ASIP Laboratory - Session 8

Paul Georg Wagner Majd Mansour

July 23, 2017

For this session a ADPCM audio decoding algorithm *adpcm.c* that decodes audio samples and plays them on the FPGA board is given. The task for this session is to create optimized CPUs for this project and minimize certain design parameters like execution time, power consumption or area requirements of the CPUs.

Exercise 2: Simulating the Application

As a first step we compiled the provided *adpcm.c* project using the COSY compiler created in the last session along with optimization level -O3. For this the modified CPU from the last session was used. Then we simulated the project using gcc, dlxsim as well as ModelSim, and verified that the output for all three versions is the same. Afterwards we repeated the simulation with the unmodified CPU from session 3 (dlx_basis) and again verified that the output for all three versions is correct.

As a next step, the project should be loaded and executed to the FPGA board. For this we first created a new Xilinx ISE project for the dlx_basis CPU version and generated a bitstream for the project. Then we compiled and uploaded the project using *make sim*, *make fpga* and *make upload*. Afterwards the audio output given by the FPGA board showed a slow playback, which resulted in a deep sound. The sound was the same when the FPGA board was run with 100 MHz and 80 MHz, and it gets even deeper when the board frequency is set to less than 66 MHz.

The deep sound even at the highest board frequency of 100 MHz was the result of a wrong sample rate setting in *dlx_ Toplevel.vhd* (48 kHz instead of 96 kHz). However, when we fixed the sample rate and uploaded the new bitstream to the FPGA board, the sound was significantly higher, but still too deep. This is because the provided audio data actually is sampled with a sample rate of 192 kHz instead of 96 kHz, as specified in the task sheet. Once we resampled the provided audio data with 96 kHz and updated the project, the sound was finally correct.

After the project was validated also on the FPGA board, the benchmark of the unoptimized *adpcm.c* project was performed. For this we first striped the *adpcm.c* file of all output, including the audio output calls and linked the provided *adpcmDataStereo_MINI.h* file in order to use the example input array instead of the actual audio data for the benchmark. Then we simulated the resulting file with ModelSim and determined the total number of instruction cycles as 388.814 cycles. Afterwards we created a new Xilinx ISE project, loaded the synthesized VHDL files of the CPU along with the benchmark framework and generated the area and timing reports. The unmodified dlx_basis CPU requires a total of 998 (6 %) of the available FPGA slices and 3260 (5 %) of the available lookup tables. The critical path through the CPU takes 6,3 nanoseconds, which results in a maximum possible CPU frequency of $\frac{1}{6{,}3\text{ ns}} \approx 158$ MHz.

The power estimation for the various projects should be performed at a fixed clock frequency of 50 MHz, the maximum possible frequency as well as the lowest possible frequency at which the code is still capable of decoding at least the required 96.000 samples per second. For the unmodified CPU version the lowest possible frequency can be calculated using the number

of cycles that is required for running the code, as well as the number of samples that the code processes. The used *adpcmDataStereo_MINI.h* data contains 1300 samples that have been decoded using 388.814 instruction cycles. This means that $\frac{388.814 \text{ cycles}}{1300 \text{ samples}} \approx 300 \frac{\text{cycles}}{\text{sample}}$ cycles are required to decode one single sample. Multiplying this value with the target sample rate gives us a minimum frequency of $300 \frac{\text{cycles}}{\text{sample}} \cdot 96.000 \frac{\text{sample}}{\text{s}} \approx 29$ MHz.

In order to generate exact power estimation reports, we first created value change dump (VCD) files for all three interesting CPU frequencies using ModelSim. Then we created another Xilinx ISE project and started the power analyzer tool. Unsurprisingly, the dynamic power requirement was highest for the 158 MHz version (99 mW), while the 28 MHz version required the least power (37 mW). Because the used FPGA does not support power gating, the leakage power is at a constant 1043 mW for all versions. Even though the 158 MHz version requires the most dynamic and total power, with only 2,81 μJ it is somewhat more energy efficient than the 28 MHz version (14,48 μJ). This is because of the much lower execution time that the 158 MHz version has (2,46 ms instead of 13,41 ms).

The results for the benchmark of the baseline version is displayed in table 1.

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

Table 1: Benchmark results for dlx_basis

Exercise 5: Extending the CPU

The main task in this session was to create two optimized CPU versions for this problem that minimizes certain design parameters. We decided to create optimized versions or this project with regard to area and performance. Having CPUs with optimized area characteristic would allow for the use of smaller FPGA boards with less leakage power. Having CPUs that are optimized for performance allows for decoding at a much higher sample rate, which then supports higher audio quality.

Optimizing Performance

There are several ways of creating a CPU that optimizes the performance of the PCM decoding algorithm. One possibility is to add CPU instructions and components that allow for parallel processing of multiple input samples. Since the input samples are only 4 bit long, we could fit 8 input samples in a single 32 bit general purpose register. Having instructions that operate on all 8 samples at once could naturally speed up the program by several factors. However, implementing a parallel processing in that manner requires to deal with data dependencies between the single processing steps. In this case, the decoding of a input sample depends in part on the decoding of the previous samples, which is why this method might run into troubles.

This is why we decided to first try to optimize the performance of the existing PCM decoding algorithm by reducing the number required CPU cycles as much as possible. Since the PCM decoding algorithm mainly consists of one single for-loop, which loads the new input sample, performs a certain amount of add and shift operations using a constant index and step table, and finally outputs the decoded sample, the best way to reduce the CPU cycles is to make the for-loop as small as possible. Hence we created new FHM components that implement certain amounts of work from a single decoding step. By replacing a bunch of

instructions inside the for-loop with just one custom instruction, the total amount of necessary instruction cycles can be significantly reduced. On the other hand, adding new resources to the CPU complicates and prolongs the critical path, which typically results in a smaller maximum reachable frequency. Hence the complication of the CPU by adding new components and the saving of instruction cycles have to be properly balanced.

Implementing Version 1

As first step, we moved the calculation of the step and index value for one decoding step into a new component. The original code stores two tables in memory, whose values have to be loaded into registers each iteration. By moving the tables, as well as the calculation steps into a new hardware resource, we can save several instructions inside the for-loop. More concretely, we added two new processor instructions `lstep %r1, %r2`, which calculates the equivalent of the previous `step = stepsizeTable[index];` instruction, as well as the `addidx %r1, %r2` instruction, which replaces the `index += indexTable[delta];` command. The instructions are performed by two new hardware resources, which we implemented in VHDL and added as a FHM resource. The implementations are given in listings 4 and 5 in the appendix.

As the next step, we implemented a distinct component to clamp values. In the original code there are multiple occurrences of code that constrains a value into a certain range using if-else constructs. By extracting this behavior to a new component and adding a new `clmp %rd, %rs0, %rs1, %rs2` instruction, this overhead can be removed. The corresponding VHDL resource is given in listing 6.

Next we implemented a new command to load a byte from memory. Each input sample that is processed in one loop iteration is four bits long, so there are two input samples in each loaded byte. The original code uses an if-else construction to load a byte from the input array at every other loop iteration and uses a flag to determine which four bits should be used. In order to avoid this overhead we implemented a new `lbh %rd0, %rd1, %rs` instruction, which loads a byte from memory, splits it into two times four bits and stores the samples in two registers. This furthermore allows us to process both samples in one loop iteration, which saves half the required iterations. Changing the structure of the for-loop also requires pre-calculation of both the step and index value. However, with the added `lstep` and `addidx` instruction this does not introduce any more overhead. The `lbh` instruction can be implemented solely in ASIPMeister and does not require any more resources.

After this modification, the main instruction for solving the PCM decoding problem can be implemented. In the C code, the actual decoding step is performed using the current step, index and input sample value. These calculations only consist of adding and shifting values, which can be effectively outsourced into a distinct component. This saves a lot of instruction cycles, because with the original code the compiler adds three NOPs after each arithmetic or logic operation, whereas with a distinct PCM decode instruction, those overhead NOPs disappear. Hence we added a `pcmdec %rd, %rs0, %rs1, %rs2` operation, which takes the current input sample, the current step value and the previous result value as input and calculates the next output sample. This instruction is also implemented by a new hardware component, whose VHDL code is given in listing 7.

Finally we also added the clamping code to the VHDL implementation of the `addidx` and `pcmdec` instruction in order to avoid a separate call to `clmp`. With these modifications, the code for the `adpcm_decode()` method then simplifies as given in listing 1. In the listing, the mandatory NOPs have been removed. In order to ensure comparability with the COSY-compiled version, the benchmark version contains three NOPs after each instruction.

Listing 1: Assembly of `adpcm_decode()` function with optimized CPU (NOPs have been removed)

```
; r1: Input array, r2: input array length
; r3, r4: Input samples
; r5, r6: Step values
; r7, r8: Index values
; r9, r10: Output samples
for_begin:
    sle %r13, %r2, %r0
    bnez %r13, for_end

; /* 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

; /* Output values (omitted) */
    subi %r2, %r2, $2
    j for_begin
for_end:
```

Benchmark Version 1

In order to benchmark the impacts that the modifications have on the program, we again simulated the modified program using ModelSim for a frequency of 50 MHz (CLK_HALF_PERIOD = 10ns). By greatly reducing the number of instructions inside the for-loop, the optimized program only requires a total of 36.462 instruction cycles to run to completion (see figure 1. This is a speedup of $\frac{388.814}{36.462} \approx 10,7$ compared to the original version.

Afterwards we created a new Xilinx ISE project, created the area and timings report, created the respective VCD files and executed the power analyzer to get accurate power estimation. The results of the benchmark is displayed in table 2.

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

Table 2: Benchmark results for dlx_opt_performance_1

The table shows that the number of required FPGA sliced increased by 62% from 998 to 1602, while the number of LUTs increased by 67% from 3260 to 4853. This increase in area requirements is not surprising, given that the optimized CPU includes several new and sufficiently complex hardware resources. The increased CPU complexity is furthermore responsible for the longer critical path of about 7,7 nanoseconds, which reduces the maximum reachable CPU frequency to $\frac{1}{7,7 \text{ ns}} \approx 130$ MHz. However, since the number of required cycles

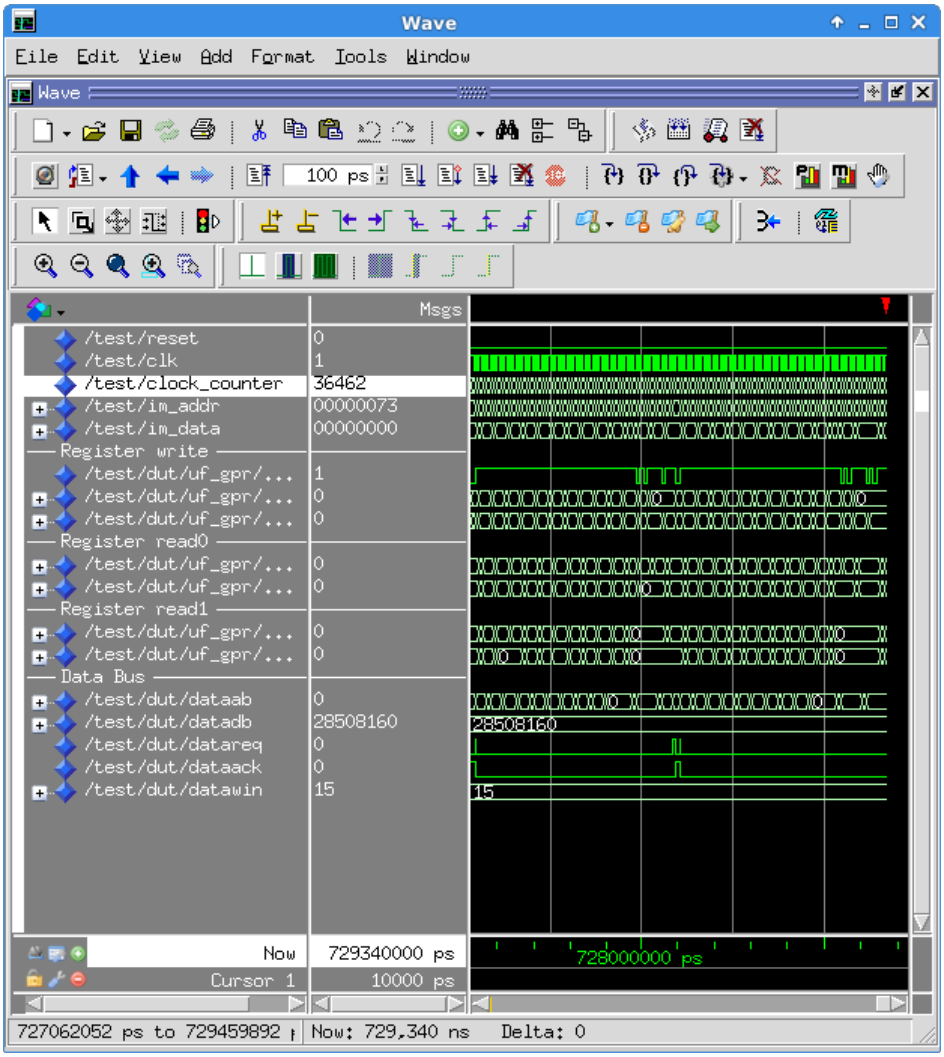

Figure 1: Output of ModelSim for the performance optimized version 1.

is much less than what the basis version needs, the execution times of the benchmark program is generally much lower than before. In both the 50 MHz and the maximum frequency case the execution time is significantly reduced, from 7,78 ms to 0,73 ms and from 2,46 ms to 0,28 ms respectively, even though the basis version uses a higher maximum frequency. The significant reduction of the necessary instruction cycles also means that the minimum frequency, with which the target sample rate of 96 kHz can be reached is also greatly reduced. More concretely, the optimized CPU is capable of decoding one sample in about $\frac{36.462 \text{ cycles}}{1300 \text{ samples}} \approx 28$ cycles, which gives a minimum frequency of $28 \text{ cycles} \cdot 96.000kHz \approx 2,7$ MHz.

In terms of power requirement, both the 50 MHz and the 130 MHz benchmark show a higher dynamic power demand, specifically 59 mW instead of 50 mW and 116 mW instead of 99 mW. This is because of the higher switching activity, caused by the newly implemented, complex instructions. On the other hand, the dynamic power demand for the minimum frequency case is less than before (23 mW instead of 37 mW), which is due to the decreased minimum frequency of 2,7 MHz instead of 29 MHz. The energy requirements are also greatly reduced (0,8 μJ instead of 8,5 μJ and 0,33 μJ instead of 2,81 μJ), except for the minimum frequency benchmark. This is because in the former two cases the execution times are significantly reduced, while for the latter the execution time slightly rises due to the decreased minimum frequency.

All in all, with a speedup of over 10 this first performance optimization has been very successful, while only moderately increasing area and power demands. The elongated critical path, resulting in a reduced maximum frequency of 130 MHz, is uncritical, because the actual FPGA board cannot be set to frequencies higher than 100 MHz. So as long as the maximum reached frequency stays above 100 MHz, there is no need to balance achievable CPU frequency and instruction count. The benchmark results regarding execution time and speedup as well as power and energy requirements for this CPU version are displayed in figures 6 and 7 in the appendix.

Implementing Version 2

The first optimized CPU showed a significant performance boost with only moderate increasing of critical path and power demands. In order to reduce the number of cycles even further, we then tried to collapse the multiple components that we created into one single big component which implements the whole PCM decoding step. Since we already process two samples in one loop iteration (by calling `pcmdec` two times), this component could even process two samples at once. In order to do this, the component needs the two input samples, the next index and the previous result as input. It then calculates the required step values, performs the decoding calculations for both input sample, and finally outputs the two decoded samples as well as the new index value, which is necessary for the next decoding call. Since we have four input and three output values, we have to use the registers for both input and output simultaneously. This is not a problem, since due to the pipeline nature of the CPU, the register reads are performed in the ID phase, while the register writes are exclusively done in the WB phase. We furthermore decided to output the second decoded sample in two registers (the one that held the input sample as well as the register for the old result value), because the second decoded sample will be required as old result value in the next instruction call. By writing this value into both registers at once, the need for register transfer instructions is avoided, which ultimately saves even more cycles. Of course this requires a total of four write ports on the GPR register file, which is the maximum of what the standard register file can provide. The new instruction has the form `pcm %r1, %r2, %r3, %r4`, where r1 and r2 hold the input samples, r3 holds the index that has been used in the previous decoding step and r4 holds the previous decoded sample. After the instruction completed, r1 and r2 hold the decoded samples, r3 holds the last used index in this decoding step and r4 again holds the second decoded sample (same as r2). Because r3 and r4 are written this way, they can remain untouched for subsequent calls to the `pcm` instruction. Just the input samples have

to be updated. This instruction is implemented using a new hardware resource, which we developed in VHDL. Its code is given in listing 8. The previously used hardware components are not necessary anymore and have been removed (along with the instructions). This might even have a positive influence on the required area.

Because the input and output operands of the `pcm` instruction have been chosen elegantly, the necessary code to implement the PCM decoding becomes even more condensed. As listing 2 shows, the for-loop only contains – besides output code – the loading of two input samples and the `pcm` instruction.

Listing 2: Assembly of `adpcm_decode()` function with optimized CPU version 2 (NOPs have been removed)

```
; r1: Input array, r2: input array length
; r3, r4: Input samples, output samples
; r5: Index input/output
; r6: Previous decoded sample
for_begin:
    sle %r13, %r2, %r0
    bnez %r13, for_end

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

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

; /* Output values (omitted) */
; /*           ...           */

    subi %r2, %r2, $2
    j for_begin
for_end:
```

Benchmark Version 2

For the benchmark of the next optimized version we again created a program file with three NOPs after each instruction and without any output code. Then we executed ModelSim, Xilinx ISE and the power analyzer to generate benchmark reports. The benchmark results are displayed in table 3.

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

Table 3: Benchmark results for dlx_opt_performance_2

With the second optimization we managed to get the instruction cycle count down to 18.258 (see figure 2), which is a speedup of $\frac{388.814}{18.258} \approx 21,3$ compared to the original version and $\frac{36.462}{18.258} \approx 2$ compared to the previous optimization. The execution times hence dropped from 0,73 ms to 0,37 ms and from 0,28 ms to 0,12 ms for the first two benchmark cases.

Furthermore the critical path has become shorter again compared to the previous version, which is because of the removed resources. For the same reason, the area requirements dropped

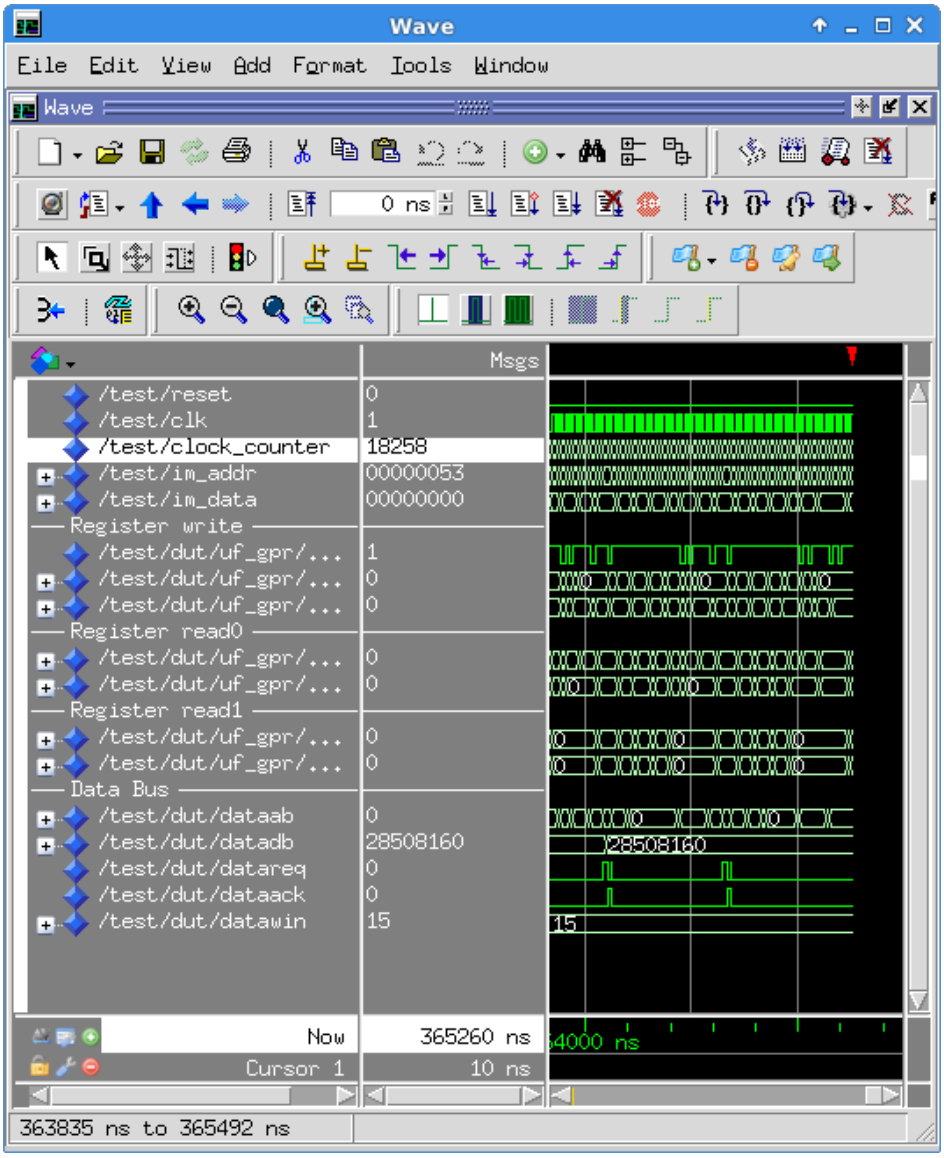

Figure 2: Output of ModelSim for the performance optimized version 2.

very slightly. This results in a higher maximum frequency of $\frac{1}{6{,}5 \text{ ns}} \approx 154$ MHz, which is almost as high as with the unmodified CPU. The power demands remained practically stable compared to the previous optimization, which is not surprising given that basically the same task is performed, just with one single hardware component instead of multiple ones. Only the minimum frequency benchmark has an even lower dynamic power requirement of 21 mW. This is because of the decreased minimum frequency of $\frac{18.258 \text{ cycles}}{1300 \text{ samples}} \cdot 96.000kHz \approx 1,3$ MHz. Given that a speedup of two results in halved execution times compared to the previous optimization, the energy requirements for the first two benchmarks also dropped by a factor of two. Once more, the minimum frequency benchmark is exempt from this because of the further reduced minimum frequency.

With this second optimization we managed to optimize the performance once more by a factor of two, without negatively affecting either area or power requirements. Furthermore the necessary code to implement a PCM decoding algorithm now is very simple and readable. Given that the number of cycles inside the for-loop is now approaching a minimum, further performance optimizations will have to concentrate more on data parallelism. The benchmark results regarding execution time and speedup as well as power and energy requirements for this CPU version are displayed in figures 8 and 9 in the appendix.

Implementing Version 3

After benchmarking the last performance optimized version of the CPU, there are a few parameters left that can be optimized in order to decrease the required execution time even further. First of all we implemented a `ble %r1, %r2, label` instruction, which branches to `label` if register r1 is less than or equal to register r2. This instruction replaces the previous `sle`/`bnez` instruction pair, which saves a total of three instructions inside the for-loop. The `ble` instruction can be implemented entirely in ASIPMeister by including an adder, similarly to the implementation of the `bgeu` instruction from the earlier session 4.

Since the number of cycles inside the for-loop has reached a minimum by now, the only way left to decrease the number of required cycles even further is to process more sample at once in a single loop iteration, i.e. increase the data parallelism of the code. In our case it is feasible to process four input samples instead of two in one single loop iteration. For this we have to implement a new load instruction called `lhh %r2, %r3, %r1`, which loads four input samples (two bytes) from the address stored in register r1 and stores each byte in a separate register. The first byte (i.e. the first two samples) are stored in register r2, while the second byte (i.e. the latter two samples) are stored in register r3. In order to save the additional `addui %r1, %r1, $4` instruction after the load, this new instruction also increases the value of register r1 by four.

The final step of this optimized CPU version is to alter the PCM decoding component in a way that it can decode four samples instead of two. This can be done by simply duplicating the existing steps to decode two samples. Of course the input samples have to be taken from the correct range of the two input registers, which is the eight to fifth bit and the fourth to last bit, respectively. As output, the highest 16 bits of the first register are set to the first decoded sample, while the lower 16 bits are set to the second decoded sample. In the same way the second output register holds the third and fourth output sample. As listing 3 shows, the for-loop has been simplified only marginally in the new version, while the main benefit arises from iterating over the for-loop in increments of four samples instead of two samples. The VHDL code of the modified PCM decoding component is given in listing 9 in the appendix.

Listing 3: Assembly of `adpcm_decode()` function with optimized CPU version 3 (NOPs have been removed)

```
; r3 = int delta_1/out_1;
; r4 = int delta_2/out_2;
; r5 = int index_in/index_out = 0;
; r6 = int output_in/output_out = 0;
for_begin:
    ble %r2, %r0, for_end

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

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

; /* Output values (omitted) */
; /*           ...           */

    subi %r2, %r2, $4
    j for_begin
for_end:
```

Benchmark Version 3

For the benchmark of the third optimized version we again created a program file with three NOPs after each instruction and without any output code. Then we executed ModelSim, Xilinx ISE and the power analyzer to generate benchmark reports. The benchmark results are displayed in table 4.

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

Table 4: Benchmark results for dlx_opt_performance_3

With the third optimization we managed to get the instruction cycle count down to 6.554 (see figure 3), which is a speedup of $\frac{388.814}{6.554} \approx 59,3$ compared to the original version and $\frac{18.258}{6.554} \approx 2,8$ compared to the previous optimization. The execution times hence dropped from 0,37 milliseconds to 0,13 milliseconds and from 0,12 milliseconds to 0,05 milliseconds for the first two benchmark cases. On the other hand this benefit has been paid for with a longer critical path of 8,03 nanoseconds instead of 6,5 nanoseconds. This results in a reduced maximum frequency of $\frac{1}{8{,}03 \text{ ns}} \approx 124$ MHz. However, since the maximum frequency is still above 100 MHz, which is the maximal frequency setting of the available FPGA board, the elongated critical path does not constitute a problem.

The dynamic power demands for the new version have risen compared to the previous optimization, from 59 mW to 67 mW and from 117 mW to 132 mW for the first two benchmark cases. This is not surprising, given that in one instruction cycle four samples have to be decoded instead of two, which results in a much higher switching activity. The minimum CPU frequency that suffices to decode the samples in time now results to $\frac{6.554 \text{ cycles}}{1300 \text{ samples}} \cdot 96.000 \text{ kHz} \approx 0,5$ MHz. Even though the new minimum frequency is much less than the previous minimum frequency (1,3 MHz), the dynamic power requirement increased slightly from 21 mW to 24

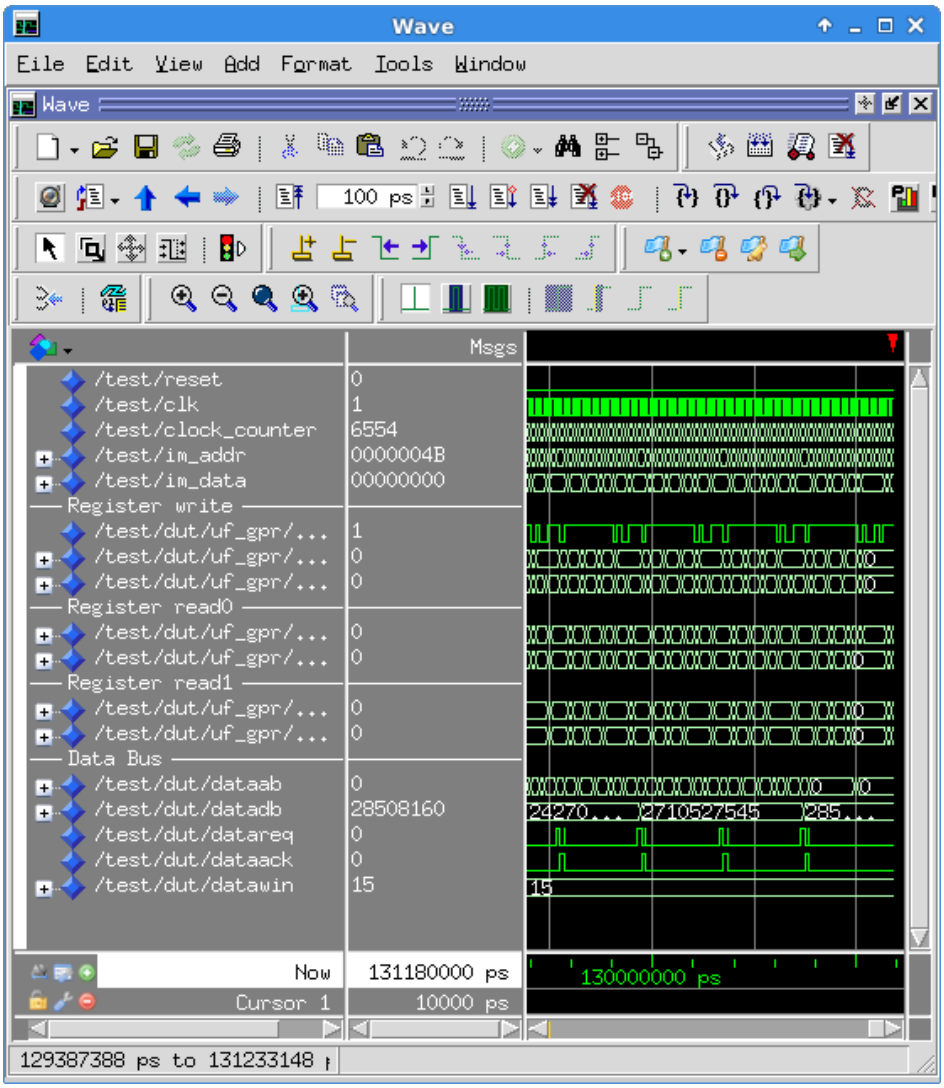

Figure 3: Output of ModelSim for the performance optimized version 3.

mW. Given that the execution time compared to the previous version again dropped by at least a factor of two, the energy requirements for the this version once more decreased significantly. In the 50 MHz benchmark case the energy consumption dropped from 0,4 μJ to 0,15 μJ, while in the maximum frequency benchmark the energy consumption dropped from 0,12 μJ to just 0,5 μJ. Hence this CPU version is simultaneously also the most energy efficient of all optimizations.

With this final performance optimized CPU we managed to decrease the number of instruction cycles once more by almost a factor of 3 compared to the previous version, even though the area and power requirements increased. Also, since there are now multiple samples stored in a single register, the code is not as readable anymore as in the previous version. The benchmark results regarding execution time and speedup as well as power and energy requirements for this CPU version are displayed in figures 10 and 11 in the appendix.

All in all, we optimized the CPU step-by-step for increasing performance, until we achieved a speedup of almost 60 compared to the basis version. The execution times and the speedup for all the CPU versions, compared at 50 MHz, are displayed in figure 4.

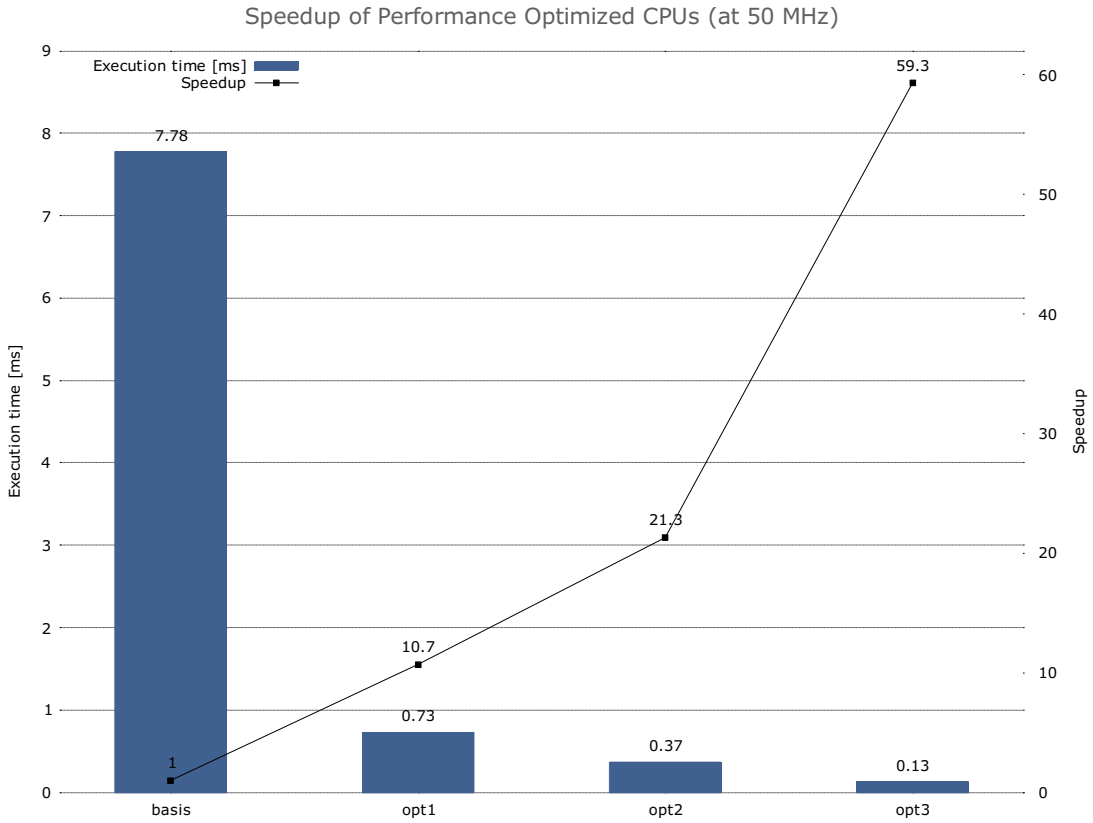

Figure 4: Speedup for the performance optimized CPU versions at 50 MHz.

Optimizing Area Requirements

Besides the optimization for performance, we also developed optimized CPUs that require as little area as possible. The area requirement is measured using the number of FPGA slices as well as the number of lookup tables (LUT) that are required for the final project.

Implementing Version 1

Minimizing the area requirements for the CPU can be done by simply removing as many hardware resources as possible from the CPU description. As a first step, we removed the `MUL0` and `DIV0` resource instances from the CPU, along with the instructions that use these resources. This was not a problem, since the `mul`, `div` and `mod` instructions are not used by the code.

Furthermore we shrunk the size of the register file from 32 to only 16 registers. Since the implementation of the registers requires much area on the FPGA board, decreasing the number of registers is the major step in minimizing the required area. With only 16 registers in the register file each register is addressed using only 4 bits instead of 5 bits, which requires a modification of either the instruction format or the micro operation implementation. We decided to leave the instruction formats unchanged (using 5 bits for register addressing) and instead modify all the macros in the micro operation implementation that are used to read or write registers. In those macros we simply removed the most significant bit before calling GPR.read() or GPR.write() with a register address. This was sufficient to make ASIPMeister generate the VHDL for the new CPU. Furthermore, we also removed the dedicated link register, and hence removed the `jal`, `jalr`, and `jr` instructions.

Of course the basis version of the code, which has been compiled using the COSY compiler, now does not run anymore on this modified CPU, because the compiler does not know about the reduced number of registers or the missing instructions. Hence we reimplemented the code in assembly using only the available instructions, and – most importantly – only the first 15 registers (registers r1-r15).

Benchmark Version 1

Like for the performance optimizations, we performed the benchmark of the area optimized CPU version by first creating a program file with three NOPs after each instruction and without any output code. Then we executed ModelSim, Xilinx ISE and the power analyzer to generate benchmark reports. The benchmark results are displayed in table 5.

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

Table 5: Benchmark results for dlx_opt_area_1

Compared to the basis version, this area optimized CPU requires only 510 (3,0%) instead of 998 (5,7%) FPGA slices and only 1661 (2,4%) instead of 3260 (4,7%) lookup tables. This means that the area requirement has decreased by almost a factor of two. The number of required instruction cycles has also decreased compared to the basis version, but this is due to the reimplementation of the code directly in assembly instead of using the COSY compiler.

Implementing Version 2

In order to minimize the area requirement even further, more hardware resources have to be removed from the CPU. One possibility is to remove the ALU hardware instance, which is a quite complex resource and hence requires a lot of space. The arithmetical operations used in the code like `add` and `sub` would then be implemented using dedicated adder instances, which are much smaller than the ALU. However, removing the ALU influences many of the existing instructions, which is why we decided not to do this. Instead, we removed even more registers from the register file, bringing the total number of registers down to just eight.

Of course, this again necessitates a reimplementation of the code in order to use only seven different registers (r1-r7). Since the number of variables that have been used in the previous implementation exceeds this number, we have to restructure the code quite significantly. For example, in each iteration the base address of the input sample array has to be reloaded yet again, because we do not have enough registers to simply store it across all iterations. Obviously this increases the number of required instruction cycles, and hence the execution time. However, as long as we stay beyond the execution time of the basis version, the optimized CPU is certainly still fast enough.

Benchmark Version 2

The benchmark results with the CPU containing only eight general purpose registers are displayed in table 6.

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

Table 6: Benchmark results for dlx_opt_area_2

Compared to the previous version, this optimization requires only 437 (2,5%) instead of 510 (3,0%) FPGA slices and only 1203 (1,7%) instead of 1661 (2,4%) lookup tables. While the area in fact decreased for this optimization, it did not decrease as much as for the last optimization. This is mainly because now only eight registers have been removed compared to the previous version, and not 16 registers as before. However, with only 2,5% of FPGA slices and only 1,7% of lookup tables in use, this optimized version is a factor of two to three better than the basis version. Furthermore, as already predicted, the number of instruction cycles increased from 202.481 to 239.506 because of the more complex implementation with only seven available registers (not counting the zero register).

The benchmark results for the two area optimized CPU versions are displayed in figure 5.

Running the Optimized Version

After we completed the benchmarks for both optimized versions, the created projects were run on the FPGA board. All tested versions produced a correct sound for at least some frequency settings. The third performance optimized version produced a correct sound at the minimal board frequency of 25 MHz, but failed for higher board frequencies. Since the calculated maximum frequency of 124 MHz is far higher, this is probably caused by the playback buffer being filled too quickly at higher frequencies, which then disturbs the correct playback. The second performance optimized version, which requires more than double the execution time to decode a sample, also runs correctly at a board frequency of 40 MHz. However, it again fails at higher board frequencies. The area optimized versions all run correctly with a board frequency of 100 MHz.

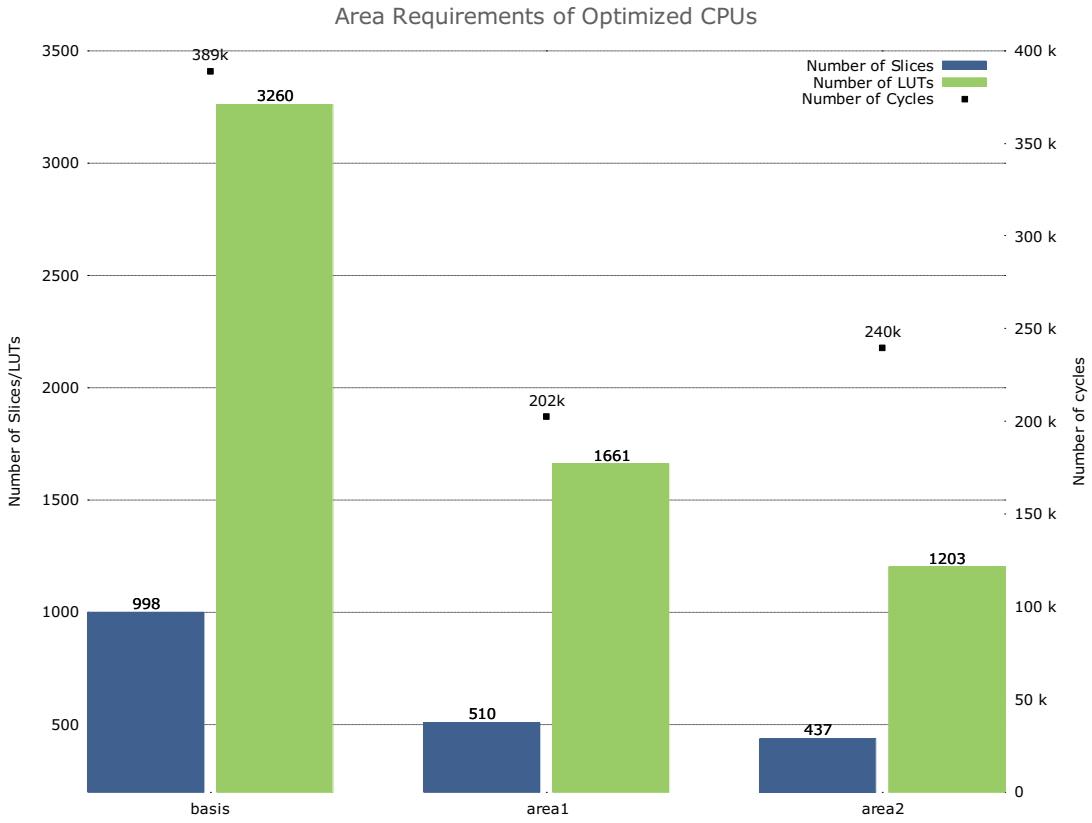

Figure 5: Area Requirements for the Area Optimized CPU Versions.

Appendices

A Detailed Benchmark Results for Performance Optimizations

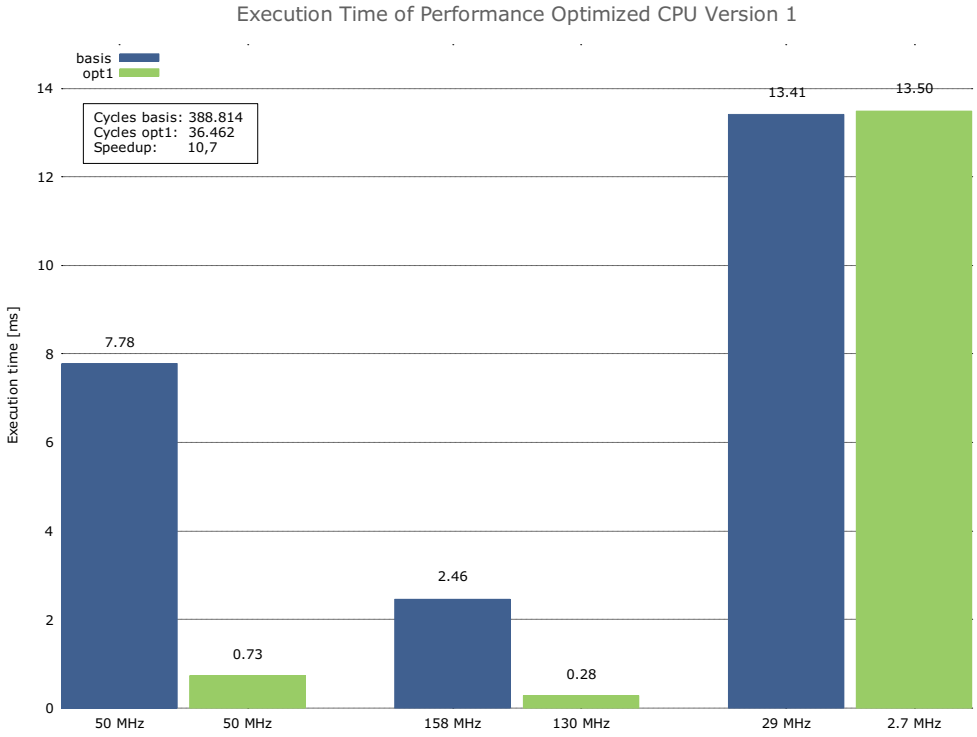

Figure 6: Execution time and speedup for the performance optimized version 1.

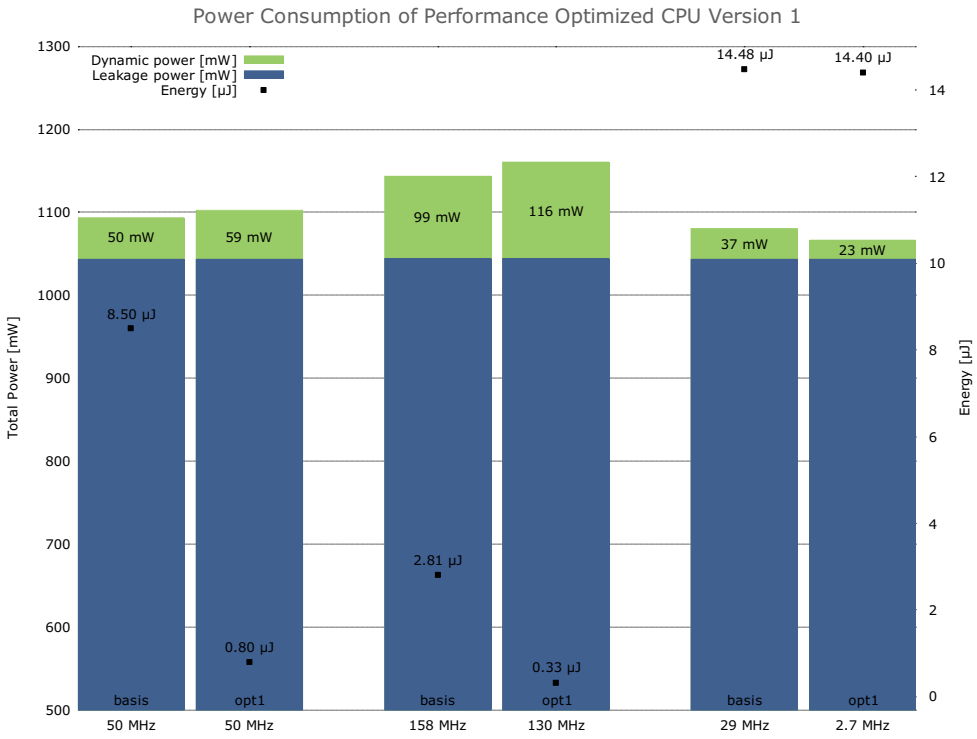

Figure 7: Power and Energy Requirements for the performance optimized version 1.

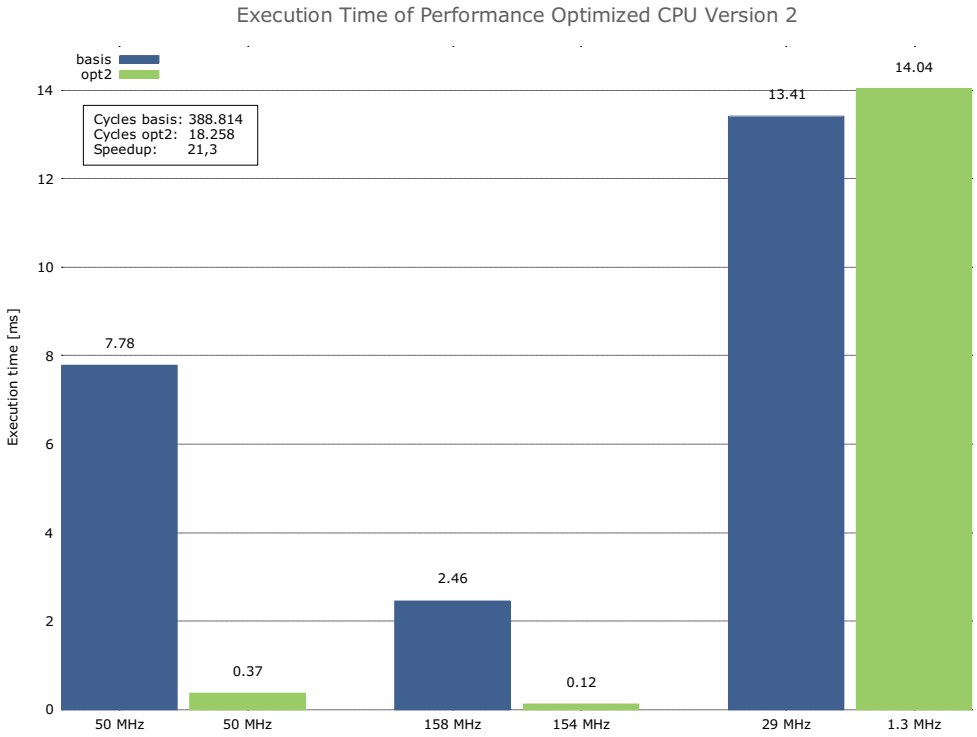

Figure 8: Execution time and speedup for the performance optimized version 2.

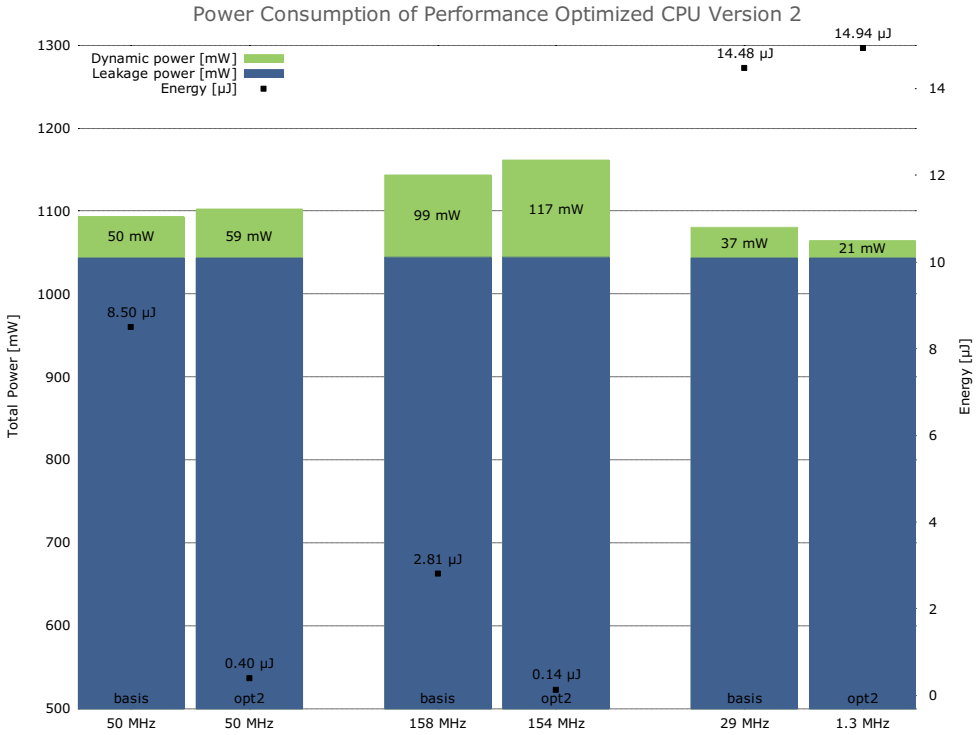

Figure 9: Power and Energy Requirements for the performance optimized version 2.

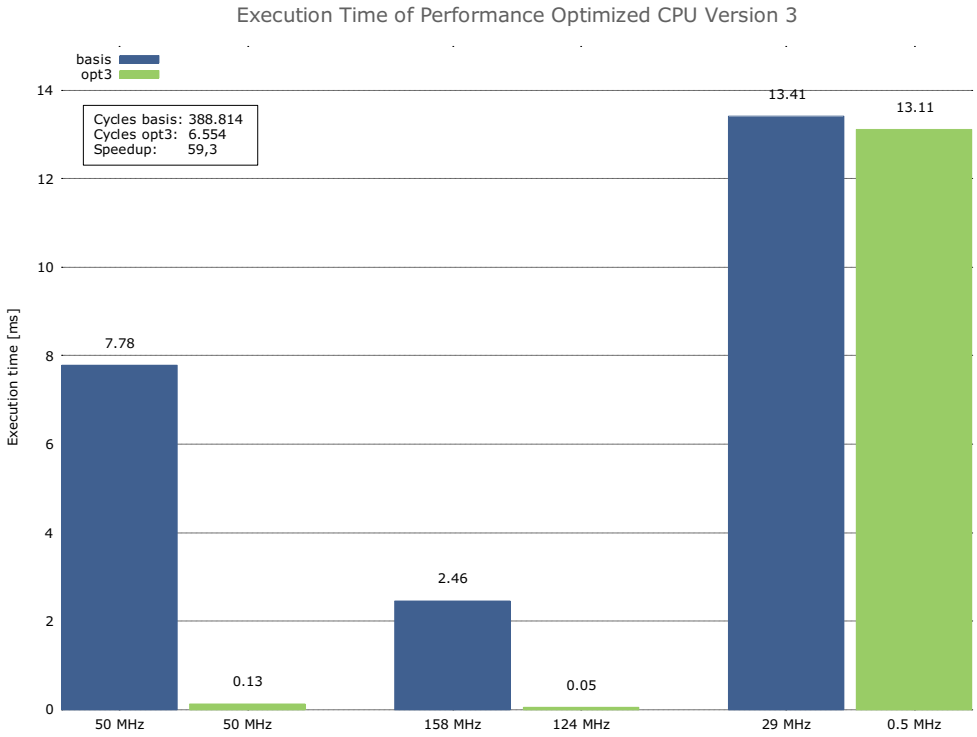

Figure 10: Execution time and speedup for the performance optimized version 3.

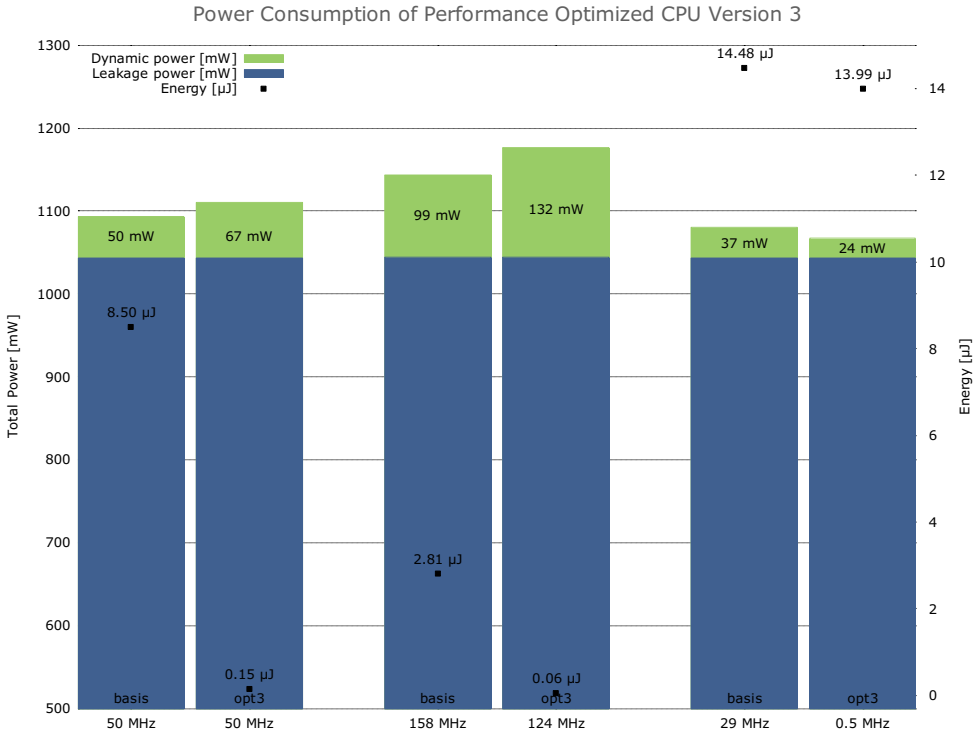

Figure 11: Power and Energy Requirements for the performance optimized version 3.

B VHDL Code for Performance Optimization

Listing 4: Implementation of `lstep %r1, %r2` instruction

```
entity fhm_stepsize_w4 is
  port (clock    : in std_logic; reset : in std_logic; enb : in std_logic;
        data_in  : in std_logic_vector(6 downto 0);
        data_out : out std_logic_vector(15 downto 0) );
end fhm_stepsize_w4;

architecture logic of fhm_stepsize_w4 is
type array_type1 is array (0 to 88) of integer;
signal stepsizeTable : array_type1:=( 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);
begin
  process (clock, reset, enb)
  begin
      if (enb = '1') then
  data_out <=
        conv_std_logic_vector(stepsizeTable(conv_integer(data_in)),16);
      end if;
  end process;
end logic;
```

Listing 5: Implementation of `addidx %r1, %r2` instruction

```
entity fhm_index_w4 is
  port (clock    : in std_logic; reset : in std_logic; enb : in std_logic;
        data_in  : in std_logic_vector(3 downto 0);
        index_in : in std_logic_vector(31 downto 0);
        index_out : out std_logic_vector(31 downto 0) );
end fhm_index_w4;

architecture logic of fhm_index_w4 is
type array_type1 is array (0 to 15) of integer;
signal indexTable : array_type1:=( -1, -1, -1, -1, 2, 4, 6, 8,
                                   -1, -1, -1, -1, 2, 4, 6, 8 );
begin
  process (clock, reset, enb)
  begin
      if (enb = '1') then
  index_out <= conv_std_logic_vector(indexTable(conv_integer(data_in)), 32) +
                                conv_std_logic_vector(conv_integer(index_in), 32);
      end if;
  end process;
end logic;
```

Listing 6: Implementation of `clmp %rd, %rs0, %rs1, %rs2` instruction

```
entity fhm_clamp_w4 is
  port (clock    : in std_logic; reset : in std_logic; enb : in std_logic;
        cmpval : in  STD_LOGIC_VECTOR (31 downto 0);
        greaterval   : in  STD_LOGIC_VECTOR (31 downto 0);
        lesserval    : in  STD_LOGIC_VECTOR (31 downto 0);
        dout    : out STD_LOGIC_VECTOR (31 downto 0) );
```

```
end fhm_clamp_w4;

architecture st of fhm_clamp_w4 is
begin
  process (clock, reset, enb)
  begin
          if (signed(cmpval) < signed(lesserval)) then
            dout <= lesserval;
          elsif (signed(cmpval) > signed(greaterval)) then
            dout <= greaterval;
          else
            dout <= cmpval;
          end if;
  end process;
end st;
```

Listing 7: Implementation of `pcmdec %rd, %rs0, %rs1, %rs2` instruction

```
entity fhm_adpcm_w4 is
  port (clock : in std_logic; reset : in std_logic; enb : in std_logic;
        delta  : in std_logic_vector(31 downto 0);
        step : in std_logic_vector(31 downto 0);
        valpred_in : in std_logic_vector(31 downto 0);
        valpred_out : out std_logic_vector(31 downto 0) );
end fhm_adpcm_w4;

architecture Behavioral of fhm_adpcm_w4 is
begin
  process (clock, reset, enb)
  variable vpdiff : signed (31 downto 0);
  begin
  if (enb_new = '1') then
        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;
  end if;
  end process;
end Behavioral;
```

Listing 8: Implementation of `pcm %rd0, %rd1, %rs0, %rs1` instruction

```
entity fhm_adpcmdecode_w4 is
  port (clock : in std_logic; reset : in std_logic; enb : in std_logic;
        delta_1  : in std_logic_vector(31 downto 0);
        delta_2  : in std_logic_vector(31 downto 0);
        index_in : in std_logic_vector(31 downto 0);
        output_in : in std_logic_vector(31 downto 0);
        out_1 : out std_logic_vector(31 downto 0);
```

```
        out_2 : out std_logic_vector(31 downto 0);
        index_out : out std_logic_vector(31 downto 0) );
end fhm_adpcmdecode_w4;

architecture Behavioral of fhm_adpcmdecode_w4 is
type array_type1 is array (0 to 88) of integer;
signal stepsizeTable : array_type1:=( 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);
type array_type2 is array (0 to 15) of integer;
signal indexTable : array_type2:=( -1, -1, -1, -1, 2, 4, 6, 8,
                                   -1, -1, -1, -1, 2, 4, 6, 8 );
begin
  process (clock, reset, enb)
  variable upper : integer := 88;
  variable lower : integer := 0;
  variable step_1 : integer;
  variable step_2 : integer;
  variable index_1 : integer;
  variable index_2 : integer;
  variable step_1_v : signed (31 downto 0);
  variable step_2_v : signed (31 downto 0);
  variable vpdiff_1 : signed (31 downto 0);
  variable vpdiff_2 : signed (31 downto 0);
  variable out_1_s : signed (31 downto 0);
  begin
      if (enb = '1') then
        step_1 := stepsizeTable(to_integer(unsigned(index_in)));
        index_1 := indexTable(to_integer(unsigned(delta_1))) +
                   to_integer(unsigned(index_in));
        if (index_1 < lower) then
                index_1 := lower;
        elsif (index_1 > upper) then
                index_1 := upper;
        end if;
        step_2 := stepsizeTable(index_1);
        index_2 := indexTable(to_integer(unsigned(delta_2))) + index_1;
        if (index_2 < lower) then
                index_2 := lower;
        elsif (index_2 > upper) then
                index_2 := upper;
        end if;
        index_out <= std_logic_vector(to_unsigned(index_2, 32));

        step_1_v := to_signed(step_1, 32);
        vpdiff_1 := shift_right(step_1_v, 3);
        if (delta_1(2) ='1') then
                vpdiff_1 := vpdiff_1 + step_1_v;
        end if;
        if (delta_1(1) ='1') then
                vpdiff_1 := vpdiff_1 + shift_right(step_1_v, 1);
        end if;
        if (delta_1(0) ='1') then
                vpdiff_1 := vpdiff_1 + shift_right(step_1_v, 2);
```

```
        end if;
        if(delta_1(3)='1') then
                out_1_s := signed(output_in) - vpdiff_1;
        else
                out_1_s := signed(output_in) + vpdiff_1;
        end if;
        out_1 <= std_logic_vector(out_1_s) AND
        "00000000000000001111111111111111";

        step_2_v := to_signed(step_2, 32);
        vpdiff_2 := shift_right(step_2_v, 3);
        if (delta_2(2) ='1') then
                vpdiff_2 := vpdiff_2 + step_2_v;
        end if;
        if (delta_2(1) ='1') then
                vpdiff_2 := vpdiff_2 + shift_right(step_2_v, 1);
        end if;
        if (delta_2(0) ='1') then
                vpdiff_2 := vpdiff_2 + shift_right(step_2_v, 2);
        end if;
        if(delta_2(3)='1') then
        out_2 <= std_logic_vector(out_1_s - vpdiff_2) AND
        "00000000000000001111111111111111";
        else
        out_2 <= std_logic_vector(out_1_s + vpdiff_2) AND
        "00000000000000001111111111111111";
        end if;
      end if;
  end process;
end Behavioral;
```

Listing 9: Implementation of `pcm %rd0, %rd1, %rs0, %rs1` instruction for the CPU version 3

```
entity fhm_adpcmdecode2_w4 is
  port (clock : in std_logic; reset : in std_logic; enb : in std_logic;
        in_1  : in std_logic_vector(31 downto 0);
        in_2  : in std_logic_vector(31 downto 0);
        index_in : in std_logic_vector(31 downto 0);
        output_in : in std_logic_vector(31 downto 0);
        out_1 : out std_logic_vector(31 downto 0);
        out_2 : out std_logic_vector(31 downto 0);
        index_out : out std_logic_vector(31 downto 0) );
end fhm_adpcmdecode2_w4;

architecture Behavioral of fhm_adpcmdecode2_w4 is
type array_type1 is array (0 to 88) of integer;
signal stepsizeTable : array_type1:=( 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);
type array_type2 is array (0 to 15) of integer;
signal indexTable : array_type2:=( -1, -1, -1, -1, 2, 4, 6, 8,
                                   -1, -1, -1, -1, 2, 4, 6, 8 );
begin
  process (clock, reset, enb)
```

```
variable upper : integer := 88;
variable lower : integer := 0;

variable delta_1 : unsigned (3 downto 0);
variable delta_2 : unsigned (3 downto 0);
variable delta_3 : unsigned (3 downto 0);
variable delta_4 : unsigned (3 downto 0);

variable step_1 : integer;
variable step_2 : integer;
variable step_3 : integer;
variable step_4 : integer;
variable index_1 : integer;
variable index_2 : integer;
variable index_3 : integer;
variable index_4 : integer;
variable step_v : signed (31 downto 0);
variable vpdiff : signed (31 downto 0);
variable out_1_s : signed (31 downto 0);
variable out_2_s : signed (31 downto 0);
variable out_3_s : signed (31 downto 0);
variable out_4_s : signed (31 downto 0);

begin

if (enb = '1') then

      delta_1 := unsigned(in_1(7 downto 4));
      delta_2 := unsigned(in_1(3 downto 0));
      delta_3 := unsigned(in_2(7 downto 4));
      delta_4 := unsigned(in_2(3 downto 0));

      step_1 := stepsizeTable(to_integer(unsigned(index_in)));
      index_1 := indexTable(to_integer(delta_1)) + to_integer(unsigned(index_in));
      if (index_1 < lower) then
            index_1 := lower;
      elsif (index_1 > upper) then
            index_1 := upper;
      end if;

      step_2 := stepsizeTable(index_1);
      index_2 := indexTable(to_integer(delta_2)) + index_1;
      if (index_2 < lower) then
            index_2 := lower;
      elsif (index_2 > upper) then
            index_2 := upper;
      end if;

      step_3 := stepsizeTable(index_2);
      index_3 := indexTable(to_integer(delta_3)) + index_2;
      if (index_3 < lower) then
            index_3 := lower;
      elsif (index_3 > upper) then
            index_3 := upper;
      end if;

      step_4 := stepsizeTable(index_3);
      index_4 := indexTable(to_integer(delta_4)) + index_3;
```

```
if (index_4 < lower) then
        index_4 := lower;
elsif (index_4 > upper) then
        index_4 := upper;
end if;
index_out <= std_logic_vector(to_unsigned(index_4, 32));

step_v := to_signed(step_1, 32);
vpdiff := shift_right(step_v, 3);
if (delta_1(2) ='1') then
        vpdiff := vpdiff + step_v;
end if;
if (delta_1(1) ='1') then
        vpdiff := vpdiff + shift_right(step_v, 1);
end if;
if (delta_1(0) ='1') then
        vpdiff := vpdiff + shift_right(step_v, 2);
end if;
if(delta_1(3)='1') then
        out_1_s := signed(output_in) - vpdiff;
else
        out_1_s := signed(output_in) + vpdiff;
end if;

step_v := to_signed(step_2, 32);
vpdiff := shift_right(step_v, 3);
if (delta_2(2) ='1') then
        vpdiff := vpdiff + step_v;
end if;
if (delta_2(1) ='1') then
        vpdiff := vpdiff + shift_right(step_v, 1);
end if;
if (delta_2(0) ='1') then
        vpdiff := vpdiff + shift_right(step_v, 2);
end if;
if(delta_2(3)='1') then
        out_2_s := out_1_s - vpdiff;
else
        out_2_s := out_1_s + vpdiff;
end if;

step_v := to_signed(step_3, 32);
vpdiff := shift_right(step_v, 3);
if (delta_3(2) ='1') then
        vpdiff := vpdiff + step_v;
end if;
if (delta_3(1) ='1') then
        vpdiff := vpdiff + shift_right(step_v, 1);
end if;
if (delta_3(0) ='1') then
        vpdiff := vpdiff + shift_right(step_v, 2);
end if;
if(delta_3(3)='1') then
        out_3_s := out_2_s - vpdiff;
else
        out_3_s := out_2_s + vpdiff;
end if;

step_v := to_signed(step_4, 32);
```

```
            vpdiff := shift_right(step_v, 3);
            if (delta_4(2) ='1') then
                    vpdiff := vpdiff + step_v;
            end if;
            if (delta_4(1) ='1') then
                    vpdiff := vpdiff + shift_right(step_v, 1);
            end if;
            if (delta_4(0) ='1') then
                    vpdiff := vpdiff + shift_right(step_v, 2);
            end if;
            if(delta_4(3)='1') then
                    out_4_s := out_3_s - vpdiff;
            else
                    out_4_s := out_3_s + vpdiff;
            end if;

            out_1 <= std_logic_vector(out_1_s(15 downto 0) & out_2_s(15 downto 0));
            out_2 <= std_logic_vector(out_3_s(15 downto 0) & out_4_s(15 downto 0));

    end if;
    end process;
end Behavioral;
```