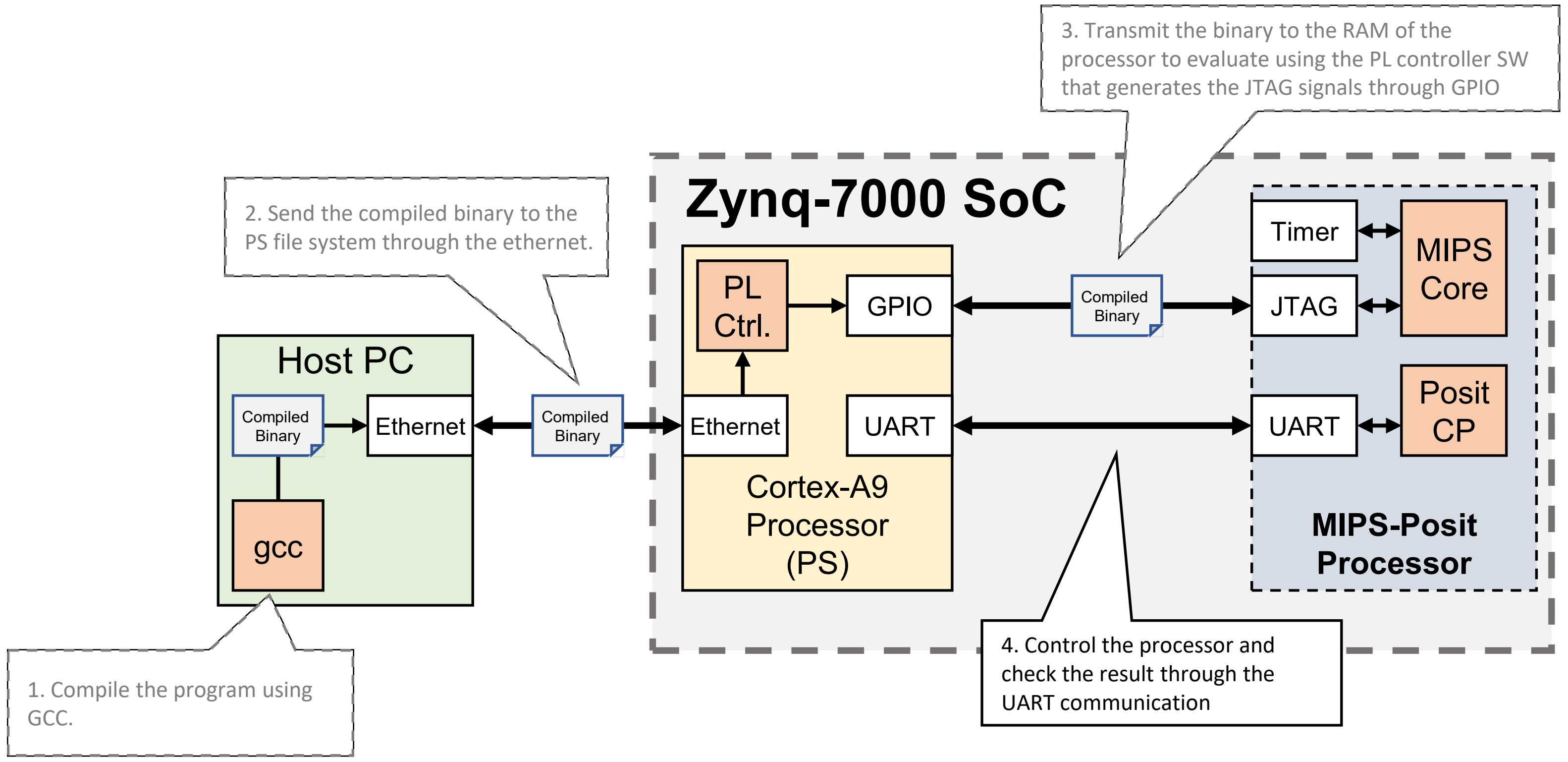
Adding Compiler Support

- GCC
  - Most of the features to generate FP instructions were already implemented
  - All we had to do is to change the encoding function for real numbers



Evaluation Platform

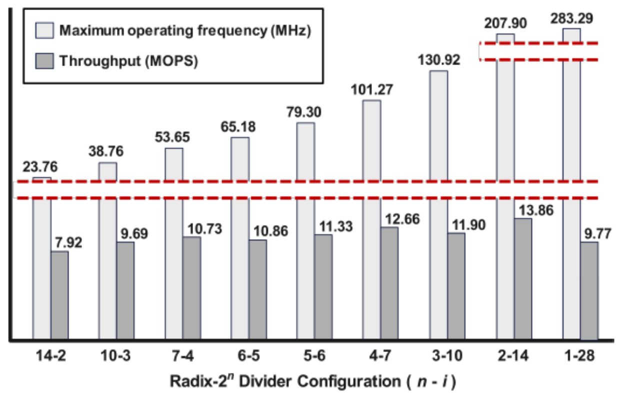
- **Exploiting the SoC FPGA**
  - Linux environment (PetaLinux)



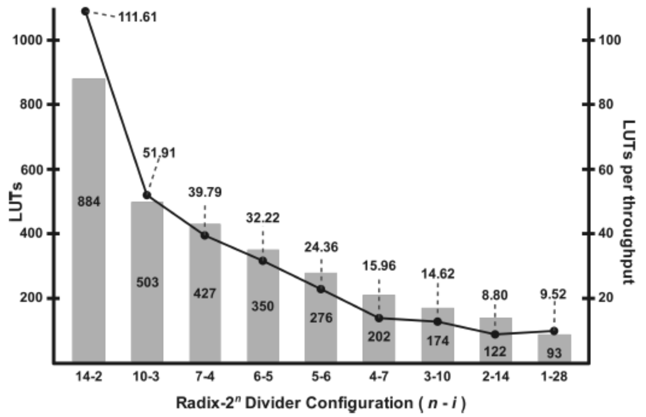
3. Transmit the binary to the RAM of the processor to evaluate using the PL controller SW that generates the JTAG signals through GPIO
Zynq-7000 SoC
2. Send the compiled binary to the PS file system through the ethernet.
Host PC
Compiled Binary
Ethernet
gcc
Compiled Binary
PL Ctrl.
GPIO
Ethernet
UART
Cortex-A9 Processor (PS)
Compiled Binary
Timer
JTAG
UART
MIPS Core
Posit CP
MIPS-Posit Processor
4. Control the processor and check the result through the UART communication
1. Compile the program using GCC.

Evaluation – Scalable Divider (FPGA)

- **Frequency & Throughput**
  - Varying frequency
    - Dynamic range: 11.92
  - Similar throughput
    - Dynamic range: 1.75



- **Area (LUTs)**
  - LUTs ∝ n
  - LUTs per throughput ∝ 1/n

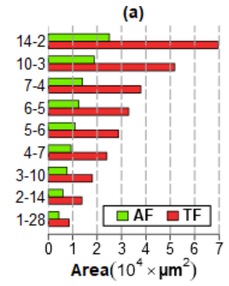


SEOULTECH

Computer Architecture Lab 컴퓨터 구조 연구실

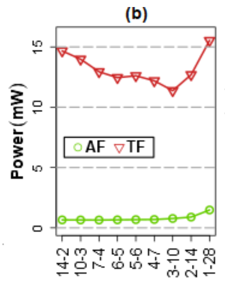
Evaluation – Scalable Divider (ASIC)

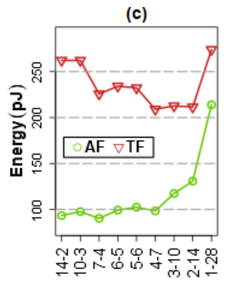
- **Area**



  - AF: 4322.35 ~ 25016.50 um^2
  - TF: 8563.48 ~ 69561.50 µm^2
  - Dynamic range: 16.09

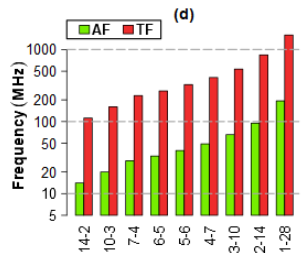
- **Power & Energy**



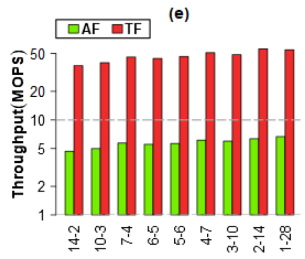


  - Power < 15.53 mW
  - Energy < 273.94 pJ

- **Frequency & Throughput**

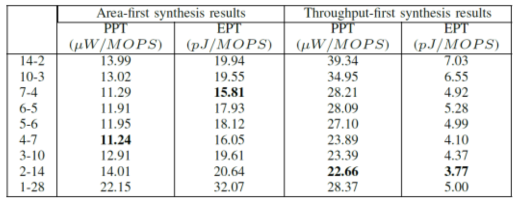


  - Frequency
    - AF: 14.04 ~ 193.42 MHz
    - TF: 111.86 ~ 1587.30 MHz
    - Dynamic range: 113.08



  - Throughput
    - AF: 4.68 ~ 6.67 MOPS
    - TF: 37.29 ~ 56.02 MOPS
    - Dynamic range: 11.97 / 1.43(AF) / 1.50 (TF)

- **Power & Energy per Throughput**



	Area-first synthesis results		Throughput-first synthesis results	
	PPT $(\mu W/MOPS)$	EPT $(pJ/MOPS)$	PPT $(\mu W/MOPS)$	EPT $(pJ/MOPS)$
14-2	13.99	19.94	39.34	7.03
10-3	13.02	19.55	34.95	6.55
7-4	11.29	**15.81**	28.21	4.92
6-5	11.91	17.93	28.09	5.28
5-6	11.95	18.12	27.10	4.99
4-7	**11.24**	16.05	23.89	4.10
3-10	12.91	19.61	23.39	4.37
2-14	14.01	20.64	**22.66**	**3.77**
1-28	22.15	32.07	28.37	5.00

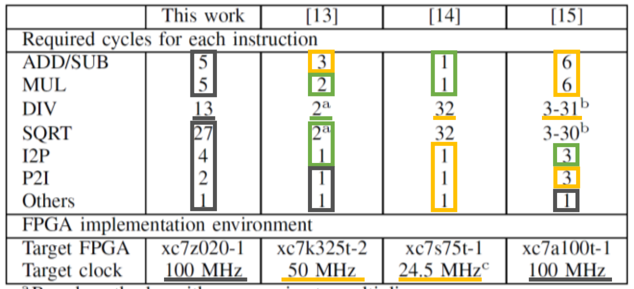
  - Best configurations
    - AF: n=7 (energy) / n=4 (power)
    - TF: n=2 (both)

SEOULTECH

Computer Architecture Lab 컴퓨터 구조 연구실

Evaluation – Processor

- **Throughput**
  - Best average throughput in exact DIV
  - Affordable performance
    - Highest operating frequency (= PERI [15])



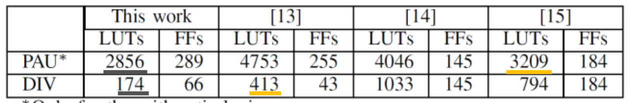
	PERCIVAL	PERC	PERI
	This work	[13]	[14]	[15]
Required cycles for each instruction				
ADD/SUB	5	3	1	6
MUL	5	2	1	6
DIV	13	2[a]	32	3-31[b]
SQRT	27	2[a]	32	3-30[b]
I2P	4	1	1	3
P2I	2	1	1	3
Others	1	1	1	1
FPGA implementation environment				
Target FPGA	xc7z020-1	xc7k325t-2	xc7s75t-1	xc7a100t-1
Target clock	100 MHz	50 MHz	24.5 MHz[c]	100 MHz

[a]Based on the logarithm-approximate multiplier.
[b]Proportionate to the bit-width of the mantissa part of numerators.
[c]Based on the critical path indicated in the timing report.

Baseline (our work)

🟨 < ⬛ < 🟩

- **Area**
  - PAU: 11.00% reduced LUTs
  - Divider: 57.47% reduced LUTs

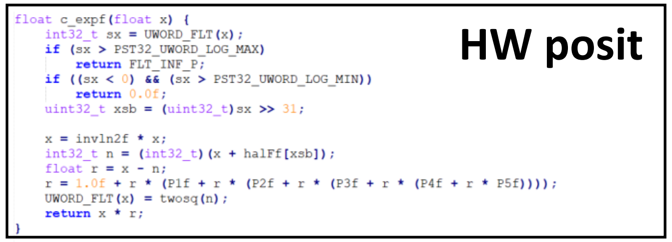


	This work		PERCIVAL [13]		PERC [14]		PERI [15]	
	LUTs	FFs	LUTs	FFs	LUTs	FFs	LUTs	FFs
PAU*	2856	289	4753	255	4046	145	3209	184
DIV	174	66	413	43	1033	145	794	184

*Only for the arithmetic logics.

Evaluation – HW Acceleration

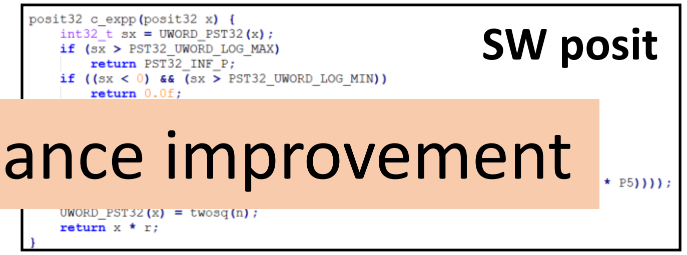
- Comparison with SW-only implementation
  - expf (fdlibm)
  - Ported using SW posit library



HW posit

```
float c_expf(float x) {
    int32_t sx = UWORD_FLT(x);
    if (sx > PST32_UWORD_LOG_MAX)
        return FLT_INF_P;
    if ((sx < 0) && (sx > PST32_UWORD_LOG_MIN))
        return 0.0f;
    uint32_t xsb = (uint32_t)sx >> 31;

    x = invln2f * x;
    int32_t n = (int32_t)(x + halFf[xsb]);
    float r = x - n;
    r = 1.0f + r * (P1f + r * (P2f + r * (P3f + r * (P4f + r * P5f))));
    UWORD_FLT(x) = twosq(n);
    return x * r;
}
```



SW posit

```
posit32 c_expp(posit32 x) {
    int32_t sx = UWORD_PST32(x);
    if (sx > PST32_UWORD_LOG_MAX)
        return PST32_INF_P;
    if ((sx < 0) && (sx > PST32_UWORD_LOG_MIN))
        return 0.0f;
                                                    * P5))));
    UWORD_PST32(x) = twosq(n);
    return x * r;
}
```

Up to **60.09x** performance improvement

- Best case: 768 iteration
  - HW acceleration: 174,360 cycles
  - SW implementation: 10,477,848 cycles

Conclusion

1) RF2P provides flexible, area-efficient posit HW acceleration using a **scalable divider generator** and **square root logic**.

2) Compiler optimization enables porting the SW from FP with only a few additional workloads.

3) The Evaluation platform provides a practical method to swiftly test the posit arithmetic.

4) Our processor can achieve **up to 60.09x** performance compared to SW, and the PAU requires an **11% lower area** than state-of-the-art processors in performance-aware settings for 100 MHz operating frequency

Q&A

Thank You!

ohhyunwoo@seoultech.ac.kr

SEOULTECH

Computer Architecture Lab
컴퓨터 구조 연구실