Lab:
Customized Embedded Processor Design for IoT

Tim Bücher, Stephan Holljes, Yannick Keller | February 6, 2019

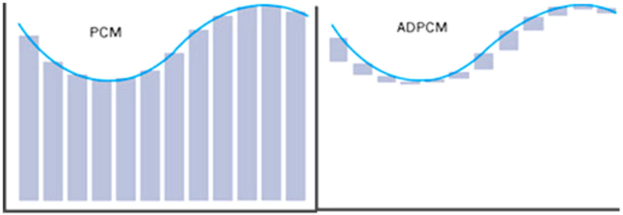
CHAIR FOR EMBEDDED SYSTEMS

Outline

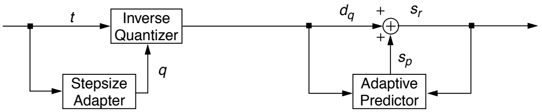
ADPCM

Adaptive Differential Pulse Code Modulation (ADPCM)

- neighbouring audio samples often similiar to each other
- variant of differential pulse-code modulation (DPCM)
- varies size of quantization step (adaptive)
- much less data to send/store

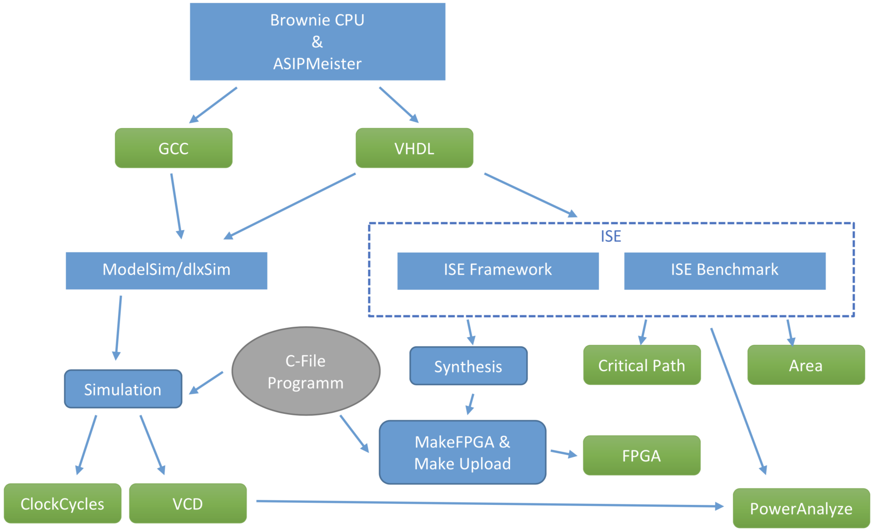


Decoder



t
Inverse Quantizer
Stepsize Adapter
q
d_q
+
+
s_r
s_p
Adaptive Predictor

Flow



Brownie CPU & ASIPMeister
GCC
VHDL
ModelSim/dlxSim
ISE
ISE Framework
ISE Benchmark
C-File Programm
Simulation
Synthesis
Critical Path
Area
MakeFPGA & Make Upload
FPGA
ClockCycles
VCD
PowerAnalyze

Optimizations

Performance and Area Optimization

Performance optimizations

Performance Optimization

- Optimized Clamping
- Optimized Value Prediction
- Added 3 extra ALUs

Performance optimizations

Performance Optimization

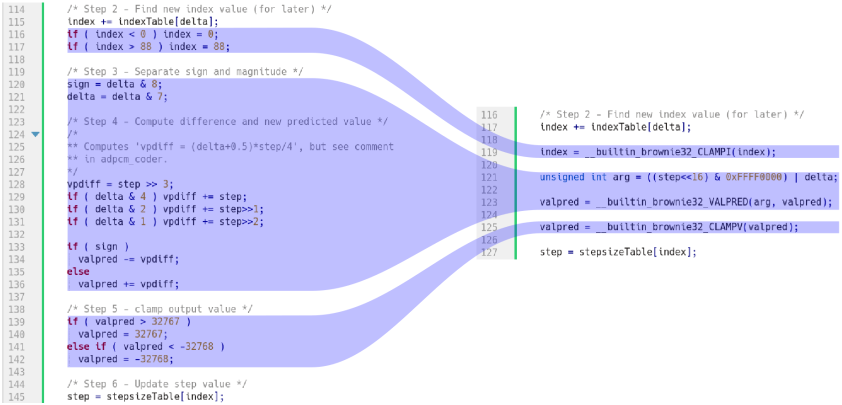
- Optimized Clamping
- Optimized Value Prediction
- Added 3 extra ALUs

Performance optimizations

Performance Optimization

- Optimized Clamping
- Optimized Value Prediction
- Added 3 extra ALUs

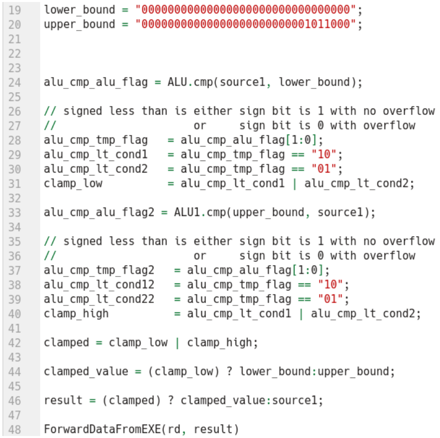
Performance optimization



```
/* Step 2 - Find new index value (for later) */
index += indexTable[delta];
if ( index < 0 ) index = 0;
if ( index > 88 ) index = 88;

/* Step 3 - Separate sign and magnitude */
sign = delta & 8;
delta = delta & 7;

/* Step 4 - Compute difference and new predicted value */
/*
** Computes 'vpdiff = (delta+0.5)*step/4', but see comment
** in adpcm_coder.
*/
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;

/* Step 5 - clamp output value */
if ( valpred > 32767 )
  valpred = 32767;
else if ( valpred < -32768 )
  valpred = -32768;

/* Step 6 - Update step value */
step = stepsizeTable[index];
```

```
/* Step 2 - Find new index value (for later) */
index += indexTable[delta];

index = __builtin_brownie32_CLAMPI(index);

unsigned int arg = ((step<<16) & 0xFFFF0000) | delta;

valpred = __builtin_brownie32_VALPRED(arg, valpred);

valpred = __builtin_brownie32_CLAMPV(valpred);

step = stepsizeTable[index];
```

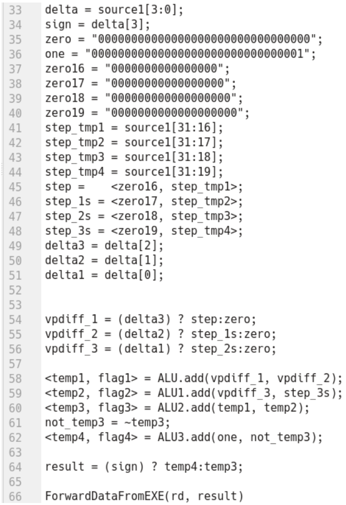
Performance optimization



```
lower_bound = "00000000000000000000000000000000";
upper_bound = "00000000000000000000000001011000";



alu_cmp_alu_flag = ALU.cmp(source1, lower_bound);

// signed less than is either sign bit is 1 with no overflow
//                    or     sign bit is 0 with overflow
alu_cmp_tmp_flag   = alu_cmp_alu_flag[1:0];
alu_cmp_lt_cond1   = alu_cmp_tmp_flag == "10";
alu_cmp_lt_cond2   = alu_cmp_tmp_flag == "01";
clamp_low          = alu_cmp_lt_cond1 | alu_cmp_lt_cond2;

alu_cmp_alu_flag2 = ALU1.cmp(upper_bound, source1);

// signed less than is either sign bit is 1 with no overflow
//                    or     sign bit is 0 with overflow
alu_cmp_tmp_flag2   = alu_cmp_alu_flag[1:0];
alu_cmp_lt_cond12   = alu_cmp_tmp_flag == "10";
alu_cmp_lt_cond22   = alu_cmp_tmp_flag == "01";
clamp_high          = alu_cmp_lt_cond1 | alu_cmp_lt_cond2;

clamped = clamp_low | clamp_high;

clamped_value = (clamp_low) ? lower_bound:upper_bound;

result = (clamped) ? clamped_value:source1;

ForwardDataFromEXE(rd, result)
```

ASIP micro-operation description for clamping functions

Performance optimization



```
delta = source1[3:0];
sign = delta[3];
zero = "00000000000000000000000000000000";
one = "00000000000000000000000000000001";
zero16 = "0000000000000000";
zero17 = "00000000000000000";
zero18 = "000000000000000000";
zero19 = "0000000000000000000";
step_tmp1 = source1[31:16];
step_tmp2 = source1[31:17];
step_tmp3 = source1[31:18];
step_tmp4 = source1[31:19];
step =    <zero16, step_tmp1>;
step_1s = <zero17, step_tmp2>;
step_2s = <zero18, step_tmp3>;
step_3s = <zero19, step_tmp4>;
delta3 = delta[2];
delta2 = delta[1];
delta1 = delta[0];


vpdiff_1 = (delta3) ? step:zero;
vpdiff_2 = (delta2) ? step_1s:zero;
vpdiff_3 = (delta1) ? step_2s:zero;

<temp1, flag1> = ALU.add(vpdiff_1, vpdiff_2);
<temp2, flag2> = ALU1.add(vpdiff_3, step_3s);
<temp3, flag3> = ALU2.add(temp1, temp2);
not_temp3 = ~temp3;
<temp4, flag4> = ALU3.add(one, not_temp3);

result = (sign) ? temp4:temp3;

ForwardDataFromEXE(rd, result)
```

ASIP micro-operation description for value prediction function

Area Optimization

Area Optimization

- Removed all unused instructions (removed 20%)

Area Optimization

ADD	ELT	LW
SUB	ELTU	SB
MUL	EEQ	SH
DIV	ENEQ	SW
DIVU	ADDI	BRZ
MOD	SUBI	BRNZ
MODU	ANDI	JP
AND	ORI	JPL
NAND	XORI	TRAP
OR	LLSI	JPR
NOR	LRSI	JPRL
XOR	ARSI	NOP
LLS	LSOI	RETI
LRS	LB	EXBW
ARS	LH	EXHW

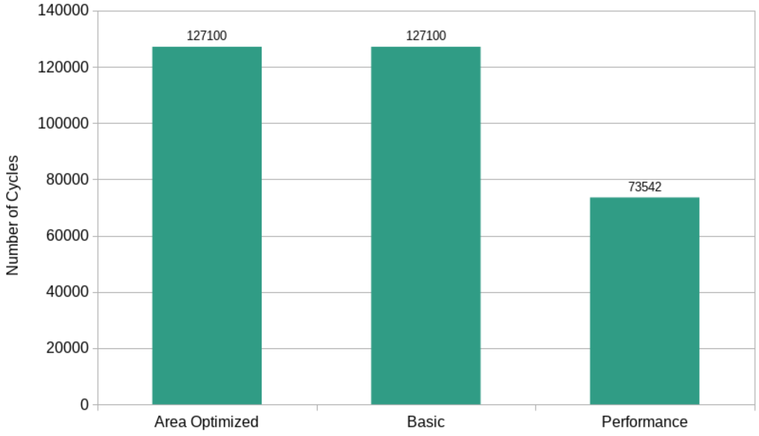
Available instructions

Area Optimization

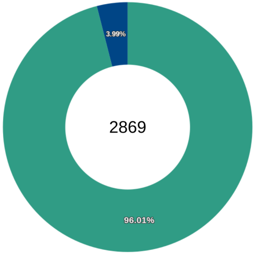
ADD	ELT	LW
SUB	ELTU	SB
MUL	EEQ	SH
DIV	ENEQ	SW
DIVU	ADDI	BRZ
MOD	SUBI	BRNZ
MODU	ANDI	JP
AND	ORI	JPL
NAND	XORI	TRAP
OR	LLSI	JPR
NOR	LRSI	JPRL
XOR	ARSI	NOP
LLS	LSOI	RETI
LRS	LB	EXBW
ARS	LH	EXHW

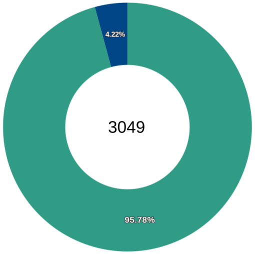
Used instructions

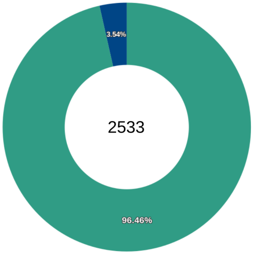
Benchmarks - Cycles



Benchmarks Area - LUTs

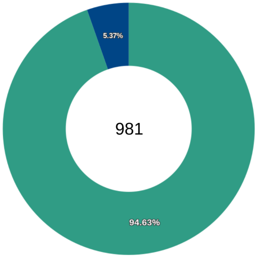




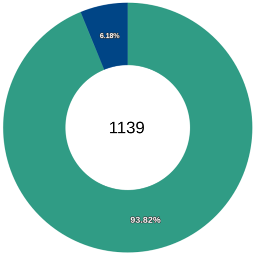


Basic	Performance	Area
100%	+6%	-12%

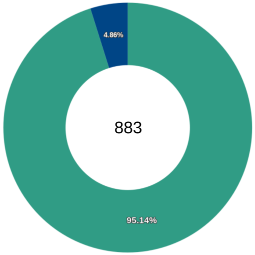
Benchmarks Area - Slices



Basic
100%

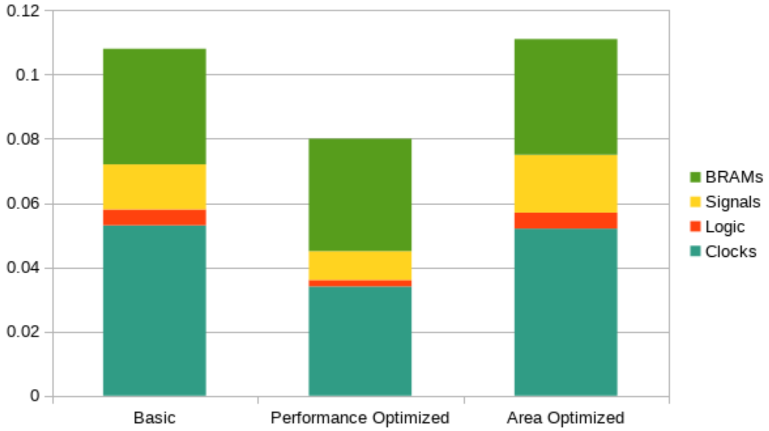


Performance
+16%



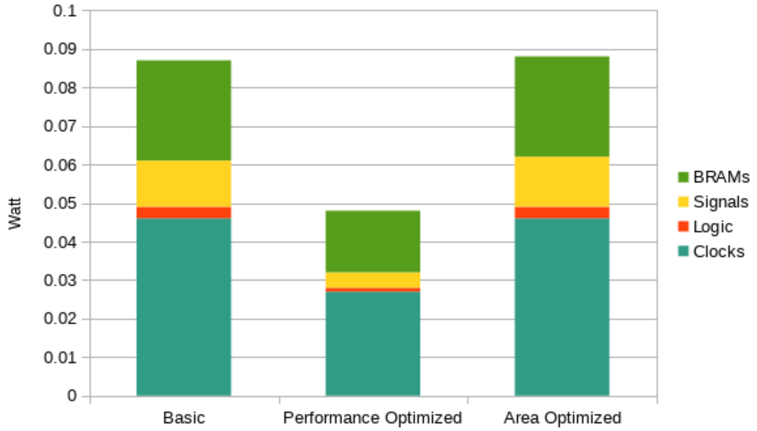
Area
-9%

Benchmarks - Power usage



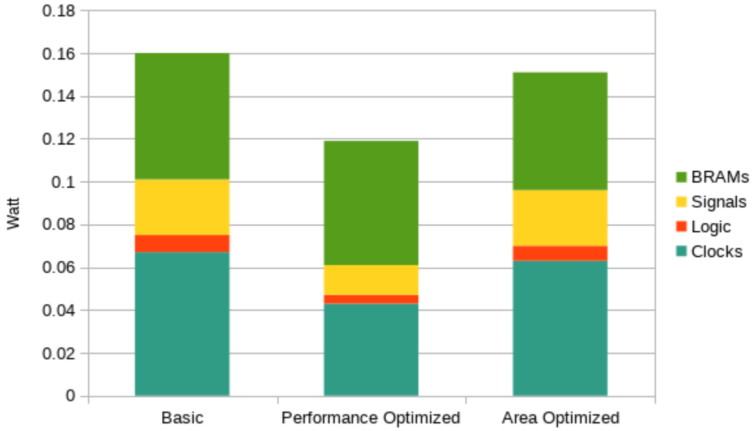
Power usage at 50MHz

Benchmarks - Power usage



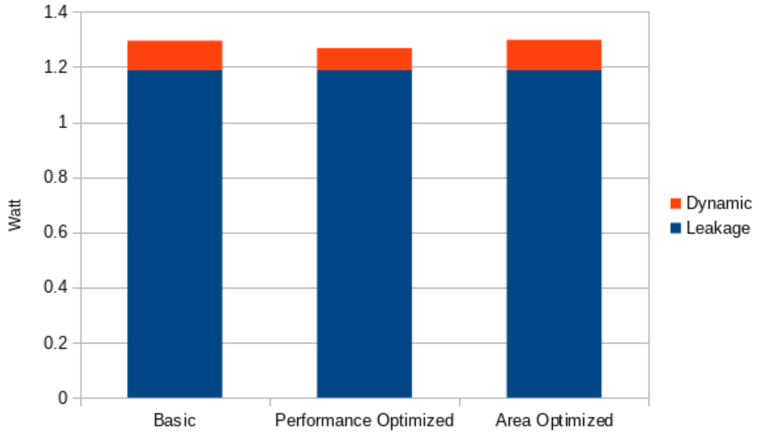
Power usage at lowest possible frequency (37MHz / 21.8MHz / 36MHz)

Benchmarks - Power usage



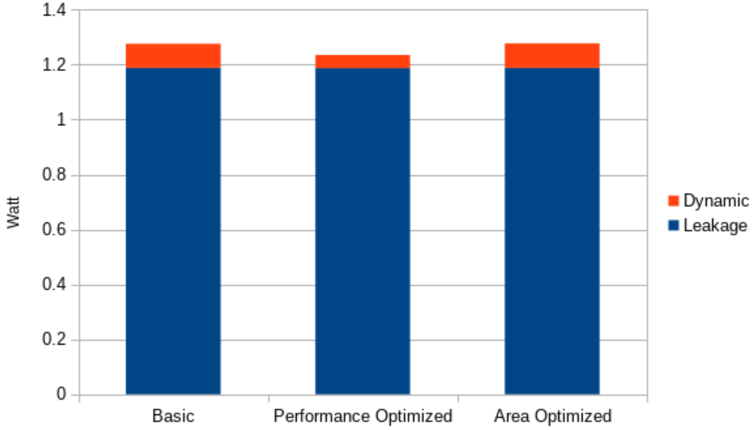
Power usage at highest possible frequency (84Mhz / 77MHz / 77MhZ)

Benchmarks - Leakage and Dynamic power



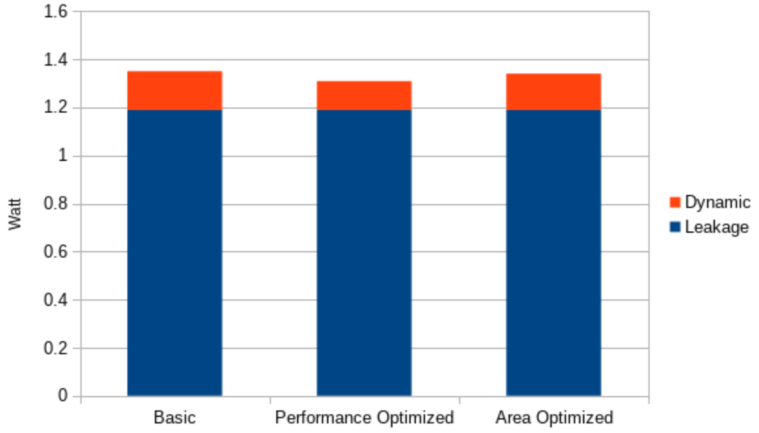
Dynamic and Leakage power at 50MHz

Benchmarks - Leakage and Dynamic power



Dynamic and Leakage power at lowest possible frequency (37MHz / 21.8MHz / 36MHz)

Benchmarks - Leakage and Dynamic power



Dynamic and Leakage power at highest possible frequency (84MHz / 77MHz / 77MHz)

Lessons learned

Combine optimizations

- Both optimizations are independent of each other
- Combination could reduce area and increase performance

Parallelize

- Decode more than one sample in one instruction
- Time spent waiting for memory write can calculate next sample

Reduce register size

- All values needed are limited to 16 bit
- Currently registers are 32 bit

Not enough ALUs

Idea:

- Use a single instruction to decode a sample

Challenges:

- Too many ALUs and calculations increase critical path
- Data dependencies

Solution:

- Use multiple instructions for portions of the algorithm

Not enough ALUs

Idea:

- Use a single instruction to decode a sample

Challenges:

- Too many ALUs and calculations increase critical path
- Data dependencies

Solution:

- Use multiple instructions for portions of the algorithm

Not enough ALUs

Idea:

- Use a single instruction to decode a sample

Challenges:

- Too many ALUs and calculations increase critical path
- Data dependencies

Solution:

- Use multiple instructions for portions of the algorithm

No easy way to add ROM

Idea:

- Use some sort of LUT for the ADPCM tables

Challenge:

- ASIPmeister does not allow to easily add ROM

Solution:

- Add a new Resource with custom VHDL code
- (Was not feasible for us in the time given)

No easy way to add ROM

Idea:

- Use some sort of LUT for the ADPCM tables

Challenge:

- ASIPmeister does not allow to easily add ROM

Solution:

- Add a new Resource with custom VHDL code
- (Was not feasible for us in the time given)

No easy way to add ROM

Idea:

- Use some sort of LUT for the ADPCM tables

Challenge:

- ASIPmeister does not allow to easily add ROM

Solution:

- Add a new Resource with custom VHDL code
- (Was not feasible for us in the time given)

Compiler too optimized (-fomit-instructions -fomg-fast)

- Compiling with gcc with -O1 -O2 and -O3 did not run on the FPGA
- However, it ran in Modelsim

Compiler too optimized (-fomit-instructions -fomg-fast)

- Compiling with gcc with -O1 -O2 and -O3 did not run on the FPGA
- However, it ran in Modelsim

CPU design too fragile?

Challenge:

- Removing unused instructions lead to failing execution on FPGA
- E.g. simply disabling the MUL instruction caused execution to fail before reaching main()

"Solution":

- Run in Modelsim

CPU design too fragile?

Challenge:

- Removing unused instructions lead to failing execution on FPGA
- E.g. simply disabling the MUL instruction caused execution to fail before reaching main()

"Solution":

- Run in Modelsim

CPU design too fragile?

Challenge:

- Removing unused instructions lead to failing execution on FPGA
- E.g. simply disabling the MUL instruction caused execution to fail before reaching main()

"Solution":

- Run in Modelsim

Reading is hard

Challenge:

- High school grade reading comprehension
- Shortest paths in Xilinx PlanAhead are not the critical path in the CPU design
- MicroOp-Code syntax has documentation

"Solution":

- RTFM
- Re-read the timing reports and re-do benchmarks

Reading is hard

Challenge:

- High school grade reading comprehension
- Shortest paths in Xilinx PlanAhead are not the critical path in the CPU design
- MicroOp-Code syntax has documentation

"Solution":

- RTFM
- Re-read the timing reports and re-do benchmarks

Reading is hard

Challenge:

- High school grade reading comprehension
- Shortest paths in Xilinx PlanAhead are not the critical path in the CPU design
- MicroOp-Code syntax has documentation

”Solution”:

- RTFM
- Re-read the timing reports and re-do benchmarks

Reading is hard

Challenge:

- High school grade reading comprehension
- Shortest paths in Xilinx PlanAhead are not the critical path in the CPU design
- MicroOp-Code syntax has documentation

"Solution":

- RTFM
- Re-read the timing reports and re-do benchmarks

Finding things to read is hard too

Challenge:

- ASIPmeister is somewhat old, documentation hard to find (or in Japanese)
- Searching the web for browniestd32 returns mostly recipies
- (Minor) Versions in the docs don't match used versions

"Solution":

- Trial and error

Finding things to read is hard too

Challenge:

- ASIPmeister is somewhat old, documentation hard to find (or in Japanese)
- Searching the web for browniestd32 returns mostly recipies
- (Minor) Versions in the docs don't match used versions

"Solution":

- Trial and error

Workflow

Challenge:

- In general a longer turn-around time for feedback than in usual software develpment
- ”Blind coding” (no linting or other tools, feedback during compilation)
- Code generation (VHDL files, binutils) takes especially long
- Debugging the FPGA was unfeasible for us
- (For one day ASIPmeister wouldn’t start at all, complaining about license issues)

Solution:

- Patience

Workflow

Challenge:

- In general a longer turn-around time for feedback than in usual software develpment
- ”Blind coding” (no linting or other tools, feedback during compilation)
- Code generation (VHDL files, binutils) takes especially long
- Debugging the FPGA was unfeasible for us
- (For one day ASIPmeister wouldn’t start at all, complaining about license issues)

Solution:

- Patience