Effortless DSP extensions design for embedded RISC-V processors

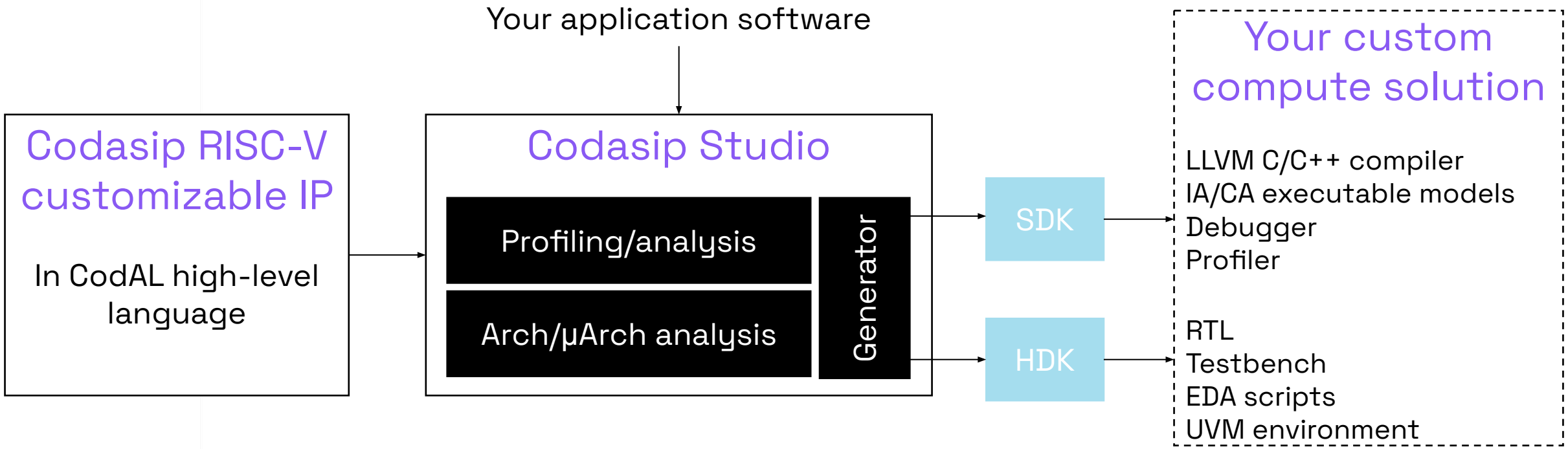
Alexey Shchekin, Codasip Gmbh
Ettore Antonino Giliberti, Codasip Group

Codasip

Motivation: embedded devices with DSP

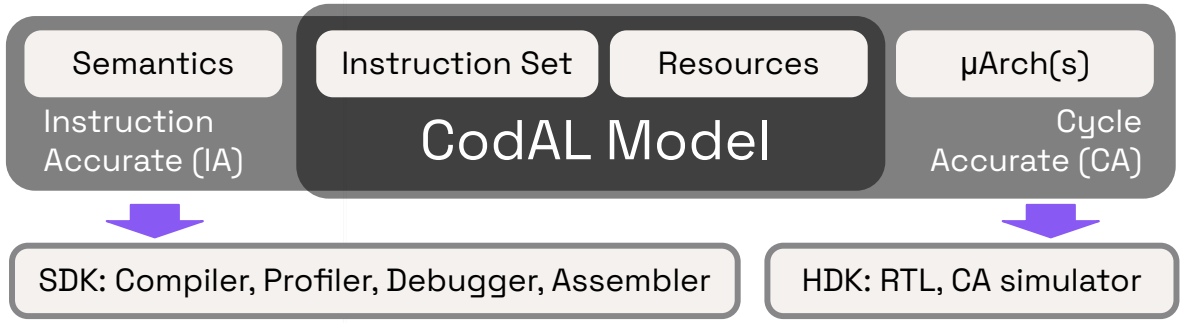
- Embedded devices are typically resource-constrained, making it difficult to efficiently run many signal-processing algorithms on embedded platforms
- Moving from general-purpose to application-specific cores can help improve the system performance and reduce power consumption, or decrease the silicon area and associated chip cost
- ASIC design is time and effort-consuming, customized processors need the dedicated compiler support and extended verification flow. The associated expenses may outweigh the benefits of application-specific designs
- This paper demonstrates an efficient and easy way to customize an embedded RISC-V core and extend it with some representative DSP modules (FFT, FIR & Median filtering, CORDIC) using the HLS approach enabled by CodAL and Codasip design tools

Codasip Studio and RISC-V Processors



- Start with a standard core
  - Embedded and application cores
  - High quality, production-ready
  - Fully RISC-V compliant
- Differentiate with Codasip Studio
  - Configure / Modify
  - Using CodAL architecture description language

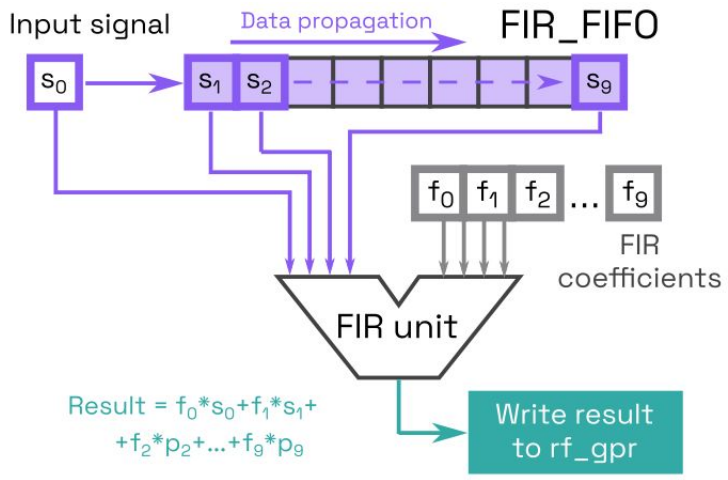
CodAL enables easy DSP customization



DSP example: FIR filter accelerator controlled by custom instructions

push_fir: pushes the new sample from src to fifo[], calculates the FIR result and stores it to the dst

push_coeff: loads FIR filter coefficients $f_0..f_n$ to dedicated registers



CodAL description of a single instruction that gets the FIR result

```
element i_fir {
   use reg_any as dst, src;   // src - new signal sample, dst - register for the FIR result
   assembly { "push_fir" dst "," src };                    // Assembly format
   binary { 0:bit[12] src PUSH_FIR dst OPC_X};             // Instruction binary pattern
   semantics
   {
      int32 ind, new_sample, result;
      new_sample = rf_gpr_read(src);   // Read the new sample from register
      result = new_sample * fir_coeff[0];
      for (ind=FIFO_DEPTH-1; ind>0; ind--) {   // Calculate FIR and shift the fir_fifo
         result += (fir_fifo[ind] * fir_coeff[ind]);
         fir_fifo[ind] = fir_fifo[ind-1];
      }
      fir_fifo[0] = new_sample;   // Push the new signal sample to the fir_fifo

      rf_gpr_write(dst, result);   // Store the FIR result to dst register
      codasip_inc_clock_cycle(3);   // Inform the simulator about instruction latency
   };
};
```

FIR accelerator for RISC-V core

Standard RISC-V

Number of clock cycles: 100+

Example C (FIR_DIM=10):

```
int fir = 0;
for (int i = 0; i < FIR_DIM; ++i)
{
        fir += input[i] * fir_coeff[i];
}
output = fir;
```

Used instructions:

- c.lw x 2*FIR_DIM
- mul x FIR_DIM
- c.add x 2*FIR_DIM
- jal x FIR_DIM
- bgeu x FIR_DIM
- c.sw x 1

Etc…

- c.lwsp, c.swsp, sh2add

+ CodAL implementation

```
element i_fir
{
   ....
   semantics
   {
      int32 val, result, i;
      new_sample = rf_gpr_read(src);
      result = new_sample * fir_coeff[0];
      for (i=FIFO_DEPTH-1; i>0; i--) {
         result += fir_fifo[i] * fir_coeff[i];
         fir_fifo[i] = fir_fifo[i-1];
      }
      fir_fifo[0] = new_sample;
      rf_gpr_write(dst, result);
      codasip_inc_clock_cycle(3);
   };
};
```

Customized RISC-V

Number of clock cycles: 6

Example C (inline ASM):

```
…
asm volatile ("push_fir %0, %1" :
"=r"(output[i]) : "r"(input[i]));
…
```

Disassembly:

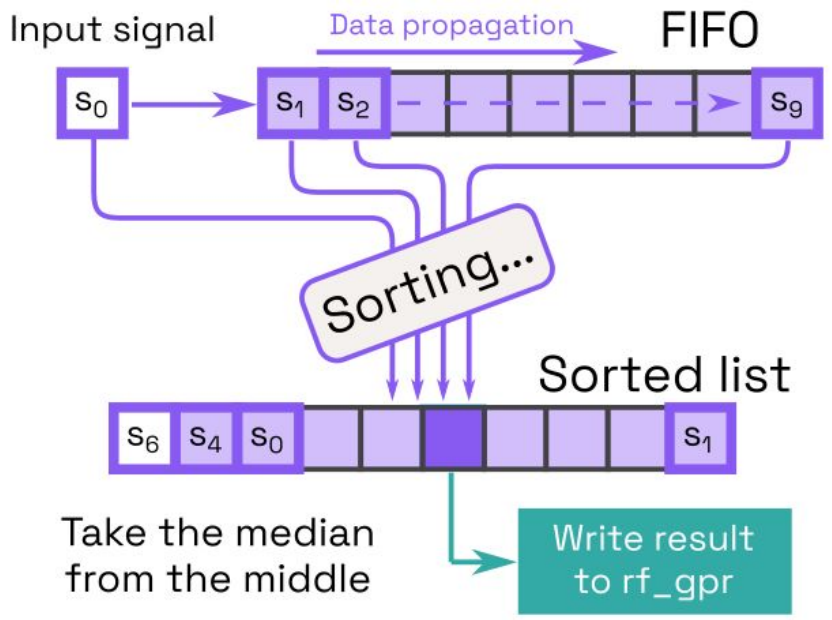
```
…
c.lw x10, 0x0 ( x10 )
push_fir x10, x10
c.sw x10, 0x0 ( x11 )
…
```

Used instructions:

- c.lw x 1
- push_fir x 1
- c.sw x 1

Median filter accelerator in CodAL

Median filter accelerator



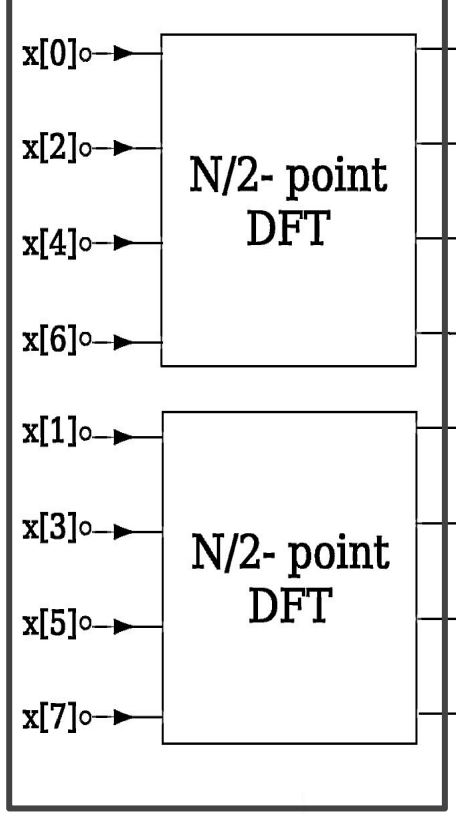
"push_median": pushes the new sample from **src** to fifo[], sorts the samples in the fifo[], picks the **median** value from the fifo[] cell in the middle and stores it to the **dst**

Single instruction gets the Median result over 10 samples

```
element i_fir {
  ...
  semantics
  {  //Get the new sample from src
     int32 i, new_sample, result;
     new_sample = rf_gpr_read(src);
     //Set the instruction latency
     codasip_inc_clock_cycle(3);
     //Find the oldest element
     old_pos = FIFO_DEPTH-1;
     for (i=0; i<FIFO_DEPTH; i++) {
        if (age[i]==FIFO_DEPTH-1){
           old_pos = i;
        } else age[i] += 1;
     }
     //Remove the oldest element
     for (i=old_pos; i<FIFO_DEPTH-1; i++) {
        med_fifo[i] = med_fifo[i+1];
        age[i] = age[i+1];
     }
     ...
     //Find the new element's position
     new_pos = FIFO_DEPTH-1;
     for (i=0; i<FIFO_DEPTH; i++) {
        s_shift[i] = (med_fifo[i] > val);
        if (s_shift[i] && new_pos>i)
           new_pos = i;
     }
     //Shift the bigger elements
     for (i = FIFO_DEPTH-1; i>0; i--) {
        if (s_shift[i]) {
           med_fifo[i] = med_fifo[i-1];
           age[i] = age[i-1];
        }
     }  //Insert the new element to FIFO
     med_fifo[new_pos] = new_sample;
     age[new_pos] = 0;
     //Take the median result
     result = med_fifo[FIFO_DEPTH/2];
     rf_gpr_write(dst, result);
  }; //Write it to the dst register
};
```

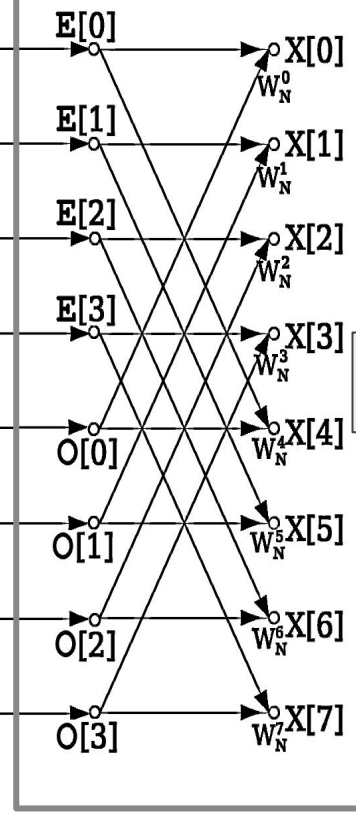
FFT: recursive calls and complex numbers

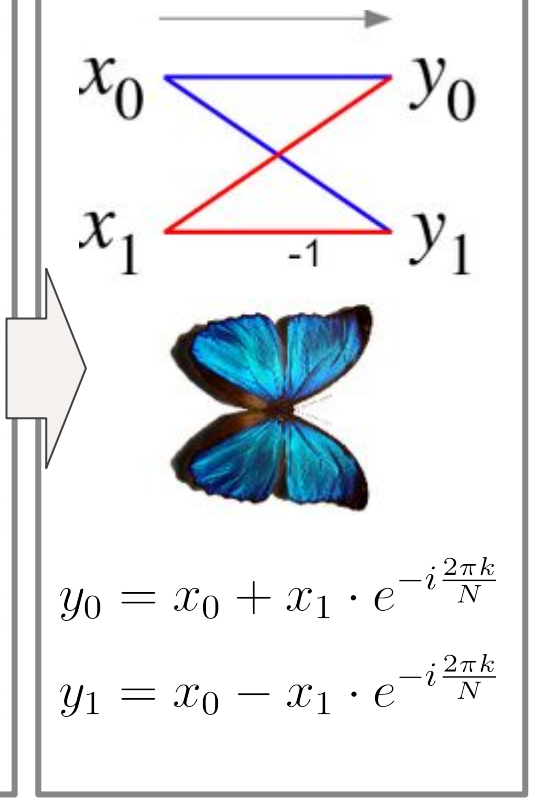
FFT recursion



$\sim N \cdot logN$

N/2 "butterfly" operations





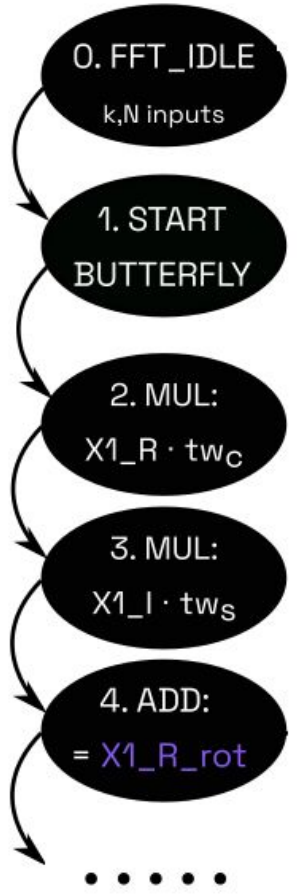
$$y_0 = x_0 + x_1 \cdot e^{-i\frac{2\pi k}{N}}$$

$$y_1 = x_0 - x_1 \cdot e^{-i\frac{2\pi k}{N}}$$

"Butterfly" $(x_0, x_1) \rightarrow (y_0, y_1)$

(0)	$\mathbf{x_0} \rightarrow (x_{0R}, x_{0I})$ $\mathbf{x_1} \rightarrow (x_{1R}, x_{1I})$
(1)	$tw_c = \cos(2\pi k/N)$
(1)	$tw_s = \sin(2\pi k/N)$
(2 - 4)	$x_{1R\_rot} = x_{1R} \cdot tw_c - x_{1I} \cdot tw_s$
(5 - 7)	$x_{1I\_rot} = x_{1R} \cdot tw_s + x_{1I} \cdot tw_c$
(8)	$y_{0R} = x_{0R} + x_{1R\_rot}$
(9)	$y_{0I} = x_{0I} + x_{1I\_rot}$
(10)	$y_{1R} = x_{0R} - x_{1R\_rot}$
(11)	$y_{1I} = x_{0I} - x_{1I\_rot}$

$(y_{0R}, y_{0I}) \rightarrow \mathbf{y_0}$; $(y_{1R}, y_{1I}) \rightarrow \mathbf{y_1}$



FFT acceleration & custom instructions set

Custom instructions for FFT accelerator, RISC-V compliant

“butterfly” instruction:

unused [31..25]	**rs_N**	**rs_k**	**opc=000**	unused [11..7]	*'custom-1' (= 0101011)*

- **k**, **N** (“butterfly” indices) - are taken from general-purpose registers (**rs_k**, **rs_N** operands)
- **x_0, x_1 inputs** - a pair of complex numbers are taken from internal registers *E_R*, *E_I*, *O_R*, *O_I*
- **Result y_0, y_1** is written back to *E_R*, *E_I*, *O_R*, *O_I*

“auxiliary” instructions (read/write from/to dedicated registers):

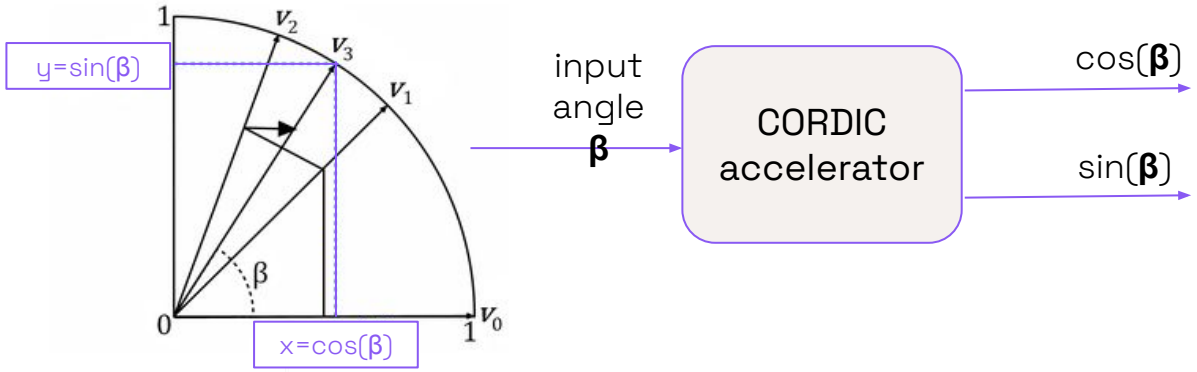
REG	unused [29..25]	**Imag_part**	**real_part**	**opc=001**	unused [11..7]	*'custom-1'*
REG	unused [29..15]			**opc=010**	**dst**	*'custom-1'*

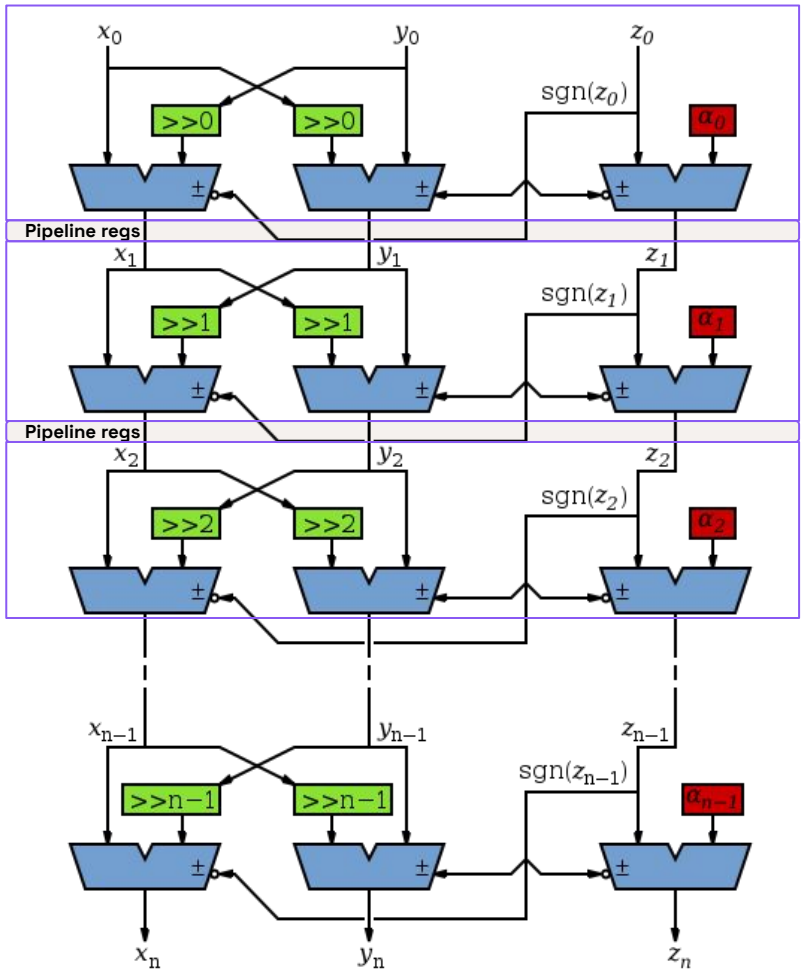
- **REG** - encodes *E_R*, *E_I*, *O_R*, *O_I* in 2 bits

CodAL: “butterfly” semantics

```
semantics
{
    float32 odd_r, odd_i, odd_r_rot, odd_i_rot, even_r,
            even_i, even_r_tmp, even_i_tmp, tw_c, tw_s;
    uXlen k, N;
        // unused by compiler
    codasip_compiler_unused();
    codasip_compiler_builtin();
        // load k,N from rf_gpr
    k = rf_gpr_read(rs_k);
    N = rf_gpr_read(rs_N);
        // load x0,x1 from dedicated registers
    even_r_tmp = codasip_bitcast_uint32_to_float32(E_R);
    even_i_tmp = codasip_bitcast_uint32_to_float32(E_I);
    odd_r = codasip_bitcast_uint32_to_float32(O_R);
    odd_i = codasip_bitcast_uint32_to_float32(O_I);
        // compute "twiddle" factors
    tw_c = codasip_cos_float32((2*M_PI*k)/N);
    tw_s = codasip_sin_float32((2*M_PI*k)/N);
        // do the job
    odd_r_rot = odd_r * tw_c - odd_i * tw_s;
    odd_i_rot = odd_r * tw_s + odd_i * tw_c;
    even_r = even_r_tmp + odd_r_rot;
    even_i = even_i_tmp + odd_i_rot;
    odd_r = even_r_tmp - odd_r_rot;
    odd_i = even_i_tmp - odd_i_rot;
        // write results back
    E_R = codasip_bitcast_float32_to_uint32(even_r);
    E_I = codasip_bitcast_float32_to_uint32(even_i);
    O_R = codasip_bitcast_float32_to_uint32(odd_r);
    O_I = codasip_bitcast_float32_to_uint32(odd_i);
};
```

CORDIC algorithm





CORDIC (COordinate Rotation DIgital Computer) is a hardware-efficient iterative method which uses rotations to calculate **sine** and **cosine**. This method implies :

- Iterative angle CW/CCW rotation by a certain (descending) values that depend on the iteration number
- A certain X, Y transformations that are based on simple arithmetics (additions, bitwise shift) and depend on the current iteration number and the angle value
- Once the input angle approaches the target with increasing accuracy, the X and Y approach the **sine** and **cosine** values.

CORDIC accelerator in CodAL

CORDIC accelerator usage

Implemented instructions:
"cordic": takes the input angle encoded in fixed-point 32-bit format from the register **src**, iteratively runs the CORDIC flow, calculates 16-bit fixed-point cos and sin values, puts them to the high- and low- half on the **dst** register.

C code, standard ISA:

```
#include <math.h>
float sin_a = sin(angle);
float cos_a = cos(angle);
```

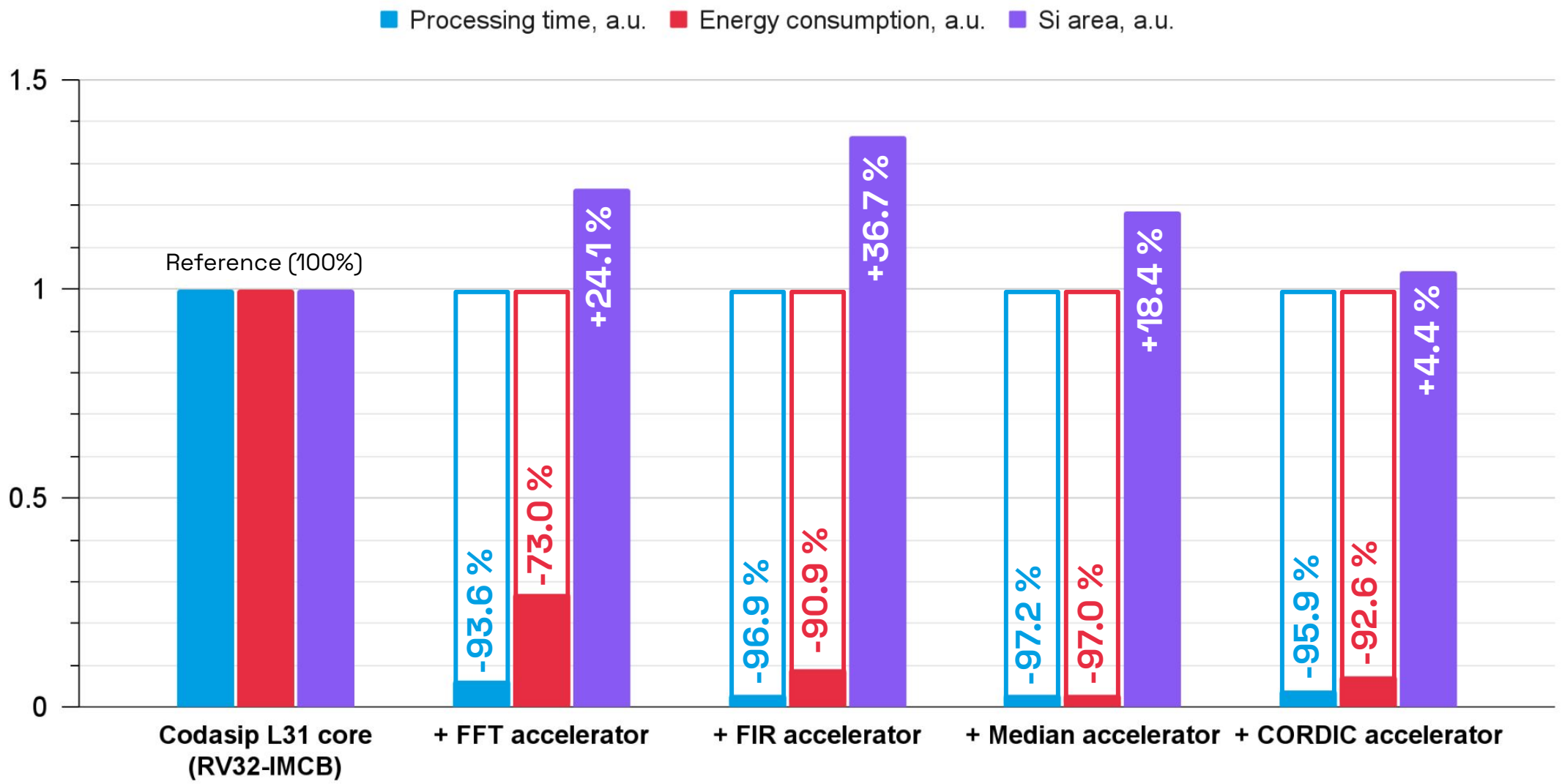
C code, custom ISA:

```
asm ("cordic %0, %1" : "=r"(result) : "r"(angle));
  +    int16<->float conversion
```

Single instruction runs the 16-cycle CORDIC flow

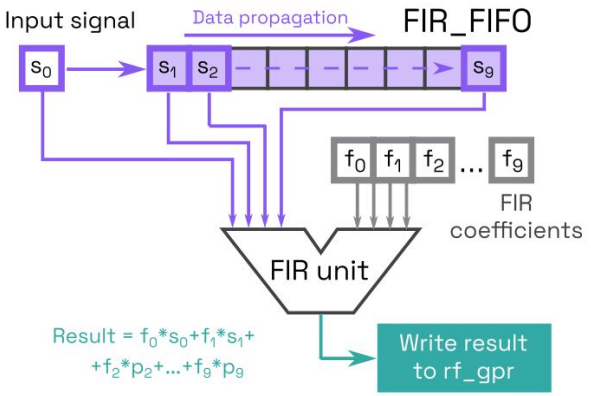
```
element i_cordic {
   assembly { "cordic" dst "," src };
   binary { 0:bit[12] src opc dst OPC_CORDIC};
   semantics {
      angle = rf_gpr_read(src);                  //Read the input angle from src
      cos = CORDIC_GAIN;                         //Set the initial cos,sin values
      sin = 0;
      for (shift=0; shift < ITERATIONS; shift++) {   // 16 iterations
         if (angle<0) {                          //Rotate CW if angle is negative
            cos += sin >>> shift;
            sin -= cos >>> shift;                //Next cos,sin values
            angle += tan[shift];
         } else {                                //Rotate CCW if angle is positive
            cos -= sin >>> shift;
            sin += cos >>> shift;                //Next cos,sin values
            angle -= tan[shift];    }
      }
      rf_gpr_write(dst, (cos :: sin));           //Write the results to dst
      codasip_inc_clock_cycle(16);               //Set the instruction latency
   };
};
```

How DSP customization affects the PPA



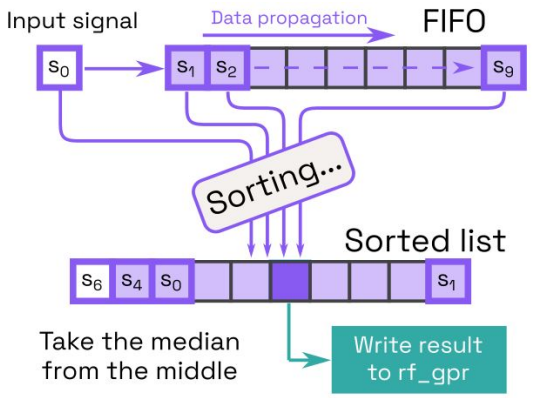
DSP accelerators implemented with CodAL

FIR filter accelerator

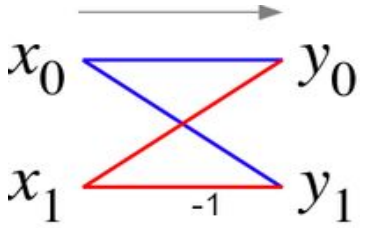


1 custom instruction call to get FIR or median value
3 cycles to get the result

Median filter accelerator



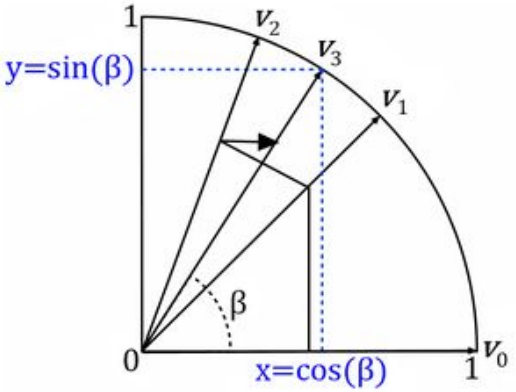
FFT accelerator



$$y_0 = x_0 + x_1 \cdot e^{-i\frac{2\pi k}{N}}$$
$$y_1 = x_0 - x_1 \cdot e^{-i\frac{2\pi k}{N}}$$

3 custom instructions call to start "butterfly" flow
42 cycles for fp32 result
20 cycles for fixed-point 32-bit

CORDIC accelerator



1 custom instruction call to start "CORDIC" flow
16 cycles to get the result

(TSMC 28nm)	**RV32-IMCB**	**RV + FIR**	**RV + Median**	**RV + FFT**	**RV + CORDIC**
Area, a.u.	100%	124.3%	118.4%	124.3%	104.4%
Max frequency	650	600	550	600	650
Performance gain	1x	14.4x	30x	14.4x	24.3x
Energy cons.	1x	0.1x	0.03x	0.27x	0.08x

Summary

1. DSP-specific customizations of embedded RISC-V cores that tackle several representative DSP algorithms (FFT, FIR & Median filtering, CORDIC) have improved the performance by at least **15** times, and energy consumption by at least **4** times, at reasonable silicon area cost.
2. Customizations with CodAL shorten the time to market by providing out-of-the-box SDK and HDK tools and more compact core description. The table below lists the code size and the human effort required to implement the described DSP customizations:

HW module	Time to implement in CodAL	Lines of CodAL code	Lines of code in Verilog
1D FFT accelerator (256 samples)	2 person-weeks	500	2300
1D FIR filter	3 person-days	150	670
1D Median filter	3 person-days	160	1180
CORDIC module	3 person-days	210	600

“Butterfly” instruction implementation

CodAL: “butterfly” semantics

```
semantics
{
    float32 odd_r, odd_i, odd_r_rot, odd_i_rot, even_r,
            even_i, even_r_tmp, even_i_tmp, tw_c, tw_s;
    uXlen k, N;
        // unused by compiler
    codasip_compiler_unused();
    codasip_compiler_builtin();
        // load k,N from rf_gpr
    k = rf_gpr_read(rs_k);
    N = rf_gpr_read(rs_N);
        // load x0,x1 from dedicated registers
    even_r_tmp = codasip_bitcast_uint32_to_float32(E_R);
    even_i_tmp = codasip_bitcast_uint32_to_float32(E_I);
    odd_r = codasip_bitcast_uint32_to_float32(O_R);
    odd_i = codasip_bitcast_uint32_to_float32(O_I);
        // compute "twiddle" factors
    tw_c = codasip_cos_float32((2*M_PI*k)/N);
    tw_s = codasip_sin_float32((2*M_PI*k)/N);
        // do the job
    odd_r_rot = odd_r * tw_c - odd_i * tw_s;
    odd_i_rot = odd_r * tw_s + odd_i * tw_c;
    even_r = even_r_tmp + odd_r_rot;
    even_i = even_i_tmp + odd_i_rot;
    odd_r = even_r_tmp - odd_r_rot;
    odd_i = even_i_tmp - odd_i_rot;
        // write results back
    E_R = codasip_bitcast_float32_to_uint32(even_r);
    E_I = codasip_bitcast_float32_to_uint32(even_i);
    O_R = codasip_bitcast_float32_to_uint32(odd_r);
    O_I = codasip_bitcast_float32_to_uint32(odd_i);
};
```

FFT: plain C

```
void fft(float* x_r, float* x_i, float* y_r, float* y_i, int N)
{
    if (N==1)   //Recursion exit
    {
        y_r[0] = x_r[0];
        y_i[0] = x_i[0];
    }
    else
    {
        //<Allocate memory for x_r_even, x_r_odd, x_i_even, x_i_odd>
        //<and fill them with corresponding numbers from x_r, x_i>
            //recursively call fft(N/2)
        fft(x_r_even, x_i_even, y_r, y_i, N/2);
        fft(x_r_odd, x_i_odd, y_r+N/2, y_i+N/2, N/2);
        for (int k=0; k<N/2; k++)
        {
            float even_r = y_r[k];
            float even_i = y_i[k];
            float odd_r = y_r[k+N/2];
            float odd_i = y_i[k+N/2];
            float tw_c = cos(2*M_PI*k/N);
            float tw_s = sin(2*M_PI*k/N);
            float odd_r_rot = odd_r * tw_c - odd_i * tw_s;
            float odd_i_rot = odd_r * tw_s + odd_i * tw_c;
            y_r[k] = even_r + odd_r_rot;
            y_i[k] = even_i + odd_i_rot;
            y_r[k+N/2] = even_r - odd_r_rot;
            y_i[k+N/2] = even_i - odd_i_rot;
        }
        //<Free memory for x_r_even, x_r_odd, x_i_even, x_i_odd>
    }
}
```

$\mathbf{x_0} \rightarrow (E_R, E_I)$; $\mathbf{x_1} \rightarrow (O_R, O_I)$:

$x_0 = E_R + i \cdot E_I$

$x_1 = O_R + i \cdot O_I$

$tw_c = \cos(2\pi k/N)$

$tw_s = \sin(2\pi k/N)$

$O_{R\_rot} = O_R \cdot tw_c - O_I \cdot tw_s$

$O_{I\_rot} = O_R \cdot tw_s + O_I \cdot tw_c$

$E_R = E_R + O_{R\_rot}$

$E_I = E_I + O_{I\_rot}$

$O_R = E_R - O_{R\_rot}$

$O_I = E_I - O_{I\_rot}$

$(E_R, E_I) \rightarrow \mathbf{y_0}$; $(O_R, O_I) \rightarrow \mathbf{y_1}$

```
asm ("load_even %0, %1" :: "r"(even_r), "r"(even_i));
asm ("load_odd %0, %1" :: "r"(odd_r), "r"(odd_i));

asm ("butterfly %0, %1, %2" :: "r"(k), "r"(N));

asm ("store_e_r %0" : "=r"(even_r) : );
asm ("store_e_i %0" : "=r"(even_i) : );
asm ("store_o_r %0" : "=r"(odd_r) : );
asm ("store_o_i %0" : "=r"(odd_i) : );
```

Custom instructions usage

FFT unit states

"Butterfly" $(x_0, x_1) \rightarrow (y_0, y_1)$

(0) $\mathbf{x_0} \rightarrow (x_{0R}, x_{0I})$
$\mathbf{x_1} \rightarrow (x_{1R}, x_{1I})$

(1) $tw_c = \cos(2\pi k/N)$

(1) $tw_s = \sin(2\pi k/N)$

(2 - 4) $x_{1R\_rot} = x_{1R} \cdot tw_c - x_{1I} \cdot tw_s$

(5 - 7) $x_{1I\_rot} = x_{1R} \cdot tw_s + x_{1I} \cdot tw_c$

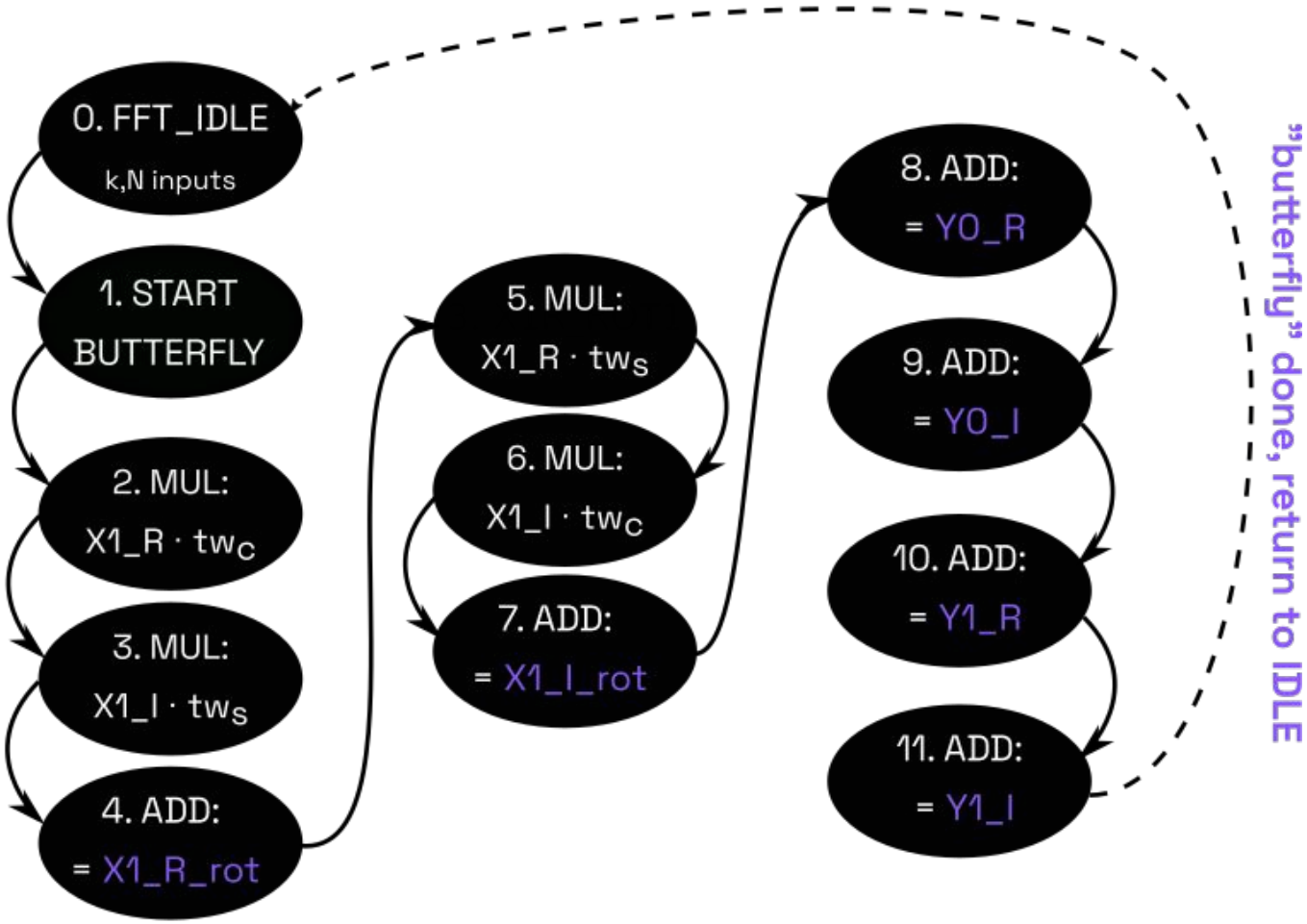
(8) $y_{0R} = x_{0R} + x_{1R\_rot}$

(9) $y_{0I} = x_{0I} + x_{1I\_rot}$

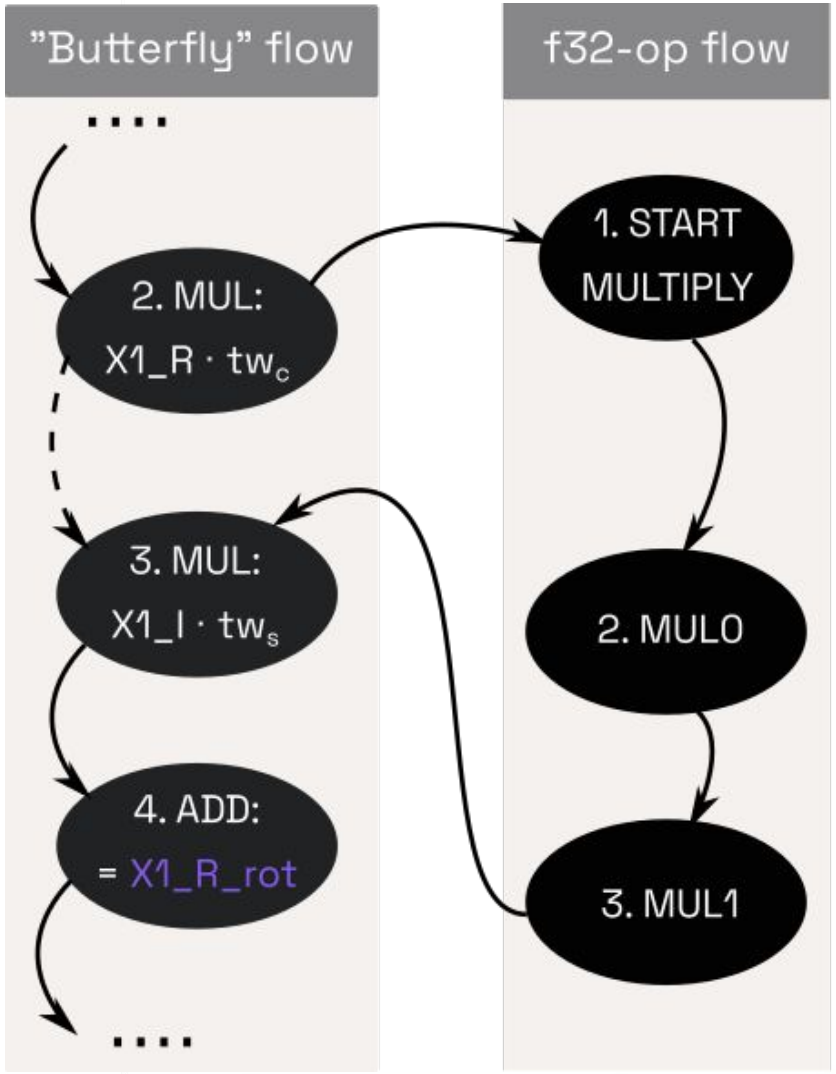
(10) $y_{1R} = x_{0R} - x_{1R\_rot}$

(11) $y_{1I} = x_{0I} - x_{1I\_rot}$

$(y_{0R}, y_{0I}) \rightarrow \mathbf{y_0}$; $(y_{1R}, y_{1I}) \rightarrow \mathbf{y_1}$



FFT unit states




```
case FFT_FPU_START_MUL:
    r_mul_res = 0;
    if (r_s1_data_i[30..0]==0 || r_s2_data_i[30..0]==0)
    {
        r_state = s_flush_i ? FFT_FPU_IDLE : r_return_state;
    }
    else
    {
        r_mul_sign1 = r_s1_data_i[SIGN_BIT];
        r_mul_sign2 = r_s2_data_i[SIGN_BIT];
        r_mul_man1 = (0::*26) :: 1 :: r_s1_data_i[22..0];
        r_mul_exp1 = (0::*8) :: r_s1_data_i[30..23];
        r_mul_man2 = (0::*26) :: 1 :: r_s2_data_i[22..0];
        r_mul_exp2 = (0::*8) :: r_s2_data_i[30..23];
        r_state = s_flush_i ? FFT_FPU_IDLE : FFT_FPU_COMPUTE_MUL0;
    }
    break;
case FFT_FPU_COMPUTE_MUL0:
    r_mul_sign1 = r_mul_sign1 ^ r_mul_sign2;
    r_mul_man1 = r_mul_man1 * r_mul_man2;
    r_mul_exp1 = r_mul_exp1 + r_mul_exp2 - 127;
    r_state = s_flush_i ? FFT_FPU_IDLE : FFT_FPU_COMPUTE_MUL1;
    break;
case FFT_FPU_COMPUTE_MUL1:
    mant_shift = r_mul_man1[47..47];
    shift_exp = r_mul_exp1[7..0] + 1;
    if (mant_shift)
    {
        r_mul_res = r_mul_sign1 :: shift_exp :: r_mul_man1[46..24];
    }
    else r_mul_res = r_mul_sign1 :: r_mul_exp1[7..0] :: r_mul_man1[45..23];
    r_state = s_flush_i ? FFT_FPU_IDLE : r_return_state;
    break;
```

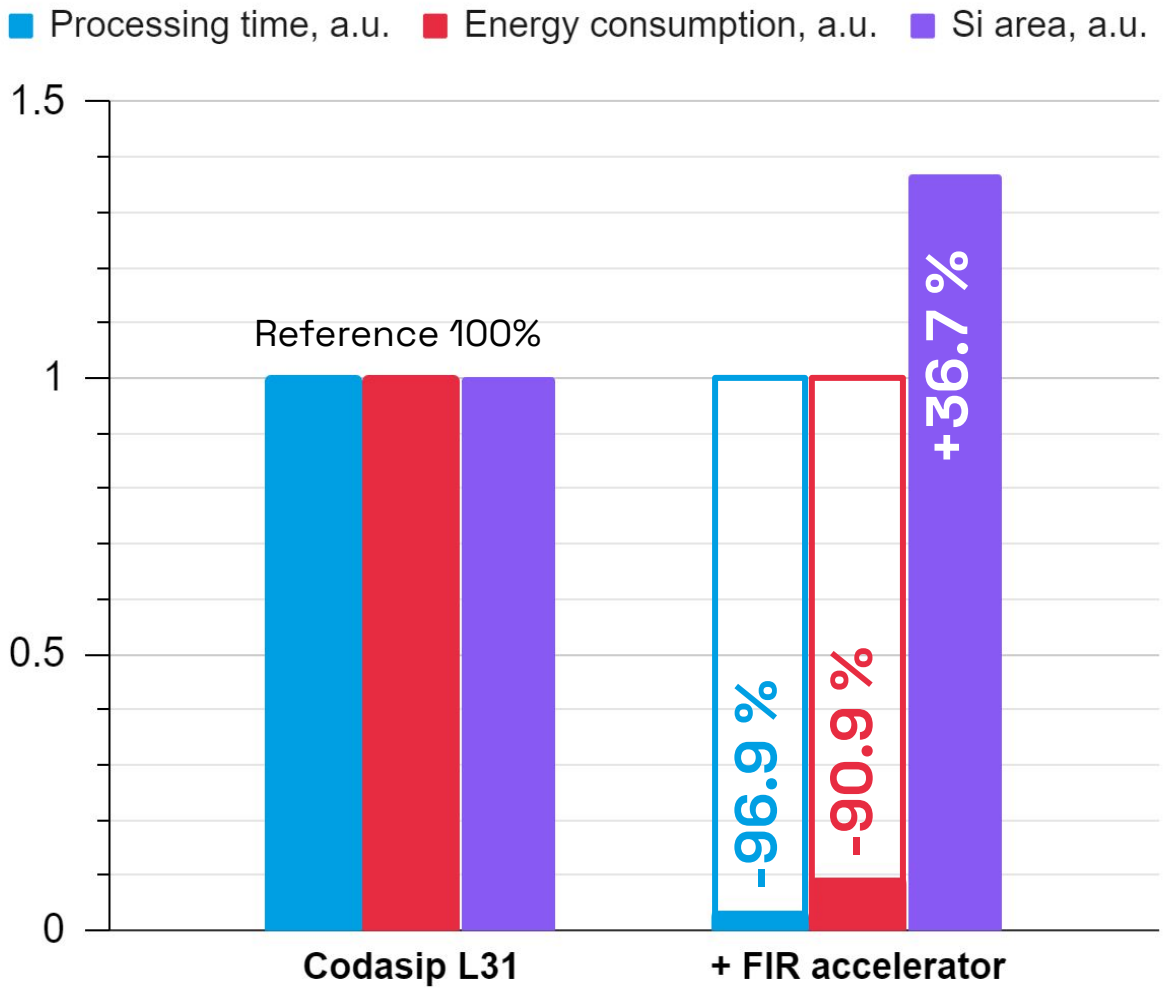
FIR filter accelerator

JULY 9–13, 2023

MOSCONE WEST CENTER

SAN FRANCISCO, CA, USA

FIR accelerator for RISC-V core: PPA effect

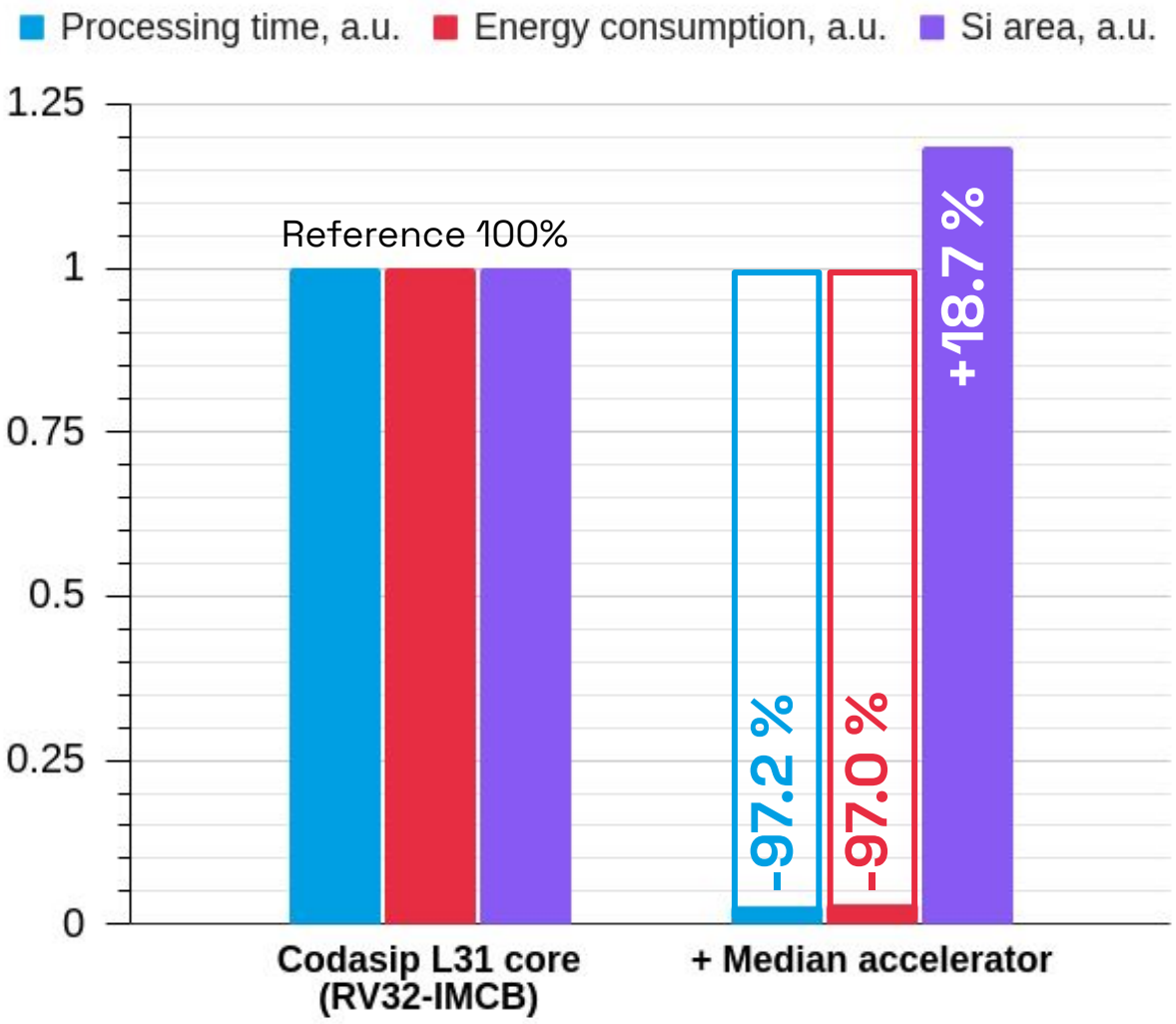


1 custom instruction call for 1 FIR result
3 cycles to get FIR result

TSCM 28 nm	**RISC-V(L31)**	**+ FIR**
Area, a.u.	100%	136.7%
Max frequency	650 MHz	650 MHz
Performance gain	1x	32x

Design time, lines of code in CodAL		**Lines of code in Verilog**
3 person-days	150	670 (~**4.5x**)

Median filter accelerator for L31 core: PPA

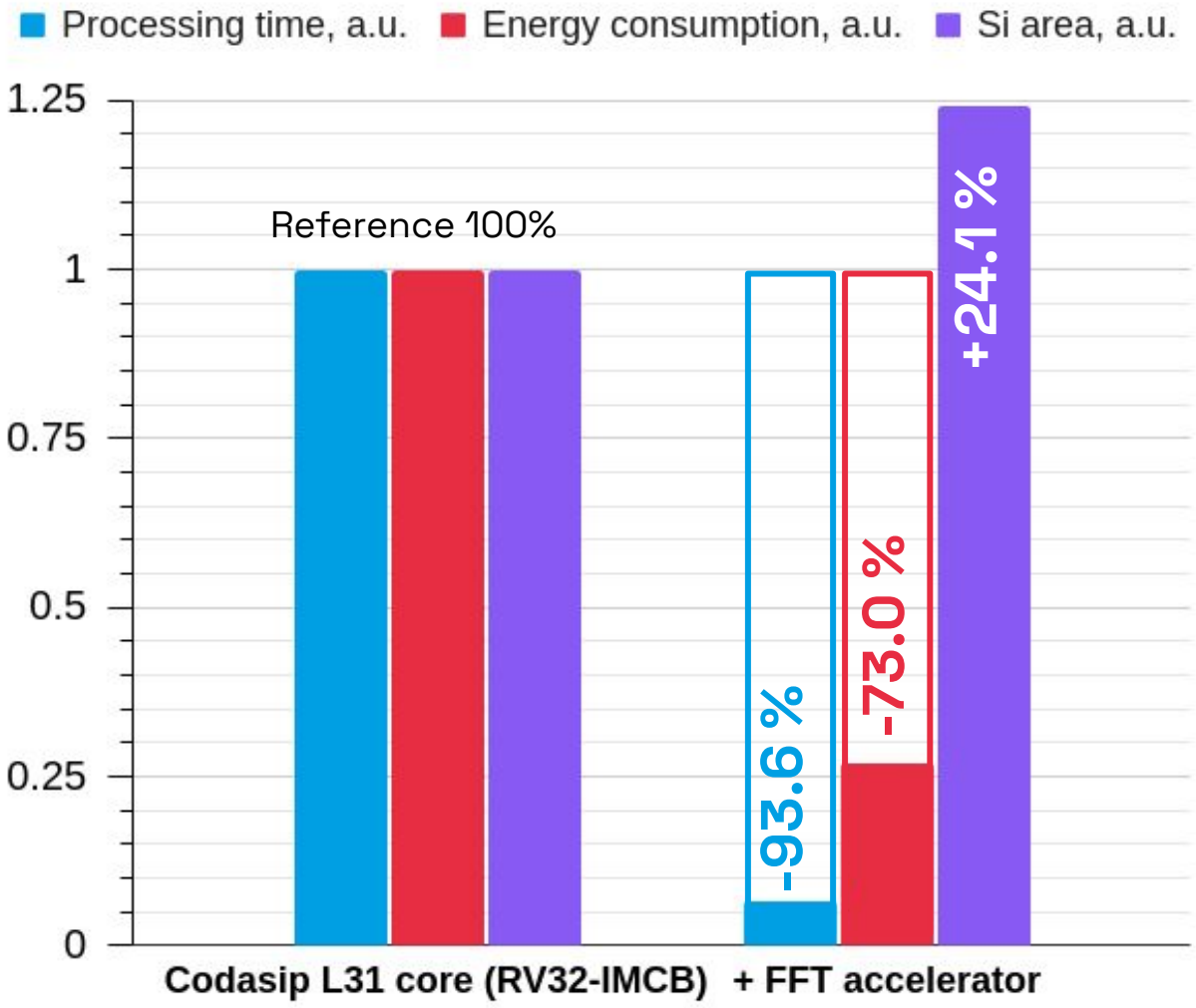


1 custom instruction call for 1 'median' result
3 cycles to get the result

TSCM 28 nm	RISC-V(L31)	+ Median
Area, a.u.	100%	118.7%
Max frequency	650 MHz	550 MHz
Performance gain	1x	**30x**

Design time, lines of code in CodAL		Lines of code in Verilog
3 person-days	160	1180 (~**7.3x**)

FFT accelerator for RISC-V core: PPA effect

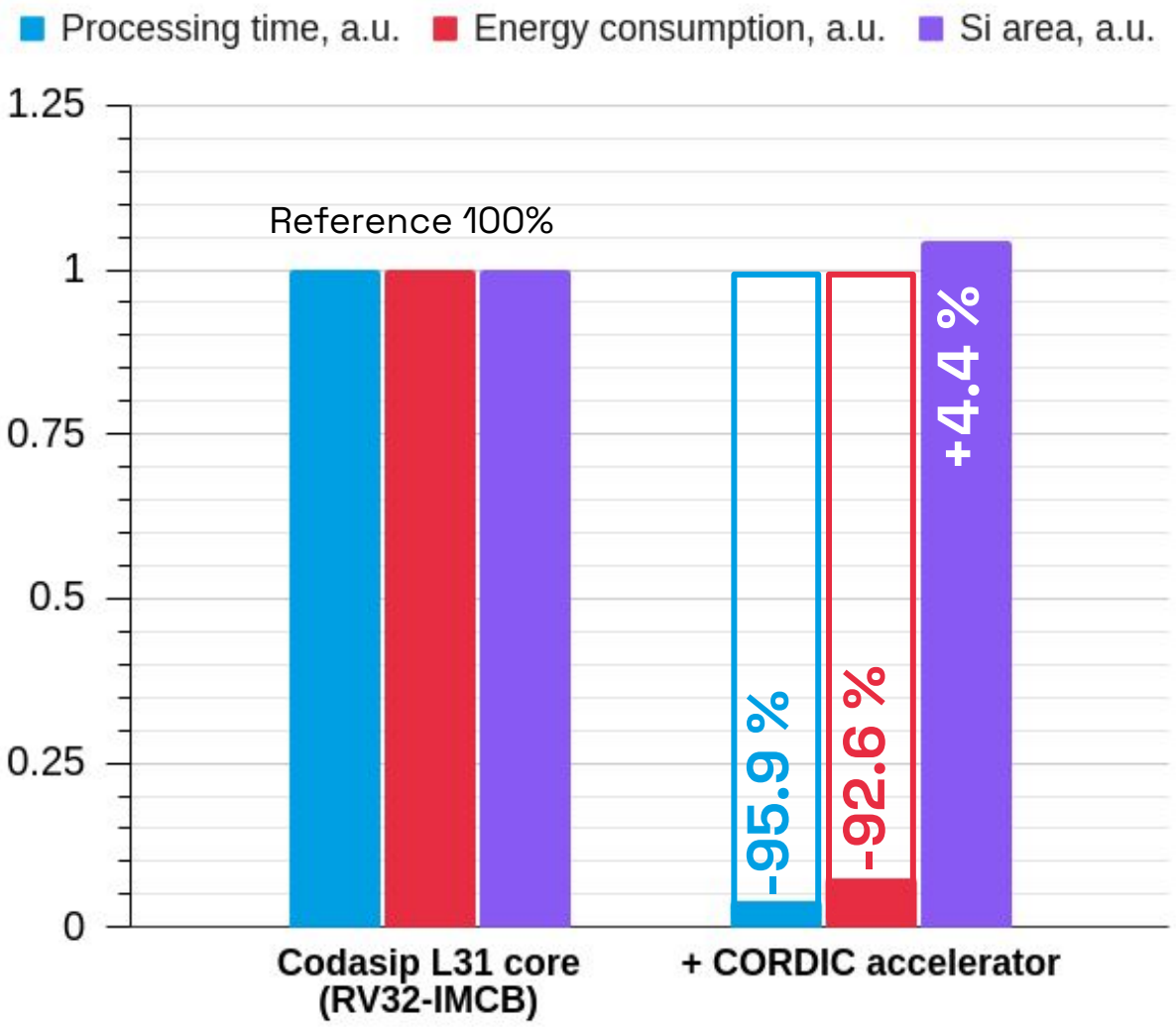


1 custom instruction call for 1 “butterfly” pair
42 cycles to get “butterfly” result (float32)
20 cycles to get “butterfly” result (fixed-point)

TSCM 28 nm	RISC-V (L31)	+ FIR
Area, a.u.	100%	124.1%
Max frequency	650 MHz	600 MHz
Performance gain	1x	14.4x

Design time, lines of code in CodAL		Lines of code in Verilog
2 person-weeks	500	2300 (~**4.6x**)

CORDIC accelerator for RISC-V: PPA effect

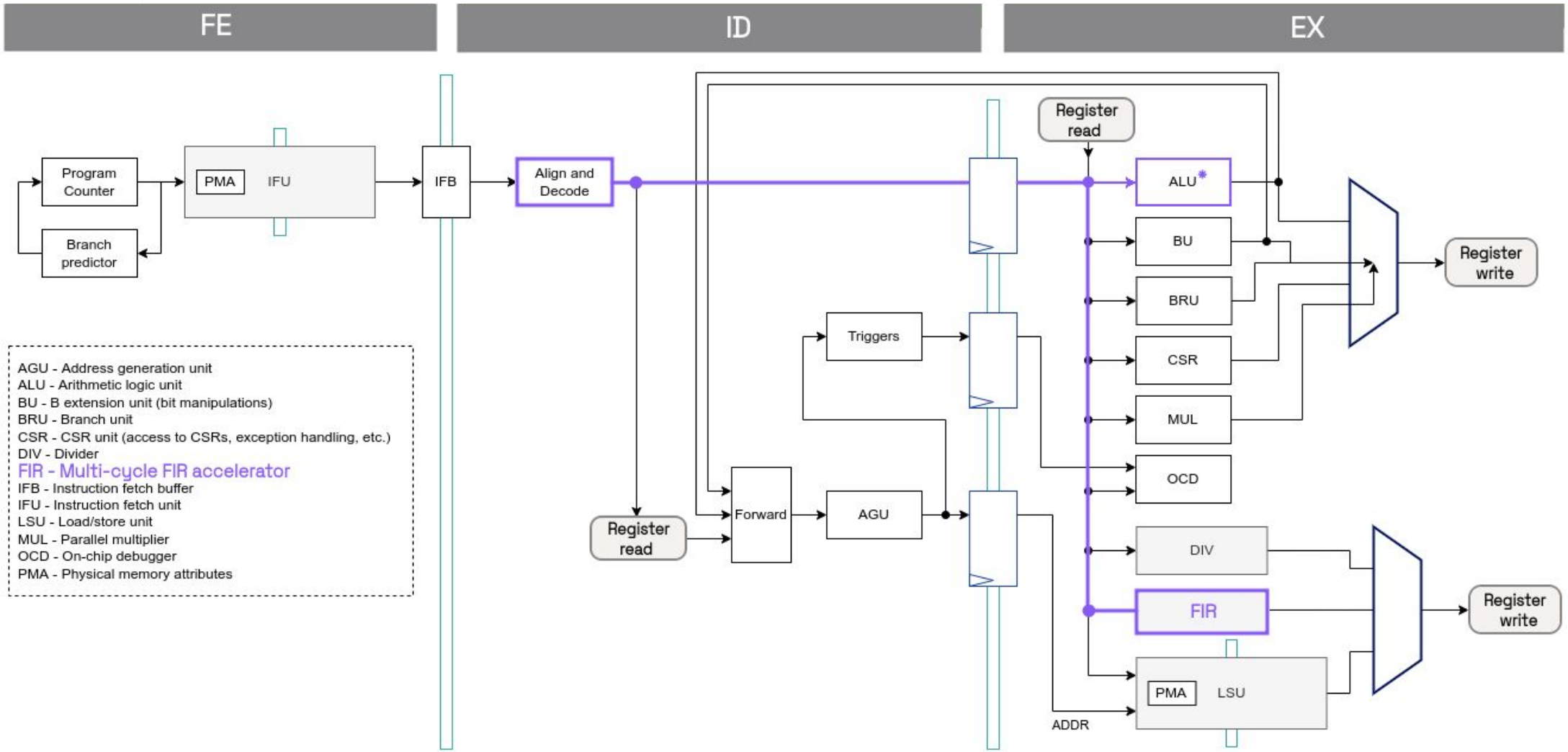


1 custom instructions call for “CORDIC” computation
16 cycles to get “CORDIC” results (sine and cosine)
16-bit fixed-point results representation

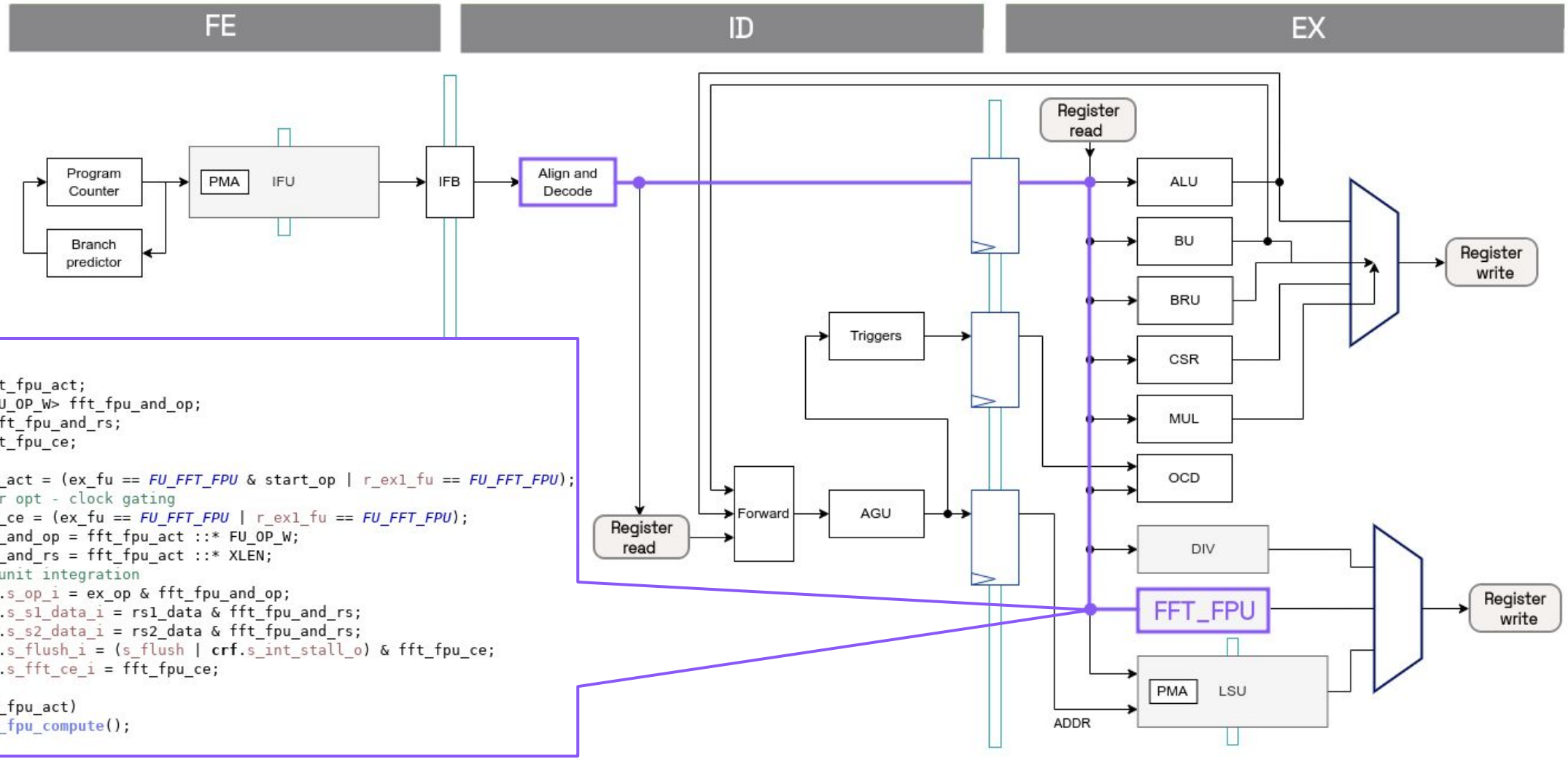
(TSMC 28nm)	RISC-V(L31)	+ CORDIC
Area, a.u.	100%	104.4%
Max frequency	650	650
Performance gain	1x	24.3x

Design time, lines of code in CodAL		Lines of code in Verilog
3 person-days	210	600 (~**3x**)

FIR accelerator for L31 core: System diagram



Microarchitecture: FFT as a multi-cycle unit

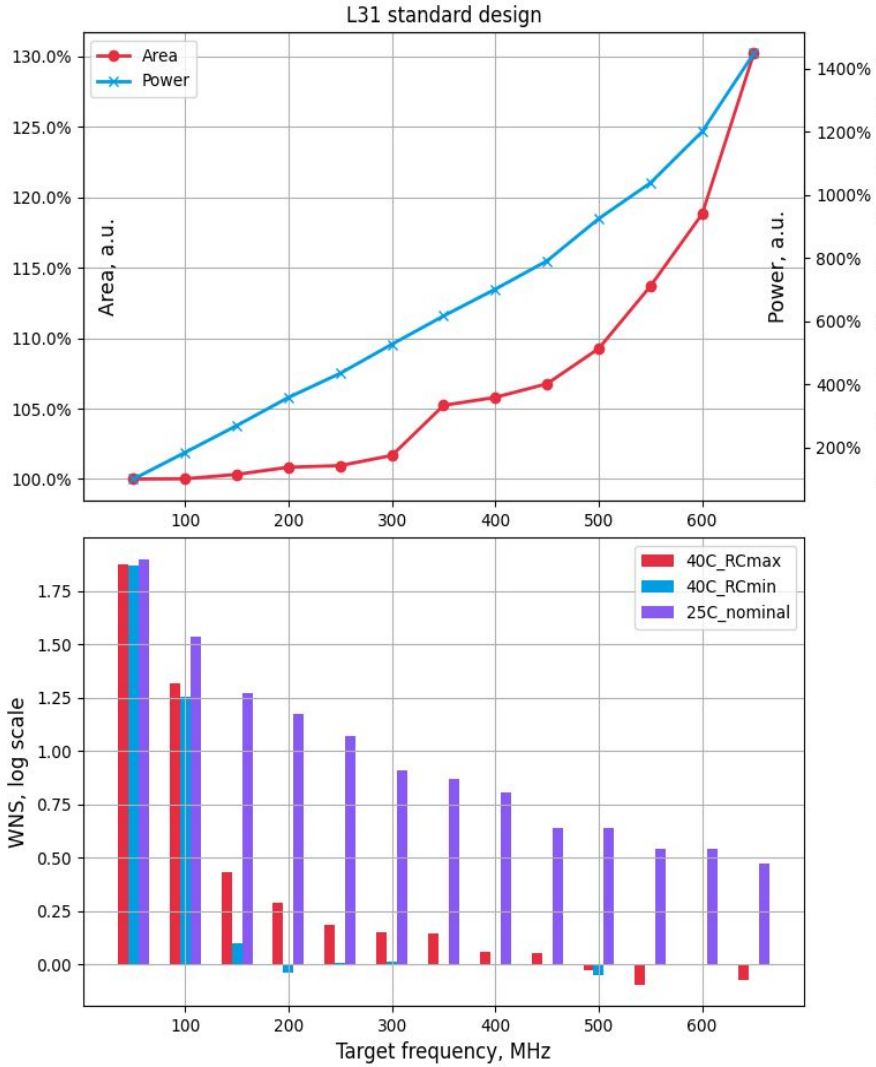


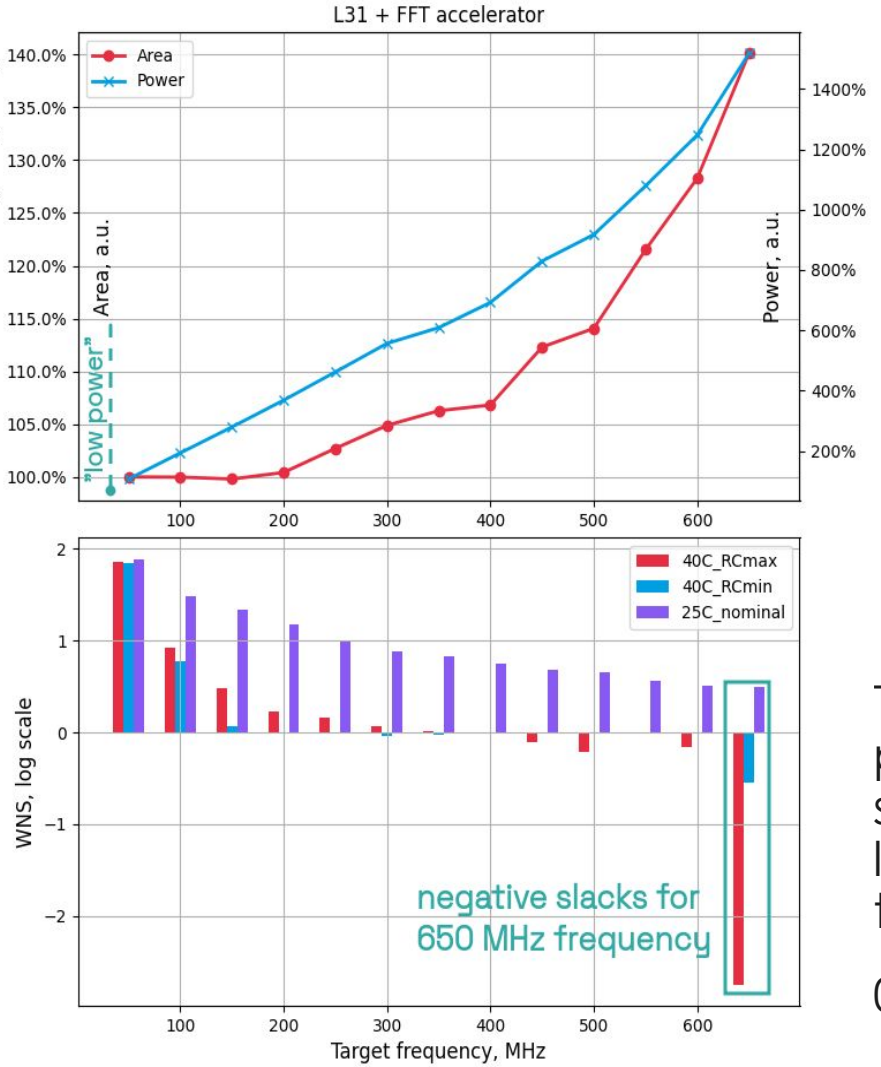

```
// FFT_FPU
{
    bool fft_fpu_act;
    uint_<FU_OP_W> fft_fpu_and_op;
    uXlen fft_fpu_and_rs;
    bool fft_fpu_ce;

    fft_fpu_act = (ex_fu == FU_FFT_FPU & start_op | r_ex1_fu == FU_FFT_FPU);
    // power opt - clock gating
    fft_fpu_ce = (ex_fu == FU_FFT_FPU | r_ex1_fu == FU_FFT_FPU);
    fft_fpu_and_op = fft_fpu_act ::* FU_OP_W;
    fft_fpu_and_rs = fft_fpu_act ::* XLEN;
    // FFT unit integration
    fft_fpu.s_op_i = ex_op & fft_fpu_and_op;
    fft_fpu.s_s1_data_i = rs1_data & fft_fpu_and_rs;
    fft_fpu.s_s2_data_i = rs2_data & fft_fpu_and_rs;
    fft_fpu.s_flush_i = (s_flush | crf.s_int_stall_o) & fft_fpu_ce;
    fft_fpu.s_fft_ce_i = fft_fpu_ce;

    if (fft_fpu_act)
        fft_fpu_compute();
}
```

FFT accelerator: Synthesis results (28nm)



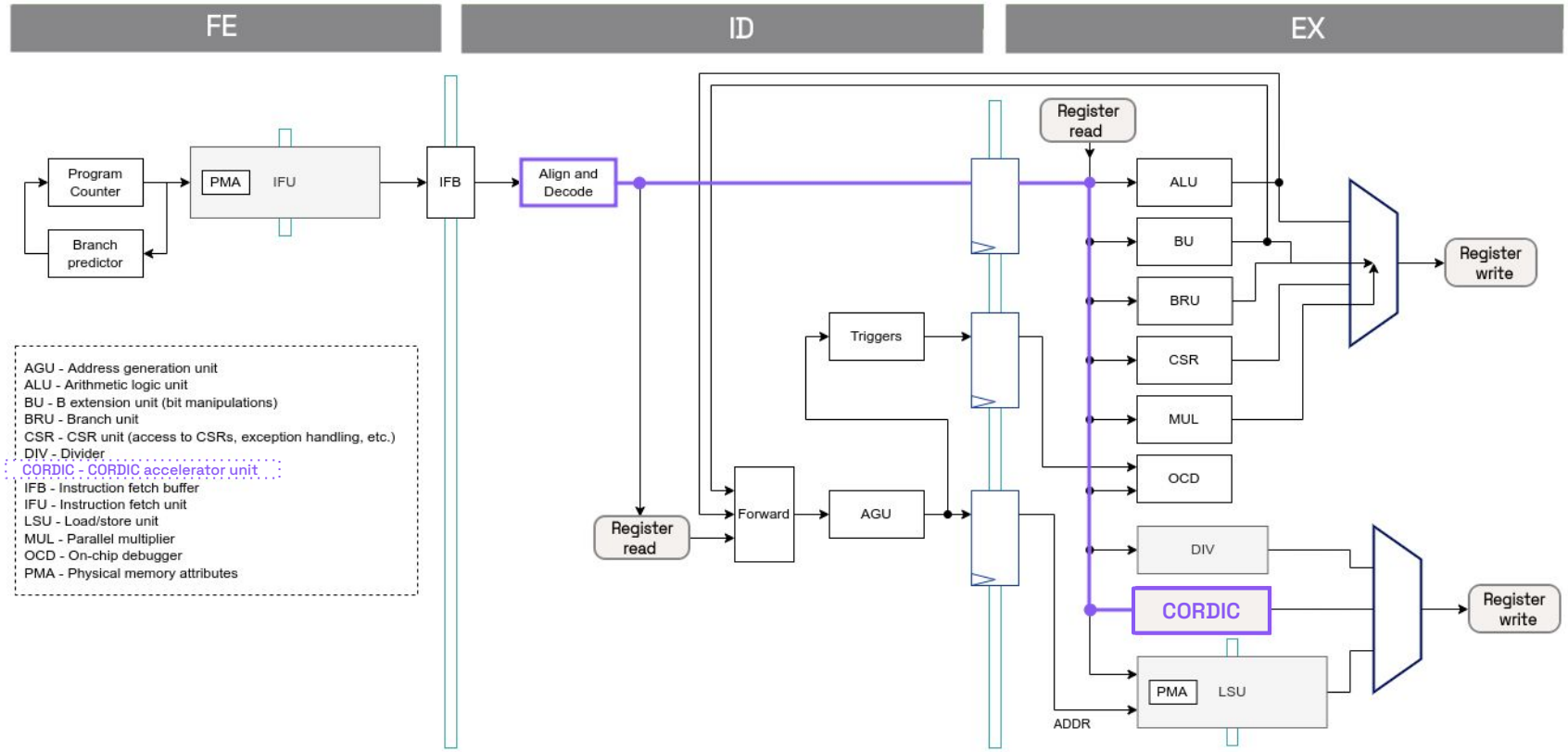


	L31	+FFT
Area, a.u. @50MHz	100%	**124.3%**
F_{max}	650	600
Area, a.u. @F_{max}	130.2%	128.3%
Performance scale	1x	0.92x

The latter shows how the performance gain should be scaled to account for the lower operation frequency of the customized core:

0.92/6.4% = **14.4x** - the net gain

Microarchitecture: CORDIC module



- Computed sine and cosine values consist of 16-bits each
- Only one 32-bits register output
- Sine and cosine computed values are concatenated in this output register.