Karlsruhe Institute of Technology

Lab: Designing Embedded Application-Specific Processors

CES – Chair for Embedded Systems

Majd Eddin Mansour Paul Georg Wagner

Supervisors: Sajjad Hussain
Dr. Hussam Amrouch

Outline

ADPCM Algorithm (pseudo code)

```
static int indexTable[16] = {
    -1, -1, -1, -1, 2, 4, 6, 8,
    -1, -1, -1, -1, 2, 4, 6, 8,
};

static int stepsizeTable[89] = {
    7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
    19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
    50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
    130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
    337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
    876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
    2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
    5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
    15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};
```

ADPCM Algorithm (pseudo code)

```
int adpcm_decoder(unsigned char* indata, int len) {
        for ( ; len > 0 ; len-- ) {

            /* Step 4 - difference and new predicted value */
            vpdiff = step >> 3;ble[index];
            /* Step 5 - clamp output value */
            if ( valpred > 32767 )-= step>>1;
             valpred = 32767;ff += step>>2;
            else if ( valpred < -32768 )> 4) & 0xf;
             valpred = -32768;e sign and magnitude */
             valpred -= vpdiff;ferstep;
            elsea = delta & 7;
             valpred += vpdiff;

} }
```

Simulating ADPCM

- Application compiled with **gcc** and executed

- Converted to assembly:
  - To understand the instructions used for the decoder
  - Easier to replace group of instructions with new instructions
  - Very helpful in area optimizing to find out the unneeded components

- Simulated with **dlxsim** and **ModelSim**
  - Using the unmodified CPU

- The output correctness is verified by comparing results

Implementing ADPCM

- CPU with App loaded to the FPGA and simulated
  - Bitrate in dlx_Toplevel.vhd set to 96 kHz audio
  - A bitstream is generated using Xilinx ISE for dlx_basis CPU
  - The bitstream is initialized with the application and uploaded

- As a result we could hear the decoded output by the FPGA
  - Very slow playback at 25, 50, 75 MHz
  - Yet slow playback at 100 MHz (deep sound)

- Sound is encoded with 192 kHz instead of 96 kHz
  - We resampled the provided audio data by decoding the sound and re-encoding it with 96 kHz

Preparing a Benchmark

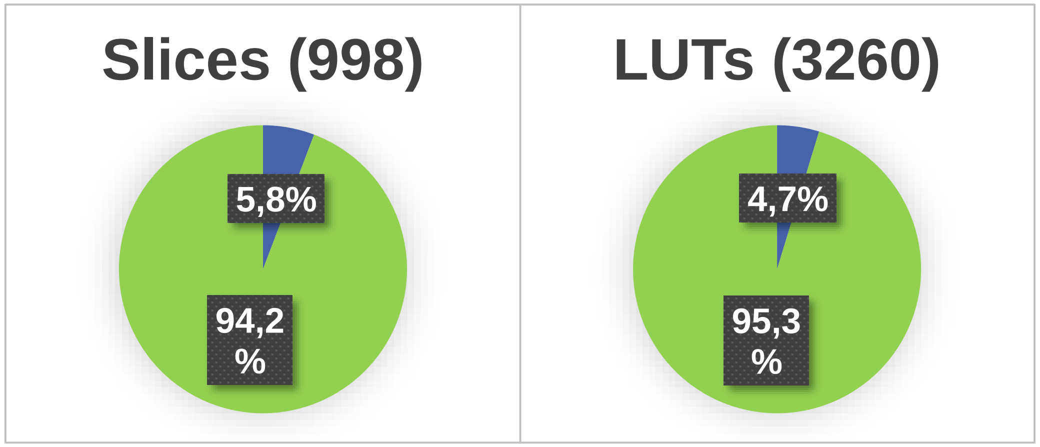
- The output and the function calling are removed from adpcm.c
  - The provided adpcmDataStereo_MINI.h is linked
  - Converted to Assembly

- Simulated with ModelSim to record the count of cycles
  - (CLK_HALF_PERIOD = 10ns // 50 MHz)

- Area and timing reports are generated the using Xilinx ISE
  - Synthesized VHDL files + benchmark framework are loaded
- Power benchmark with Xilinx ISE
  - After creating VCD files using ModelSim

Benchmark Results (Basis CPU)

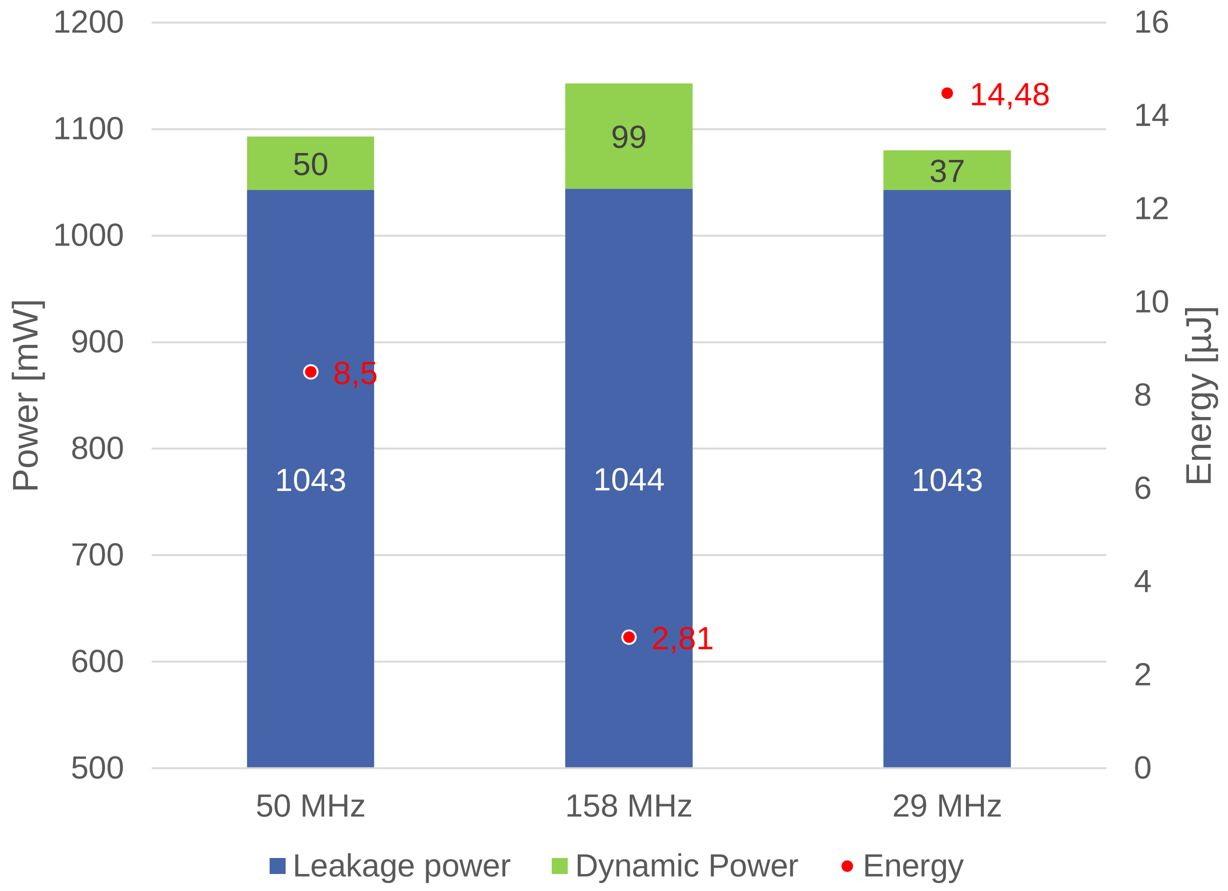
Cycles **388.814**



Critical path [ns] **6,316** $\Rightarrow$ $\frac{1}{6,316} \approx 158\ Mhz$ (Maximum)

$\frac{388.814}{1300} \approx 300\ \frac{cycle}{sample}$ $\Rightarrow$ $300 \cdot 96000 \approx 29\ Mhz$ (Minimum)

Benchmark Results (Basis CPU)



Execution Time [ms]	7,78	2,46	13,41

PERFORMANCE OPTIMIZATION

Performance Optimization

- Goal: Minimizing the execution time
  - $t_{exe}\ [\mathrm{ns}] = c\ [\mathrm{cycles}] \cdot t_{cyc}\ [\mathrm{ns}]$
- Reduce number of cycles
  - Optimize existing code
  - Create suitable hardware components/instructions
  - Enhance data parallelism
- Convert project to assembly to allow finer control
  - Main point of optimization: for-loop

Performance Optimization 1

- Outsourcing index and step calculation to hardware
  - New VHDL component
  - Saves two address calculations/memory accesses
  - Saves two if-statements

```
step = stepsizeTable[index];
index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;
```

⇨

```
lstep %r5, %r8
addidx %r7, %r8, %r3
```

Performance Optimization 1

- VHDL implementation of *lstep* and *addidx* instruction
  - Input: sample, old index
  - Output: step, new index

```vhdl
process (clock, reset, enb)
begin
  if (enb = '1') then
    step_out <= std_logic_vector(stepsizeTable(unsigned(index_in)));
    index := std_logic_vector(indexTable(unsigned(data_in))) +
                               std_logic_vector(unsigned(index_in));
    if (index < lower) then
      index := lower;
    elsif (index > upper) then
      index := upper;
    end if;
    index_out <= index;
  end if;
end process;
```

Performance Optimization 1

- Loading two samples at once
  - Load one byte
  - Store two samples in two registers
  - No new VHDL component required
- Also process two samples in one loop iteration

```c
/* Step 1 - get the delta value */
if ( bufferstep ) {
        delta = inputbuffer & 0xf;
} else {
       inputbuffer = *indata++;
       delta = (inputbuffer >> 4) & 0xf;
}
bufferstep = !bufferstep;
```

$\Rightarrow$

```
lbh %r3, %r4, %r1
addui %r1, %r1, $1
```

Performance Optimization 1

- Outsource PCM decoding to hardware
  - Input: Sample (r3), Step (r5), Previously decoded sample (r10)
  - Output: Decoded sample (r9)

```
vpdiff = step >> 3;
if ( delta & 4 ) vpdiff += step;
if ( delta & 2 ) vpdiff += step>>1;
if ( delta & 1 ) vpdiff += step>>2;

if ( sign )
 valpred -= vpdiff;
else
 valpred += vpdiff;

if ( valpred > 32767 )
 valpred = 32767;
else if ( valpred < -32768 )
 valpred = -32768;
```

$\Rightarrow$

```
pcmdec %r9, %r5, %r3, %r10
```

Performance Optimization 1

- VHDL implementation of *pcmdec* instruction

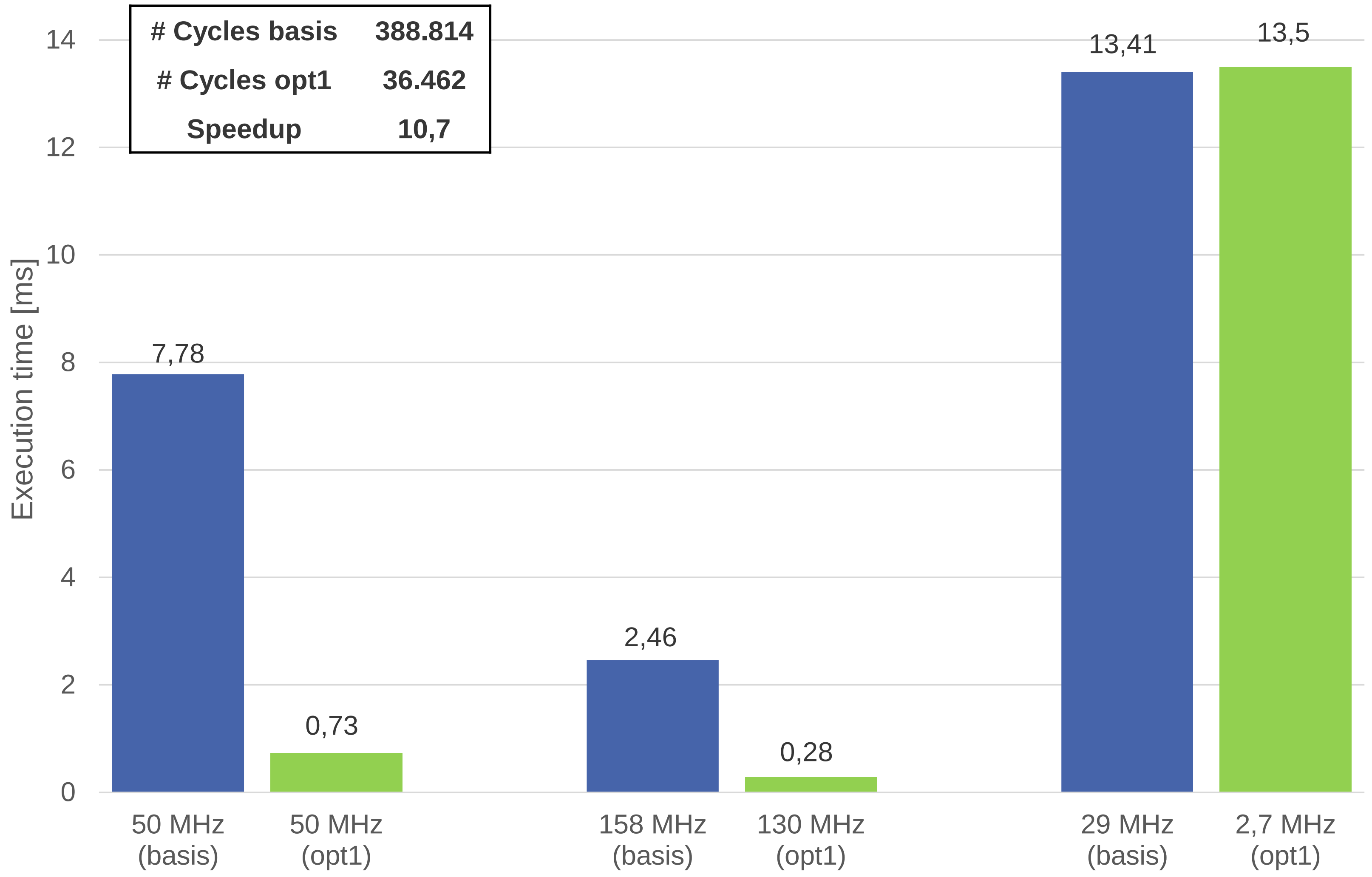
```
vpdiff := signed("000" & step(31 downto 3));
if (delta(2) ='1') then
        vpdiff := vpdiff + signed(step);
end if;
if (delta(1) ='1') then
        vpdiff := vpdiff + signed("0" & step(31 downto 1));
end if;
if (delta(0) ='1') then
        vpdiff := vpdiff + signed("00" & step(31 downto 2));
end if;

if(delta(3)='1') then
        valpred_out <= signed(valpred_in) - vpdiff;
else
        valpred_out <= signed(valpred_in) + vpdiff;
end if;
```

Performance Optimization 1

- Optimized code inside the for-loop

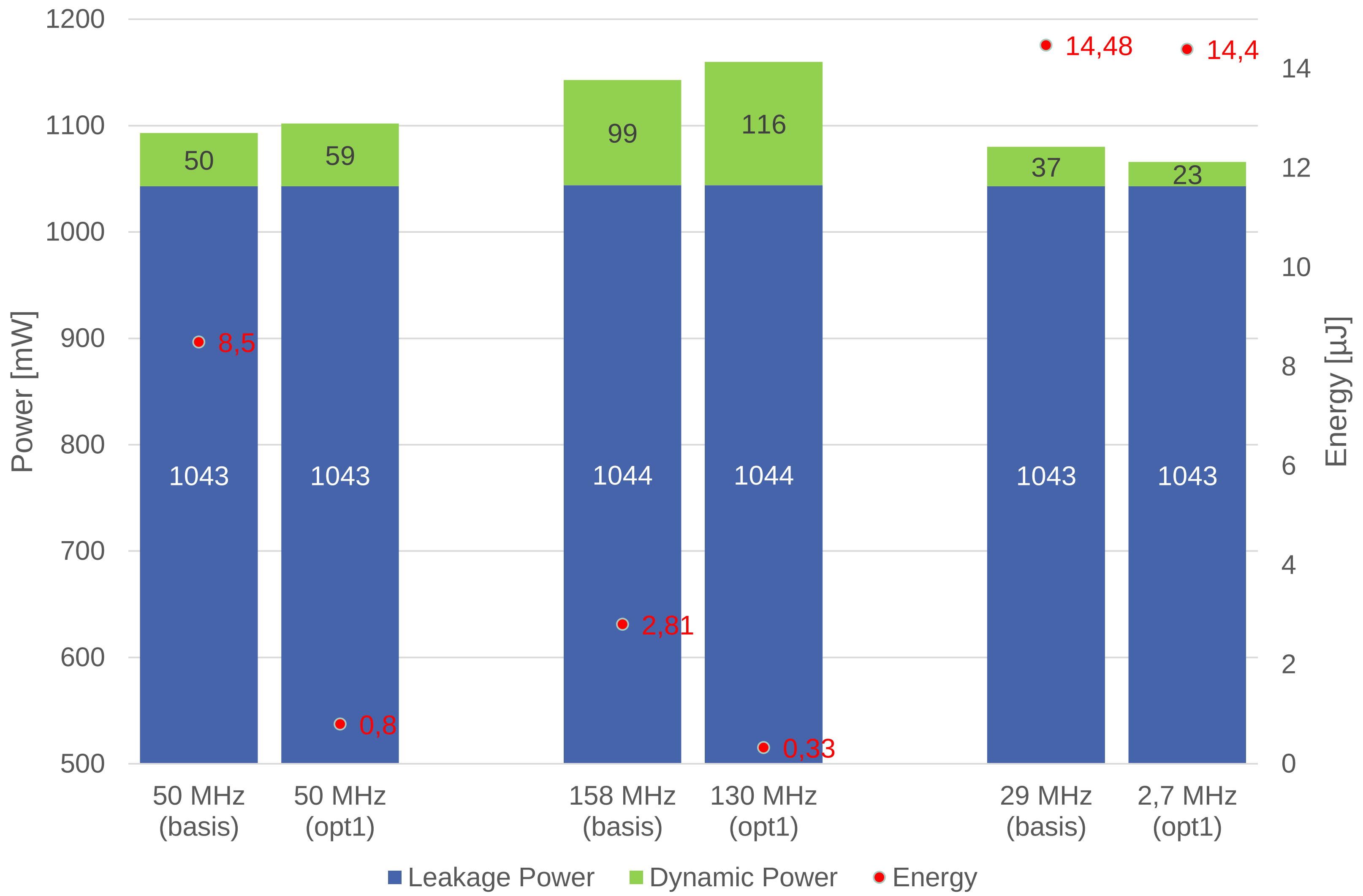
```
; /* Load two samples into two registers */
    lbh %r3, %r4, %r1
    addui %r1, %r1, $1

; /* Update index and step */
    lstep %r5, %r8
    addidx %r7, %r8, %r3
    lstep %r6, %r7
    addidx %r8, %r7, %r4

; /* Decode two samples */
    pcmdec %r9, %r5, %r3, %r10
    pcmdec %r10, %r6, %r4, %r9
```

Performance Optimization 1: Results – Execution Time



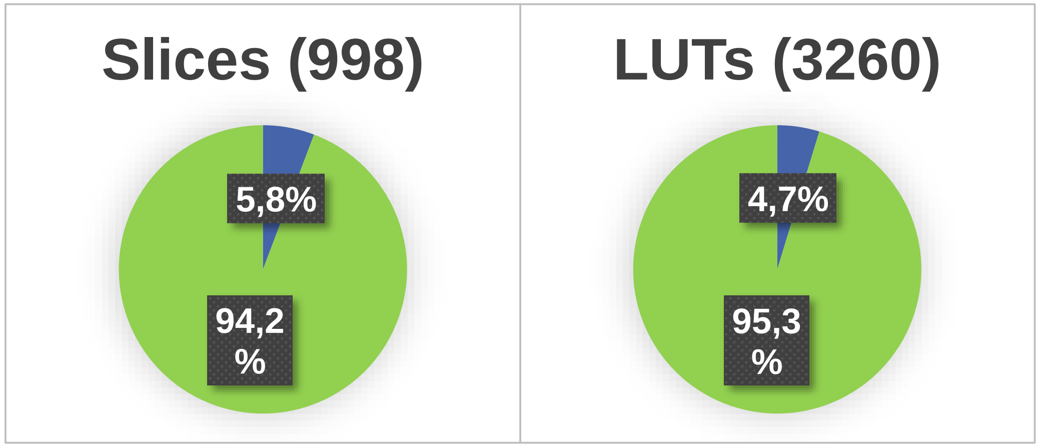
# Cycles basis	**388.814**
# Cycles opt1	**36.462**
Speedup	**10,7**

Performance Optimization 1: Results – Power & Energy

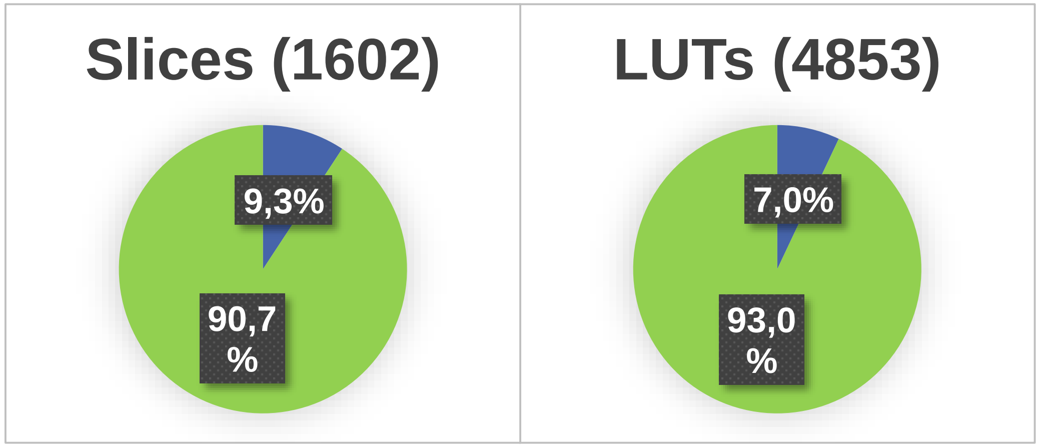


Performance Optimization 1: Results – Area

- basis:



- opt1:

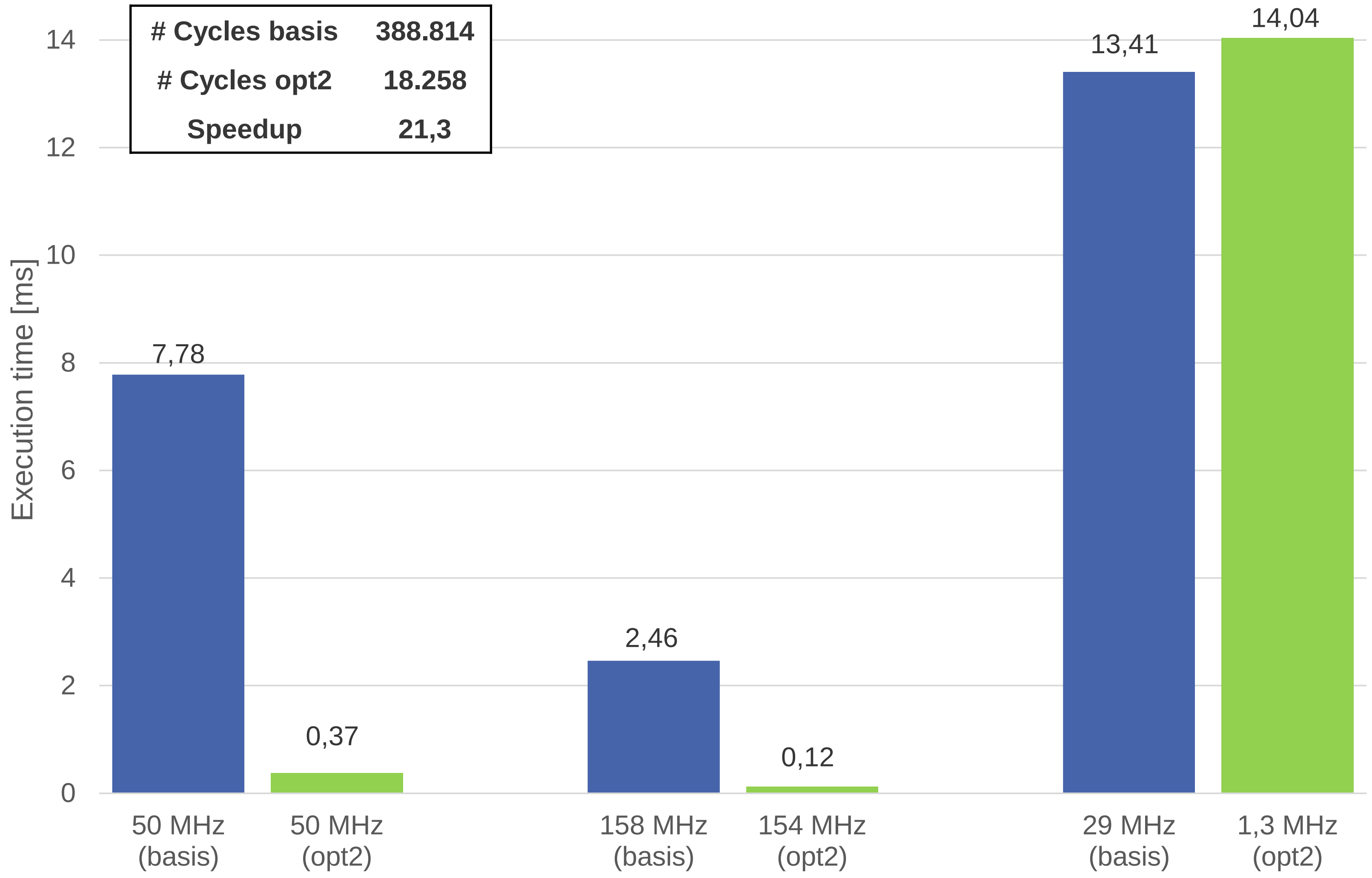


Performance Optimization 2

- Reducing number of components to just one
  - Step calculation and decoding of two samples is done in a single component
- Advantages
  - Reduced number of instructions
  - Code inside the for-loop becomes very simple

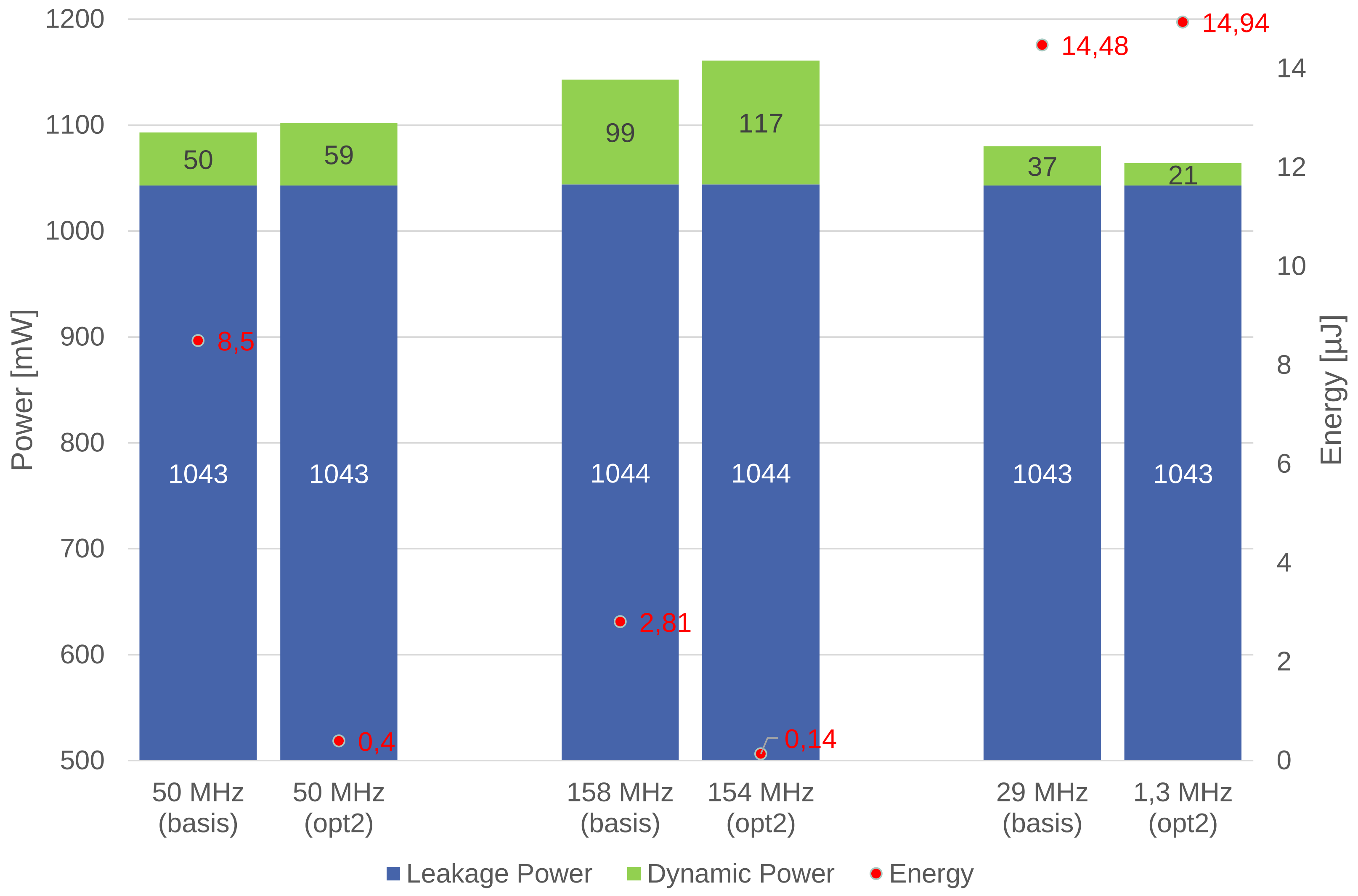
```
; /* Load two samples into two registers */
    lbh %r3, %r4, %r1
    addui %r1, %r1, $1

; /* Decode samples */
    pcm %r3, %r4, %r5, %r6
```

Performance Optimization 2: Results – Execution Time



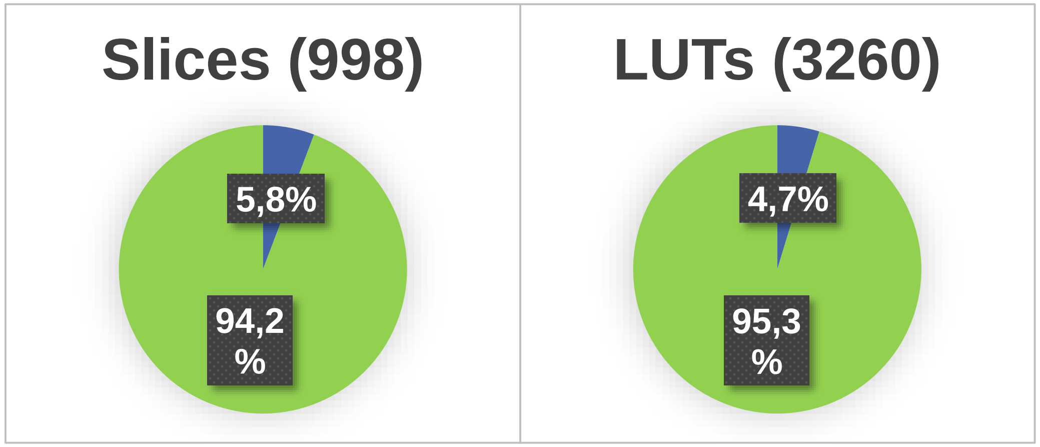
# Cycles basis	**388.814**
# Cycles opt2	**18.258**
Speedup	**21,3**

Performance Optimization 2: Results – Power & Energy

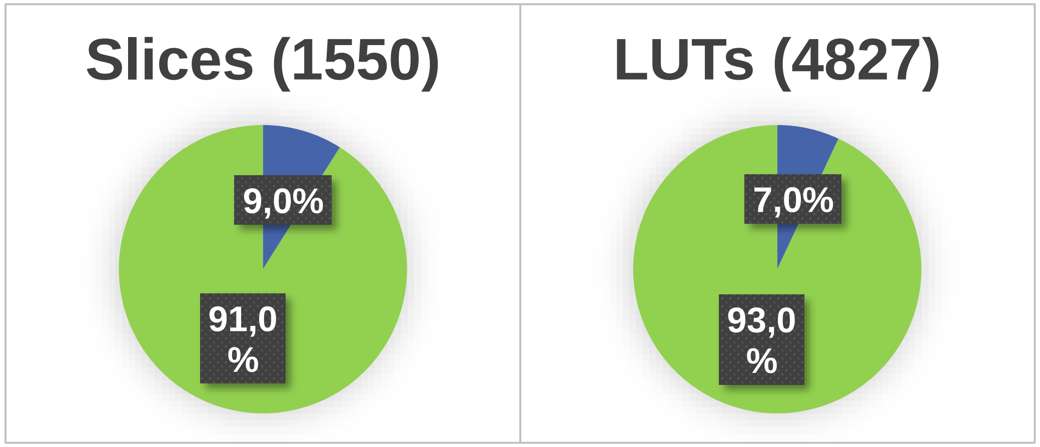


Performance Optimization 2: Results – Area

- basis:



- opt2:

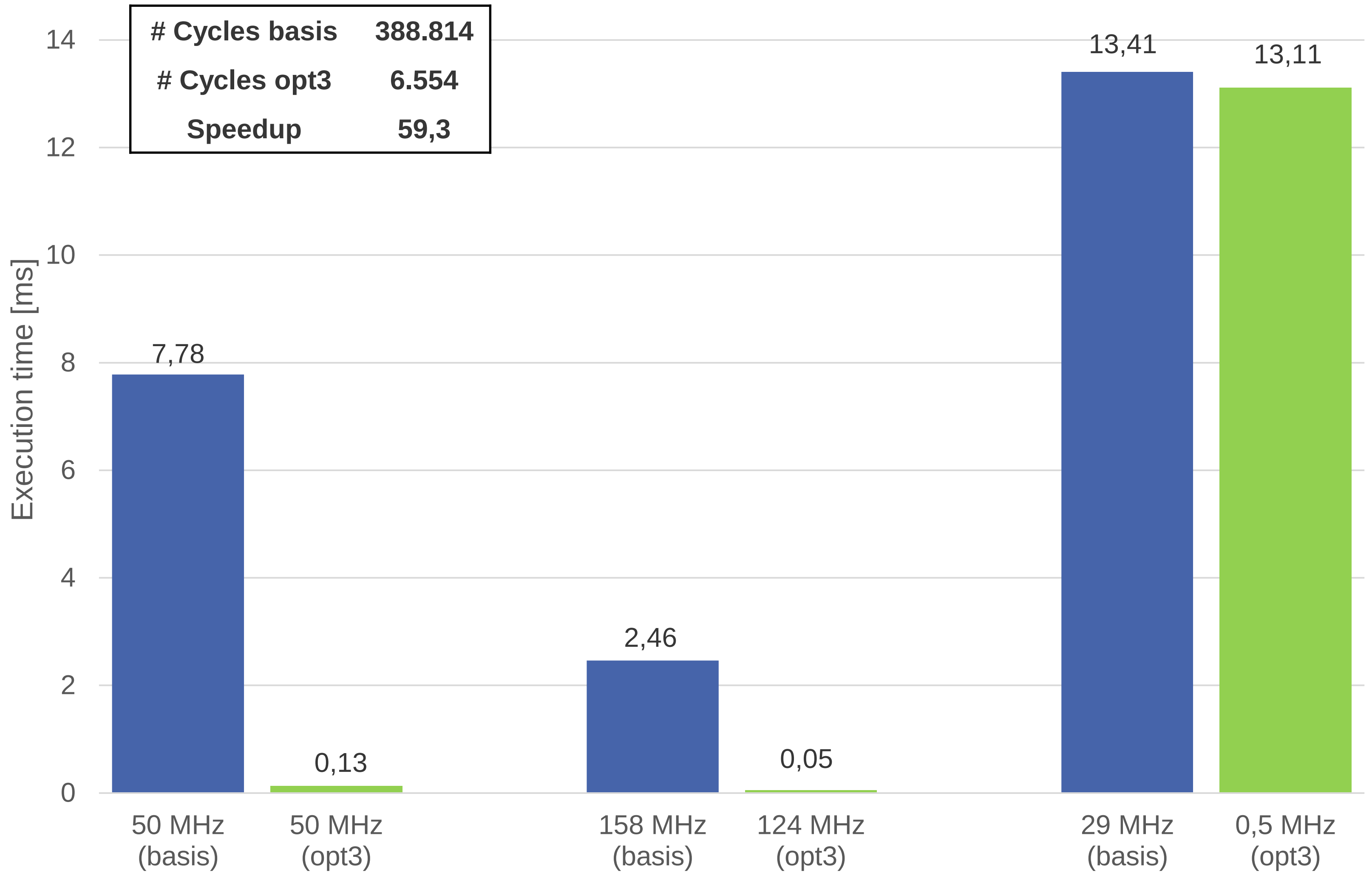


Performance Optimization 3

- Increase the number of samples decoded in one step
  - Currently: two output registers holding two decoded samples
  - In each register two 16-bit output samples can be stored
  - Goal: decode four samples with one instruction
- New load instruction loads four input samples
  - Also increases address register

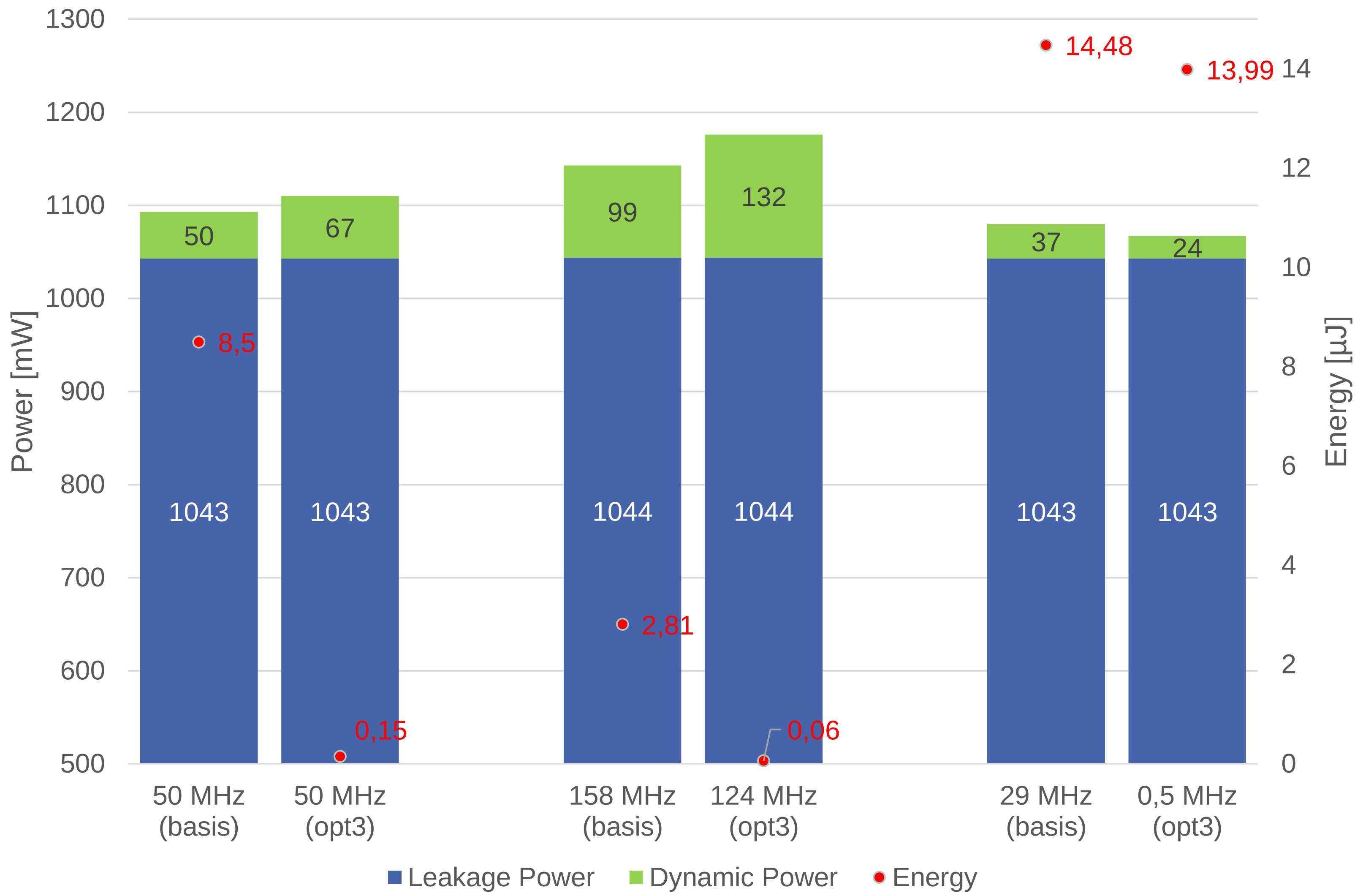
```
; /* Load four samples into two registers */
   lhh %r3, %r4, %r1

; /* Decode samples */
   pcm %r3, %r4, %r5, %r6
```

Performance Optimization 3: Results – Execution Time



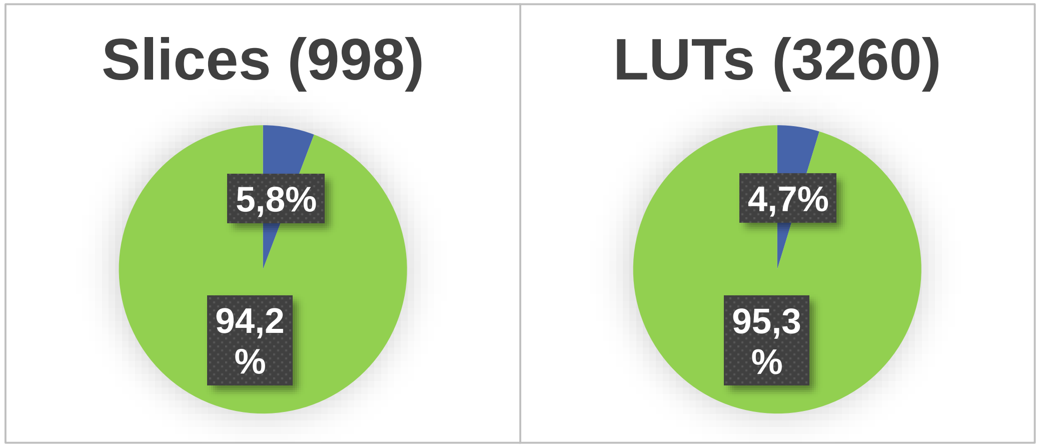
# Cycles basis	**388.814**
# Cycles opt3	**6.554**
Speedup	**59,3**

Performance Optimization 3: Results – Power & Energy

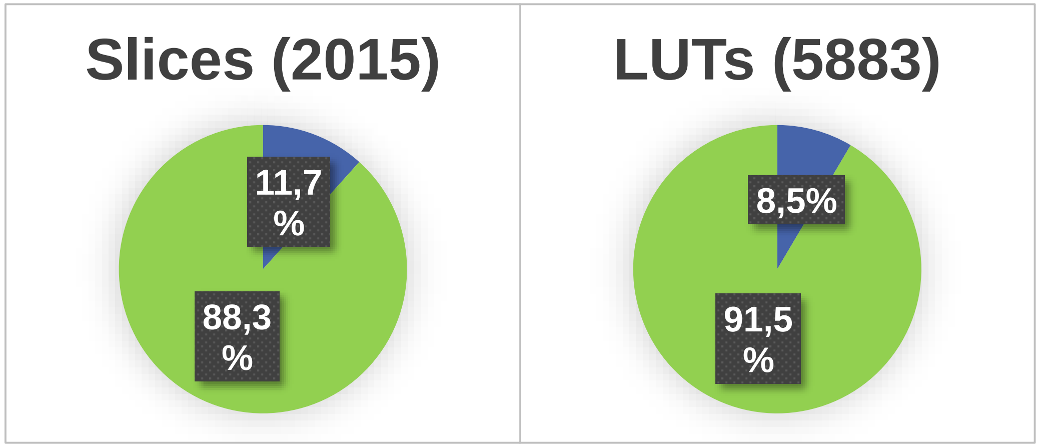


Performance Optimization 3: Results – Area

- basis:



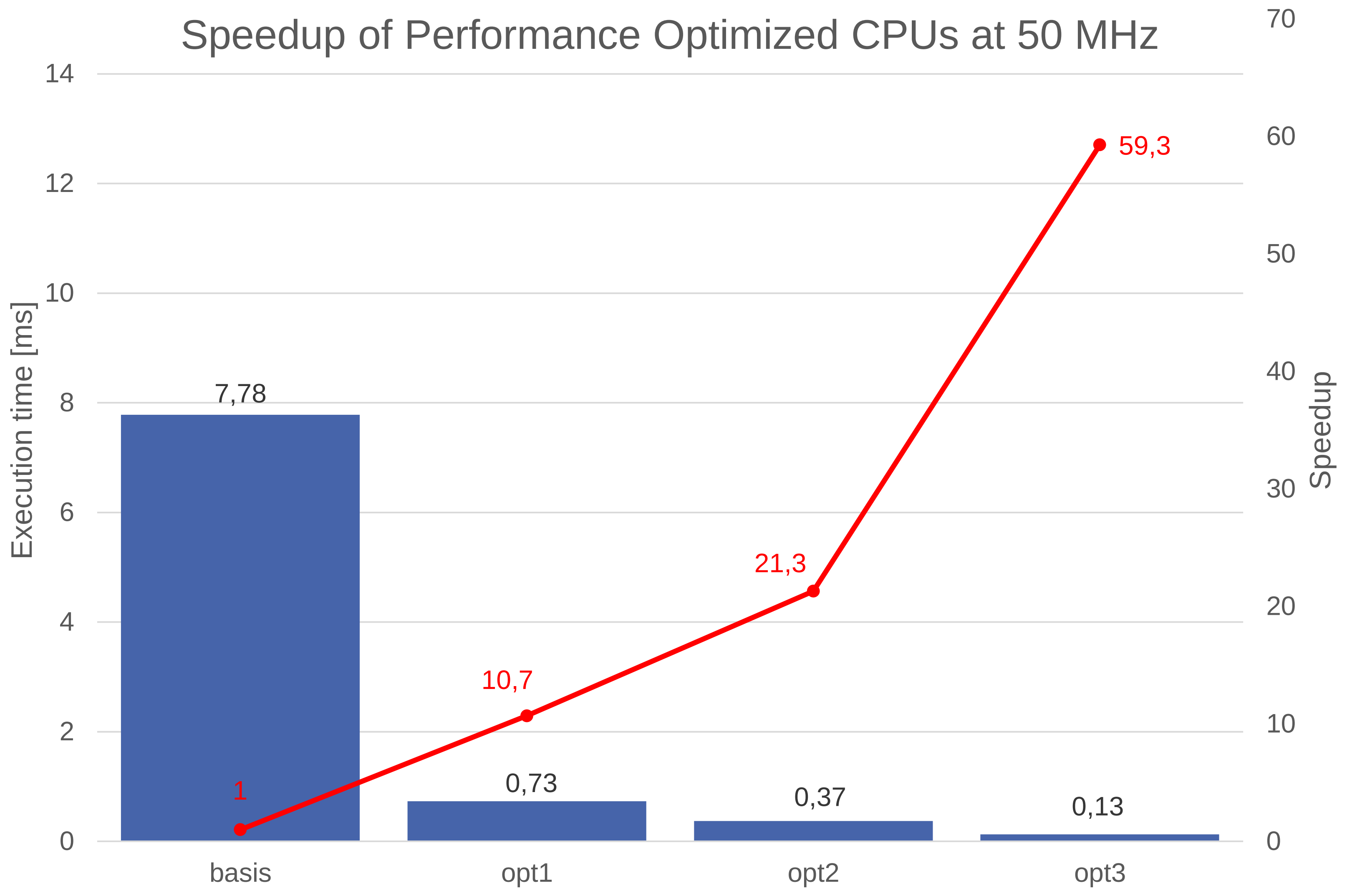
- opt3:



Performance Optimization: Problems

- Constrained number of operand registers
  - With 32 bit long instruction words, 6 bit opcode and 5 bit register addresses, only 5 register operands are supported
  - Used operand registers for input and output simultaneously
  - Unproblematic due to pipeline structure
- Standard register file only supports a maximum of 4 read and 4 write ports
  - Data parallelism requires storing of multiple samples in one register
  - Theoretical limit: 7 output samples written in one instruction
  - The third optimized CPU supports a maximum of 4 output samples

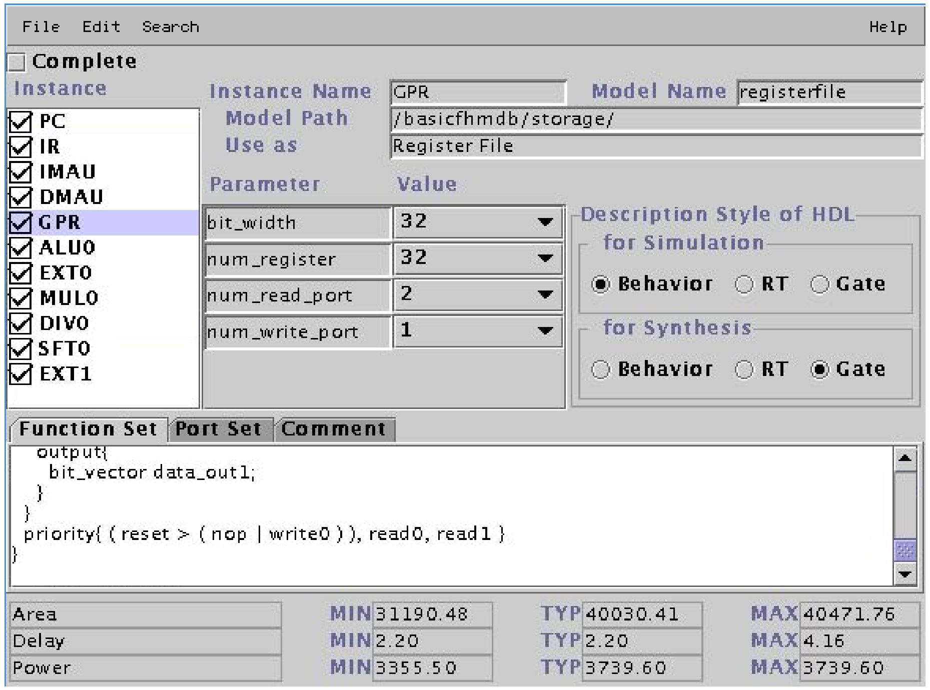
Performance Optimization 3: Overall Speedup



AREA OPTIMIZATION

Area Optimization

- Goal: Reducing the needed slices and LUTs
- Method: Using as few components as possible
  - By simply removing as many hardware resources as possible from the CPU description



☐ Complete

Instance: ✓ PC, ✓ IR, ✓ IMAU, ✓ DMAU, ✓ GPR, ✓ ALU0, ✓ EXT0, ✓ MUL0, ✓ DIV0, ✓ SFT0, ✓ EXT1

Instance Name: GPR
Model Name: registerfile
Model Path: /basicfhmdb/storage/
Use as: Register File

Parameter	Value
bit_width	32
num_register	32
num_read_port	2
num_write_port	1

Description Style of HDL
for Simulation: [✓ Behavior] RT Gate
for Synthesis: Behavior RT [✓ Gate]

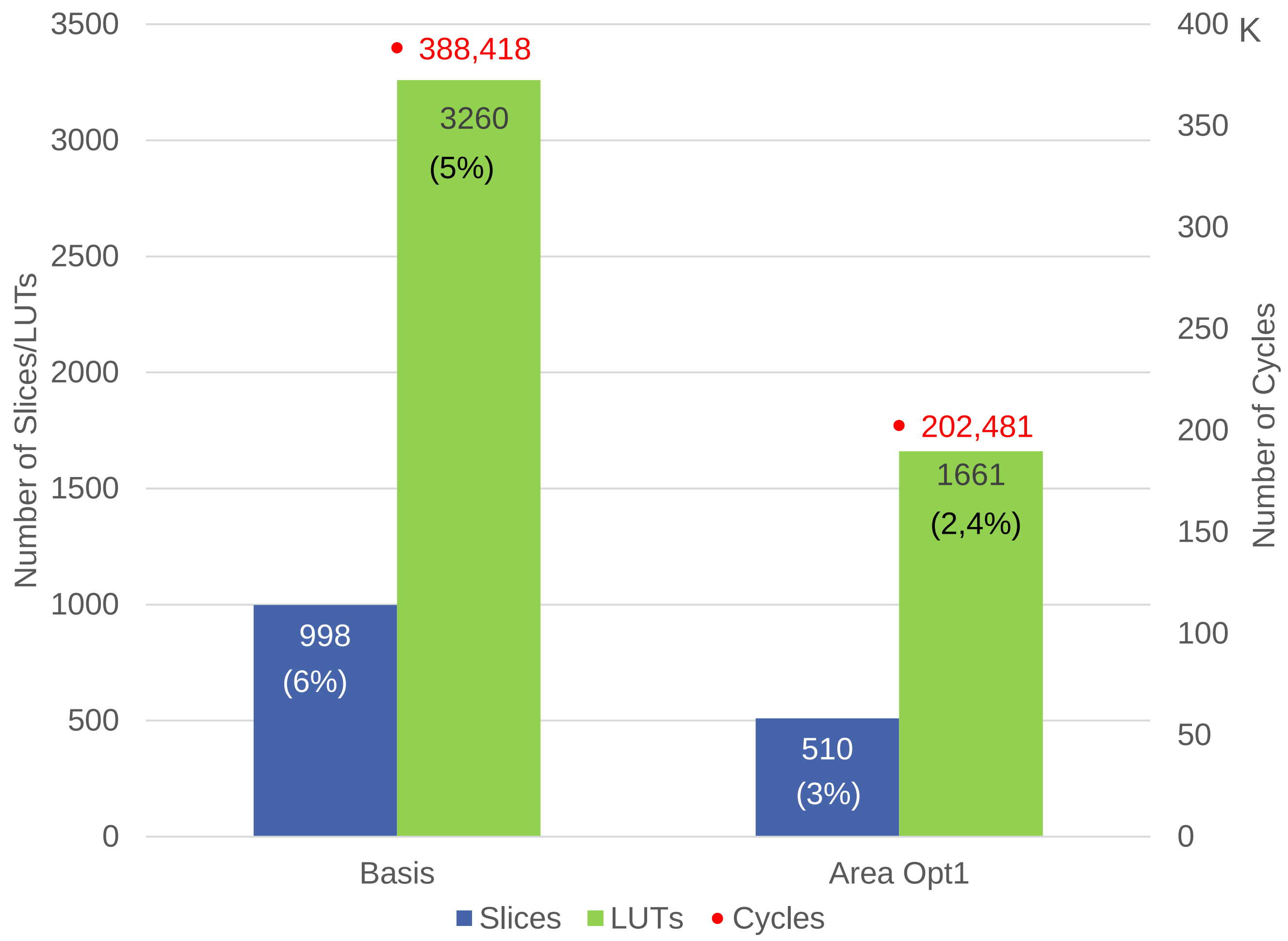
```
  output{
    bit_vector data_out1;
  }
 }
 priority( ( reset > ( nop | write0 ) ), read0, read1 )
}
```

Area	MIN	31190.48	TYP	40030.41	MAX	40471.76
Delay	MIN	2.20	TYP	2.20	MAX	4.16
Power	MIN	3355.50	TYP	3739.60	MAX	3739.60

Area Optimization 1

- The MUL0 and DIV0 resources are removed from the CPU
    - Along with the instructions that use these resources
- The size of the register file is reduced from 32 to only 16 registers.
    - Registers are addressed using only 4 bits instead of 5 bits.
        - The micro operations which use the register address is modified
        - In those macros we simply removed the most significant bit before calling GPR.read() or GPR.write() with a register address
    - The dedicated link register, and hence the jal, jalr, and jr instructions are removed

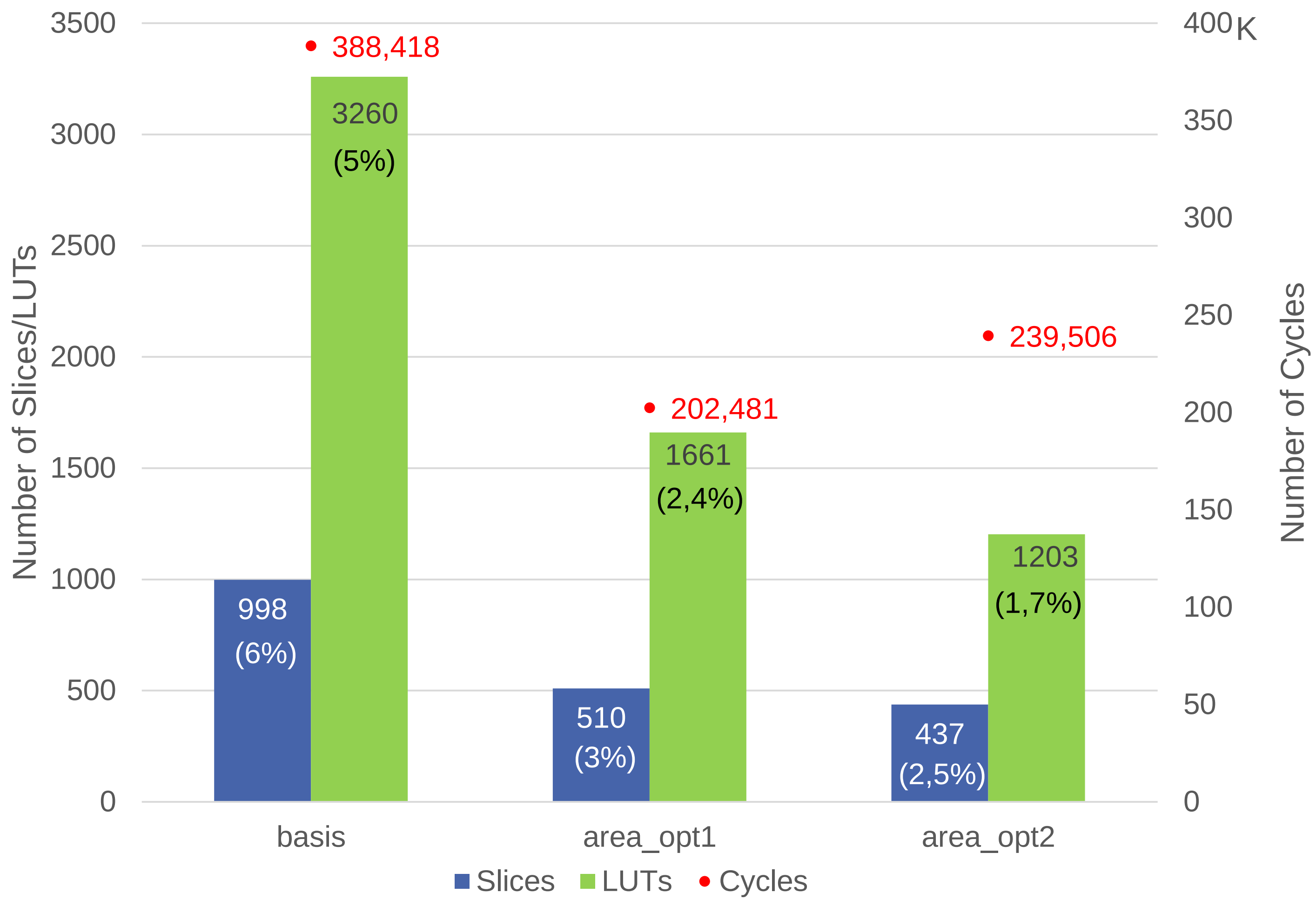
Area Optimization 1: Results - Slices, LUTs



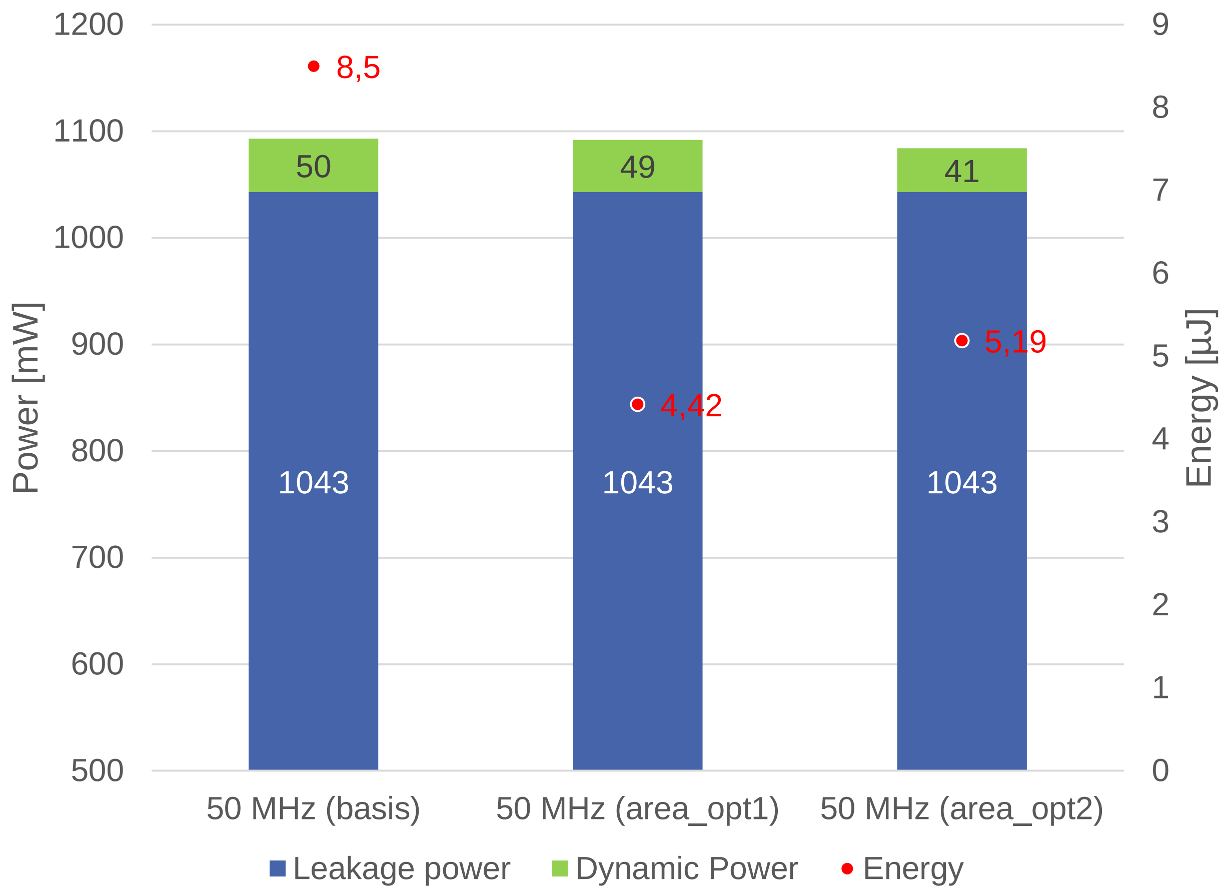
Area Optimization 2

- Yet more registers are removed from the register file
- Registers are generalized and reused as much as possible
  - For example, in each iteration the base address of the input sample array has to be reloaded
- The code is re-implemented in order to use only seven registers
- This increases the number of required instruction cycles, and hence the execution time

Area Optimization 2: Results - Slices, LUTs



Area Optimization: Results – Power & Energy



Execution Time [ms]	7,78	4,05	4,79

Area Optimization: Problems

- As we used the registers (from r0 to r7), r0 could not be used as a general purpose register.
  - Could be hardware coded

- After re-implementing the code for the second optmization, it was not feasable to debug with dlxsim
  - Dlxsim shows correct results, but ModelSim does not

Conclusion

- Optimizing performance can be applied on costs of either power and energy or area
  - In our project the performance optimization caused a small loss of area

- Optimizing area can be applied on costs of performance, power or energy

Appendix: Benchmark Results

- Basis CPU

Critical path	Cycles	# Slices	% Slices	# LUTs	% LUTs
6,316 ns	388.814	998	6%	3260	5%

Frequency	Total Power	Leakage Power	Dynamic Power	Time	Energy
50 MHz	1093 mW	1043 mW	50 mW	7,78 ms	8,50 µJ
158 MHz	1143 mW	1044 mW	99 mW	2,46 ms	2,81 µJ
29 MHz	1080 mW	1043 mW	37 mW	13,41 ms	14,48 µJ

- Performance Optimized Version 1

Critical path	Cycles	# Slices	% Slices	# LUTs	% LUTs
7,679 ns	36.462	1602	9%	4853	7%

Frequency	Total Power	Leakage Power	Dynamic Power	Time	Energy
50 MHz	1102 mW	1043 mW	59 mW	0,73 ms	0,80 µJ
130 MHz	1160 mW	1044 mW	116 mW	0,28 ms	0,33 µJ
2,7 MHz	1066 mW	1043 mW	23 mW	13,50 ms	14,40 µJ

Appendix: Benchmark Results

- Performance Optimized Version 2

Critical path	Cycles	# Slices	% Slices	# LUTs	% LUTs
6,492 ns	18.258	1550	9%	4827	7%

Frequency	Total Power	Leakage Power	Dynamic Power	Time	Energy
50 MHz	1102 mW	1043 mW	59 mW	0,37 ms	0,40 µJ
154 MHz	1161 mW	1044 mW	117 mW	0,12 ms	0,14 µJ
1,3 MHz	1064 mW	1043 mW	21 mW	14,04 ms	14,94 µJ

- Performance Optimized Version 3

Critical path	Cycles	# Slices	% Slices	# LUTs	% LUTs
8,026 ns	6554	2015	12%	5883	9%

Frequency	Total Power	Leakage Power	Dynamic Power	Time	Energy
50 MHz	1110 mW	1043 mW	67 mW	0,13 ms	0,15 µJ
154 MHz	1176 mW	1044 mW	132 mW	0,05 ms	0,06 µJ
0,5 MHz	1067 mW	1043 mW	24 mW	13,11 ms	13,99 µJ

Appendix: Benchmark Results

- Area Optimized Version 1

Critical path	Cycles	# Slices	% Slices	# LUTs	% LUTs
7,207 ns	202481	510	3,0%	1661	2,4%

Frequency	Total Power	Leakage Power	Dynamic Power	Time	Energy
50 MHz	1092 mW	1043 mW	49 mW	4,05 ms	4,42 µJ
138 MHz	1143 mW	1044 mW	99 mW	1,47 ms	1,68 µJ
15 MHz	1068 mW	1043 mW	25 mW	13,50 ms	14,42 µJ

- Area Optimized Version 2

Critical path	Cycles	# Slices	% Slices	# LUTs	% LUTs
6,167 ns	239506	437	2,5%	1203	1,7%

Frequency	Total Power	Leakage Power	Dynamic Power	Time	Energy
50 MHz	1084 mW	1043 mW	41 mW	4,79 ms	5,19 µJ
162 MHz	1154 mW	1044 mW	110 mW	1,48 ms	1,71 µJ
15 MHz	1065 mW	1043 mW	22 mW	13,31 ms	14,17 µJ