Evaluation of a Behavioral Level Low Power Design Flow Based on a Design Case

Frank Poppen, Wolfgang Nebel

OFFIS Research Institute

Frank.Poppen@Offis.de

ABSTRACT

SoC designers face two main problems nowadays. First, the complexity of ASICs is doubling every 18 months, following Moore's Law, while the productivity of designers evolves at a much slower pace. This leads to a problem known as "the design gap". Second, designers of sub-micron devices have to observe the power dissipation of their SoC. The enhancement of battery energy capacity fails to keep up with increasingly power consuming applications. This is critical to handheld products like cellular telephones and PDAs, which will drive the market in a future world of wireless communication.

Higher levels of abstraction need to be introduced to approach the design gap and in order to handle the billions of transistors of future designs. Synopsys offers its Behavioral Compiler (BC), which introduces this higher level of abstraction. Designers do not need to schedule manually and do binding of operations if they enter the algorithms only.

Integrating BC into a design flow together with Synopsys' PowerCompiler offers the opportunity of a high-level low-power design-flow with automated gated clock insertion and operand isolation. This results in a promising methodology reducing development time and power dissipation.

This paper introduces a behavioral level low power design flow and evaluates its applicability based on a design case. The design space is being explored and several architectures for the chosen filter algorithms are synthesized starting from a behavioral HDL specification of a bank of infinite impulse response filters (IIR). Each solution is examined at gate level for power dissipation.

1 Introduction

During the early days of software engineering, programmers had to code algorithms in hardware-near languages, like assembler. The results were fast and efficient routines for designated processors. The big disadvantage of this code was its uselessness for any other architecture. For every new processor, the engineer had to start all over again. This resulted in the 1960's software crisis, which was overcome with the invention of the first compilers. Today, it is unthinkable to code huge software projects in assembler.
The EDA industry has reached a similar point. While IC designers' productivity is growing by a rate of 21% per year, the submicron silicon technology capability increases by 58%. Designers have growing difficulties to handle the fast advancing technologies. This phenomenon is known as "the design gap". "IP reuse" is the buzzword that comes up most of the time as a solution. Reality shows that today's designs are still not implemented for reuse, just like software coded in assembler. This is not necessarily the designers fault. Part of the problem is based on the design flows that are used. Shifting the entry point to a higher abstraction level – from register transfer (RT) to behavioral – might be the solution. Synopsys offers the Behavioral Compiler (BC) in order to achieve this.
Behavioral synthesis has to prove its advantage over other methodologies. In [1] and [2] it has been shown that BC can replace a RT level design entry and generate improved designs, with respect to timing and area. In [3] we stated that a behavioral flow has no negative effect on power consumption. The results of both flows are the same. The advantage of BC is a simplified design space exploration. This fact can improve power consumption, if the designer gets a better overview on his possible implementations. In this paper, we would like to explore the entire design space. The design case is a more complex version of the Gammatone Filterbank (GFB) introduced in [3].
In the following chapter, we give a quick explanation of the design used. We introduce the algorithms, the top-level architecture and the differences to the design case of [3]. We also give a quick introduction into the behavioral coding style (Chapter 3). In Chapter 4 we depict the behavioral design flow employing Synopsys' BC, DC and Power Compiler, Mentor's ModelSim and OFFIS' ORINOCO®. Chapter 5 resembles the core of this paper. Several case studies are explained and analyzed. The conclusions and recommendations are summarized in Chapter 6. Appendix A shows the tables of estimated power values. The paper closes with the references in Appendix B.

2 Design Case

The design case is a bank of 60 IIR fourth order Gammatone filters [12]. It is an ideal case because it contains typical components of a SoC: controller, data flow path, ROM and RAM. The algorithm of one IIR is shown in Figure 1. It is the same we use in [3]. The variation is the number of filters – 60 instead of 6. The data flow contains 24 multiplications, 12 additions and 4 subtractions. The calculation depends on the input (IIF), the filter constants (a, Rb, Ib) which are stored in ROM (1.4 kBits) and eight previously calculated values (Real1 to 4 and Imag1 to 4) that are kept in RAM (7.5 kBits). These calculations are executed 60 times at a frequency of 16.276 kHz, which results in approximately 39 MIPS.
The design consists of two main parts. Each part is specified in an operational loop (compare with chapter 3). The *channel_loop* implements the data flow of Figure 1, while the *controller_loop* specifies ROM and RAM accesses. The *controller* reads the filter constants and the previously calculated values from ROM and RAM. After reading, the values are fed to the *channel*. Results are written back into the RAM. Figure 2 shows the simple structure of the design.

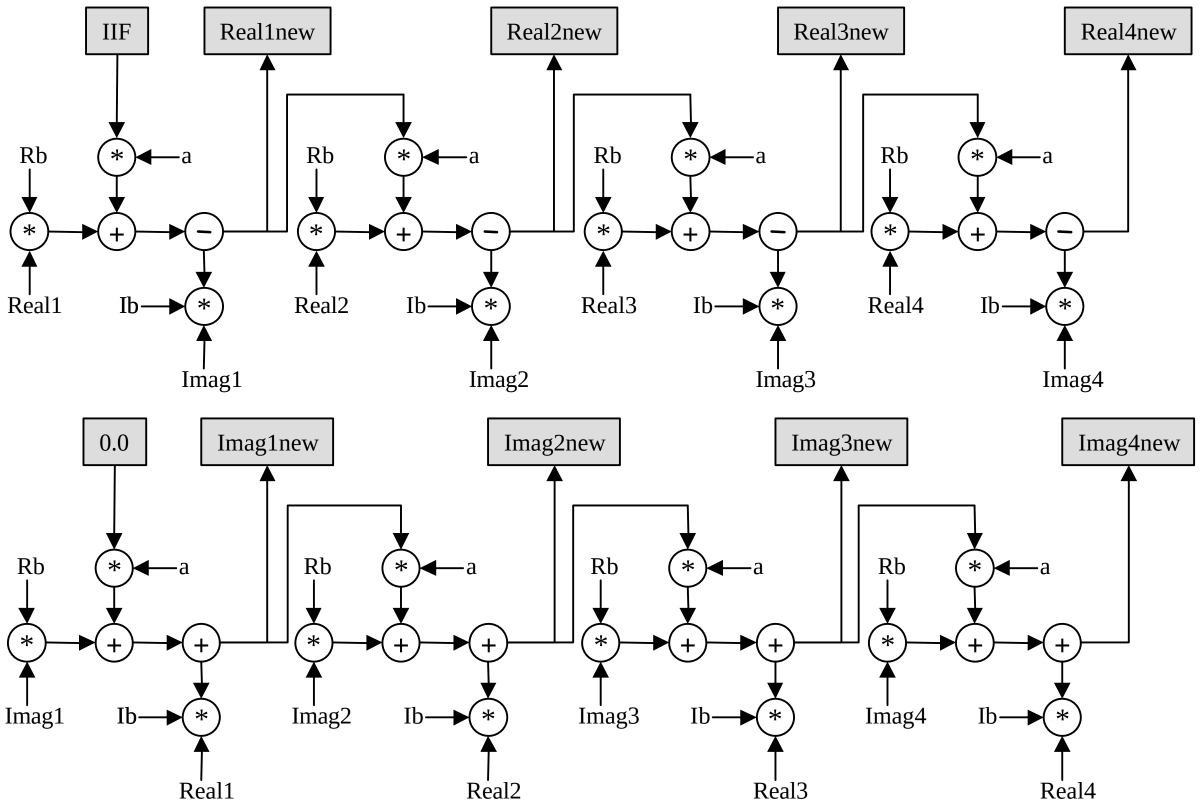


Figure 1: Algorithm of Channel

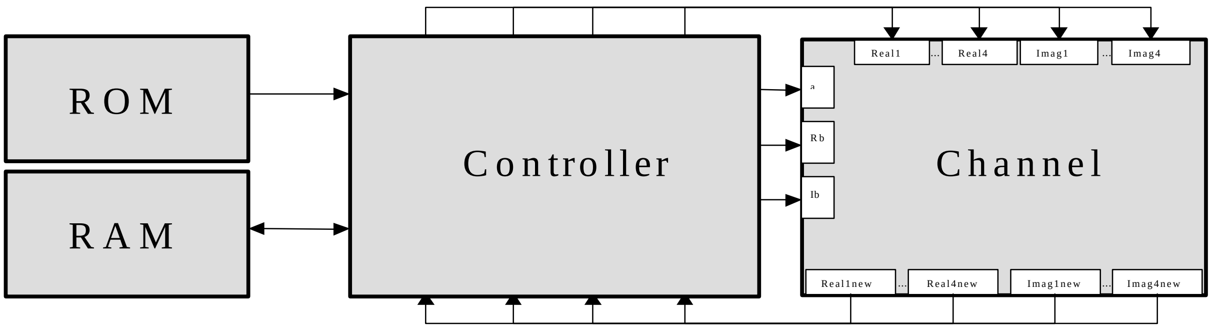


Figure 2: Top Level Block Diagram

3 Behavioral VHDL

BC requires a special HDL coding style for behavioral VHDL or Verilog. We chose behavioral VHDL since it is a more powerful language at the behavioral level. The strength of Verilog can be seen at the lower levels of abstraction – RTL and gate level.
The advantage of BC over DC is component inferencing. There exist several cases where DC requires component instantiation – e.g. for memory usage. The handling of signals like memory enable, output enable, write enable etc. are not transparent for the designer. He has to implement the correct signal assignments for read or write accesses next to the designs functionality. A tool called MemoryWrapper encapsulates memory signal assignments for BC. The methodology is described in [9]. BC infers memory for declared arrays of variables. Have a look at the following code:

```
variable VsReal1ram       : TarrayRamReal1;
    ...
variable VsImag4ram       : TarrayRamImag4;
constant  RR_RAM_16_BIT   : resource := 0;
attribute variables of RR_RAM_16_BIT  : constant is "VsReal1ram ... VsImag4ram";
attribute map_to_module of RR_RAM_16_BIT : constant is "RR_RAM_16_BIT_wrap";
```

Further reads or writes are reduced to:

```
signal_read_value_from_ram  <= VsReal1ram(integer_address);
VsReal1ram(integer_address)  := signal_write_value_to_ram;
```

Another disadvantage of DC is the use of n-stage multipliers in Synopsys' DesignWare library (DW). DC can only instantiate these components. Inferencing is not an option. BC is able to use these components through component inferencing. This makes the handling transparent and easy.
BC is capable in finding an allocation, binding and scheduling for the given algorithm and its operations. Multiplications and additions or subtractions are mapped to Synopsys' DesignWare foundation library automatically. The structure of a process differs compared to RTL code. It consists of at least two infinite loops – the reset and the main loop:

```
p_gfb : process
begin
    reset_loop: loop
                ...                                      -- Reset all signals and variables
    wait until SlClk'event and SlClk = '1';
    if SlReset = '1' then exit reset_loop; end if;
        operational_loop : loop                          -- Normal operation
              ...
        wait until SlClk'event and SlClk = '1';
        if SlReset = '1' then exit reset_loop; end if;   -- End of Superstate 1
           ...
        wait until SlClk'event and SlClk = '1';
        if SlReset = '1' then exit reset_loop; end if;   -- End of Superstate N
        end loop operational_loop;
    end loop reset_loop;
end process p_gfb;
```

Each '*wait until Clk*' statement followed by an '*exit loop*' defines a so-called super state. Super states may take more than one clock-cycle to execute. More details can be found in [8]. The behavioral level specification excluding testbench consists of 462 lines of code.

4 Low Power Behavioral Design Flow

In this chapter, we explain the design flow we used for architecture generation, synthesis, power estimation and power optimization. Our methodology is visualized in Figure 3. The input is behavioral VHDL. Three loops are implemented: *reset_loop*, *controller_loop* and *channel_loop* (compare with Chapter 3). We created several constraints files to explore the design space. BC generates an appropriate allocation, scheduling and binding within these constraints. We are able to engender different architectures this way without changing the initial VHDL specification.

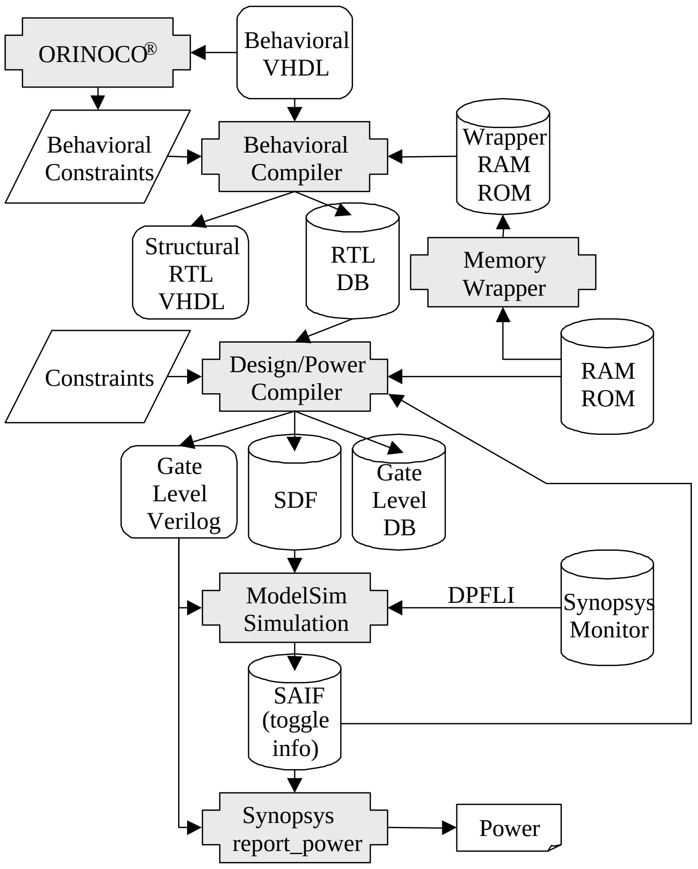


Figure 3: Used Design Flow

Our in-house developed tool, ORINOCO® [7], offers behavioral level power estimation and supports SoC-designers to create behavioral synthesis constraints for low power design. We ignored the first feature in the scope of this work and concentrated on the behavioral low power synthesis. We experienced that the power dissipation of an algorithm depends on the architecture and the processed input data [4 – 6]. The values shown in Figure 4 are estimates of the power dissipation in architectures with different allocations – one to thirteen resources. Keep in mind that all thirteen architectures evaluate the same algorithm. The binding of operations is a design decision with more than one resource. The figure shows that the binding has great effect on power dissipation. If a binding destroys correlation in the data path, it raises the switching activity. The power dissipation is high (black bar). A well-chosen binding will enhance data correlation and reduce power dissipation (gray bar). An interesting fact is visible in Figure 4.

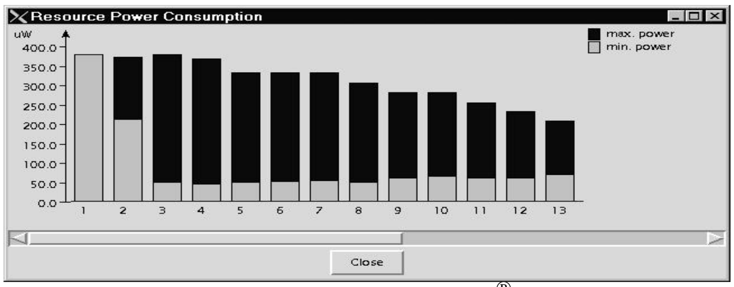


Figure 4: ORINOCO®

One resource might consume more power than three instances of the same component. When allocation and binding are well chosen, the most power efficient design is not necessarily the one using the least resources. ORINOCO® generates behavioral constraints files to apply the estimated best binding and allocation to BC. This file includes the BC command set_common_resource. For each resource, this constraint is set. A list contains the operations to be bound to each resource.

```
set_common_resource { \
    p_gfb/reset_loop/controller_loop/channel_loop/mult_95/  \
            ...
    p_gfb/reset_loop/controller_loopchannel_loop/mult_167/ \
} –max_count 1
```

BC schedules user defined RAM and ROM with the help of a wrapper. Synopsys' tool MemoryWrapper encapsulates vendor memories. This offers a simple interface to include memory IP in the flow. Reference [9] depicts how to start and use this tool. The wrapper is needed to define the signal sequences for read and write accesses on the memories. In this way, BC is able to map arrays of variables on these and schedule the accesses.
After scheduling, binding and allocation, structural RT level VHDL code is written out for functional verification (not drafted in Figure 3). A Synopsys' database (DB) serves as input for Design Compiler (DC) and Power Compiler respectively. The next step is a regular RT to gate level synthesis. We used LSI Logic's G10^{TM}-p cell-based CMOS technology. The constants are: 3.3V, 0.35 micron, 25°C.
The result is a gate level netlist of the design, which is written out to Verilog and DB. In [3] we used VHDL for gate level simulations. We changed to Verilog for two reasons. Firstly, ModelSim simulates our Verilog netlists observable faster than VHDL. Secondly, backannotation of SDF and SAIF works smoother with Verilog.
Power Compiler offers the features "recompile for power optimization", clock gating and operand isolation. We employ the first methodology to improve the energy balance. Have a look at
[10] for a more detailed description. However, in the scope of this paper we do not handle clock gating and operand isolation. We included these methodologies in [3].
Hereafter, the signal activity is being traced during a ModelSim VHDL/Verilog co-simulation. We use a VHDL testbench to stimulate the Verilog netlist. The testbench is the same we use to stimulate the behavioral VHDL specification. We do not need different testbenches for each abstraction level. A monitoring entity is linked to the simulator via the foreign interface. This monitor traces signal activities and writes them out in a Switching Activity Interchange Format (SAIF).
With the command read_saif the activity information is back annotated to Synopsys' Power Compiler. This enables Power Compiler to estimate dynamic power and to optimize the design. The command report_power gives an overview on the power dissipation. The netlist can be improved in reverence to power with another compile-run. Power Compiler requires a power-characterized library for this function.

5 Analysis

Many parameters influence the power dissipation of a design. In this chapter, we introduce several cases where some of these parameters are varied while others are fixed.
The first fixed parameter is "choice of algorithm". Several algorithms to implement digital filters are known, e.g. finite impulse response filters (FIR) or infinite impulse response filters (IIR). Some of them are more power efficient then others. It is possible to use high level

power estimation tools like ORINOCO® to find power efficient algorithms very early in the design flow. This is a very promising methodology, since the best power optimizations are achieved at the higher abstraction levels (compare with Figure 5).

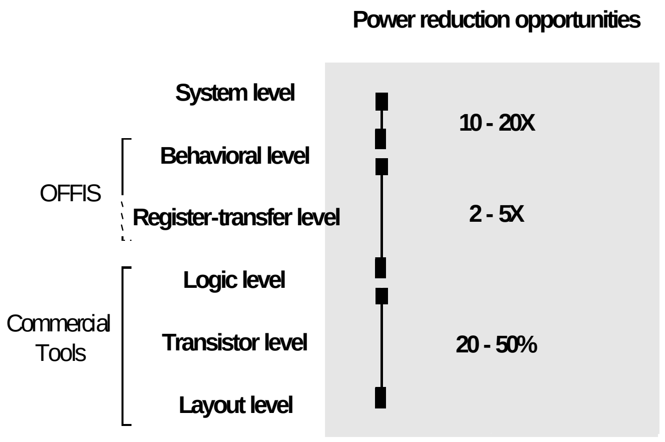


Figure 5: Power Reduction Opportunities [11]

However, including several algorithms, would go beyond the scope of this paper. This work examines only one algorithm to implement the GFB (compare with Chapter 2).
The second fixed parameter is “choice of target technology”. We chose to use LSI Logic’s G10^{TM}-p cell-based CMOS technology. The constants are mentioned in Chapter 4.
We varied the parameters “simulation stimuli and period”, “architecture” and “power optimization methodologies”. All power estimates exclude the power consumption of RAM and ROM. These values are invariant to the test cases and are therefore of no significance to this paper. We begin to analyze the effect of different stimuli and simulation periods on power estimates.

5.1. Effects of Stimuli and Period

In Chapter 4 we introduce the behavioral low power design flow. We see, that a gate level simulation is necessary to obtain the switching activity for power estimation. Formula (1) is an approximation of the switching power $P_{dynamic}$ of static CMOS designs.

$$P_{dynamic} = KC_{out}V_{dd}^{2} f \tag{1}$$

C_{out} is the output capacity and V_{dd} the supply voltage. These constants are defined by the vendor’s technology library. K is the average number of rising transitions during one clock cycle and f is the clock frequency. We generate SAIF files during gate level simulation to determine these values.
The 60 filters of the GFB – each has a different mid-frequency – react differently depending on the stimuli. Table1, 3 and 5 show the estimated *average* power dissipation, for the input stimuli “sinus wave”, “white noise” and “human speech sample”. The power dissipation in RAM and ROM is invariant to all the test cases in this paper. Thus, we tabulated the power consumption in “mW” without the memories’ share. The left column names the architecture. GFB is implemented using the multiplier dw02_mult(nbm[1]). The architectures pipeN_GFB use the components dw02_mult_N_stage(str[2]), which are pipelined DesignWare multipliers, instead. The right columns reveal the used resources. Each row of power values is the result of a simulation with different simulation periods – from 3.5 to 14 ms.

[1] None-Booth-Recoded Wallace Tree Synthesis Model
[2] Pipelined Wallace Tree Multiplier Synthesis Model

We see, that the estimated power dissipation for a sinus stimulus rises with the simulation period. The reason for this effect is the transient response of digital filters. The filters need some time to lock into phase. Before that time, the activity is low. For the second experiment with white noise as input (Table 3) the values are nearly independent from the simulation period. White noise covers the entire spectrum of frequency. This stimulates all 60 filters instantaneously and at all times. In Table 5, we observe a contrary trend. The figures become smaller. Our human speech sample has higher amplitudes at the beginning. This induces more activity to the corresponding filters. Then the input quiets down. The activity in the filters is reduced and the average power dissipation becomes smaller.
Tables 2, 4 and 6 show the absolute relative error standardized to the values of the estimates with the longest simulation period, which is 14 ms. The sinus stimulus requires a longer simulation time for meaningful average power estimates. The discrepancy is higher than 3 %. The other two stimuli are not as dependent as this signal to a longer simulation period. Even with a short period of 3.5 ms, the error is below 1 %. The following power estimations will use this short simulation period and the human speech stimulus. This is legitimized due to the small divergence and the fact that a speech stimulus is closest to the GFB's application domain.

5.2. Effects of Architectures

In this chapter, we examine a greater variety of architectures. In [3] we presumed that DesignWare's N-stage multipliers are not a good choice for low power designs. Table 7 and its visualization in Figure 6 confirm this thesis. Listed are the power estimates for architectures that infer multipliers from a regular dw02_mult(nbm) to a dw02_mult_6_stage(str) – a multiplier with six pipeline-stages. We also varied the number of inferred components per architecture. We started the series with one allocated multiplier and raised the resource usage to three.

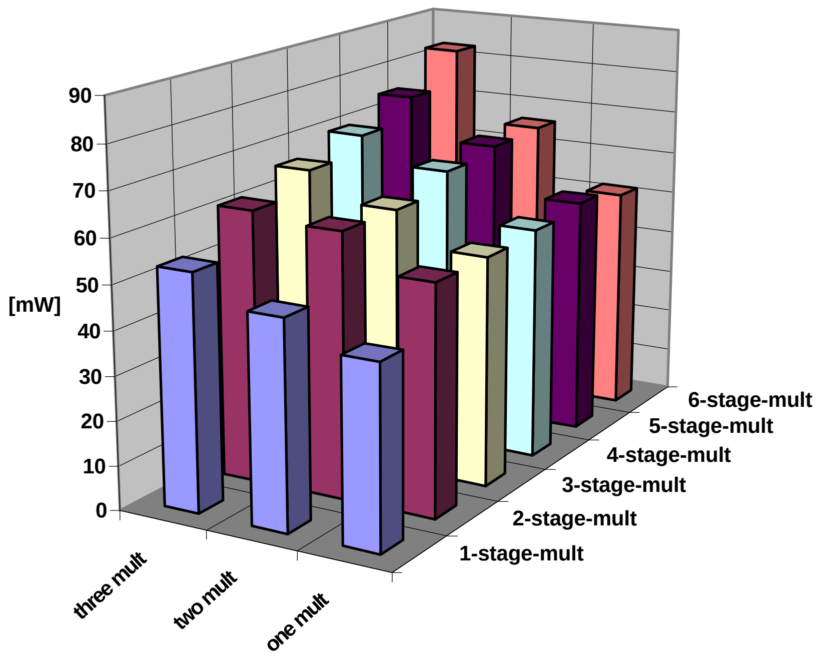


Figure 6: Power Estimation on Architectures

Our behavioral script only constrained the number of multipliers. The usage of adders and subtractors is unconstrained. This has some interesting effects on BC's scheduling strategy. BC infers three adders, one adder-subtractor and one subtractor for a design with three non-pipelined multipliers. Only one adder and one adder-subtractor are used for the one multiplier design. The usage of several multipliers requires extra control logic to route the dataflow. These extra multiplexors extend the length of the critical path such that BC requires additional adders and subtractors. The usage of pipelined multipliers slackens the critical path. These designs require only one adder and one adder-subtractor. Still, the power dissipation is higher compared to the design with the regular multiplier and the extra adders and subtractors. The effect of adders and subtractors on power is minor compared to the dissipation of the multipliers.

5.3. Power Optimization Technologies

In this chapter we use three approaches to optimize power consumption: Synopsys' PowerCompiler, OFFIS' ORINOCO® and variation of number representation. Clock gating and operand isolation are further promising low power techniques. Inside the frame of this work we chose not to evaluate these methodologies. We discussed these topics in [3].
ORINOCO® is a high level power estimation and optimization tool. It finds a power efficient allocation and binding so that the activity on resources like multipliers and adders is reduced. This proceeding is illustrated in Figure 7.

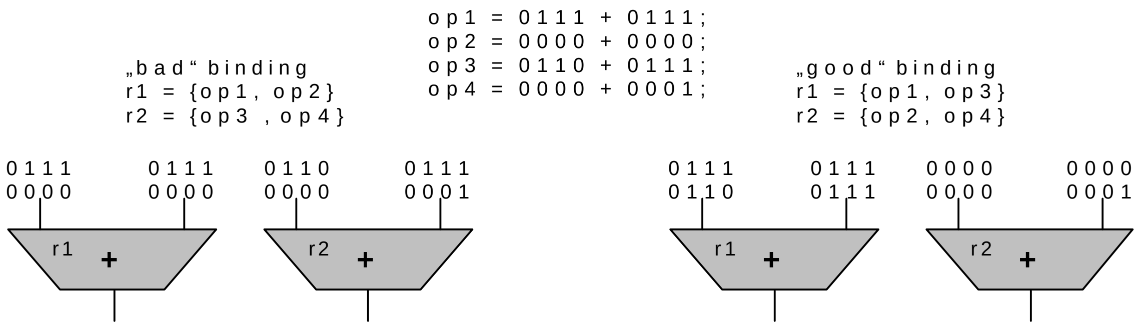
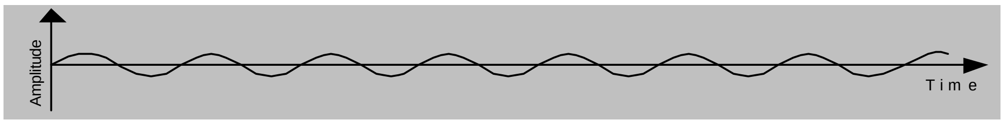


Figure 7: Power Efficient Binding

Shown is an allocation of two adders, r1 and r2. The operations op1 to op4 need to be scheduled and bound to the two resources. The "bad binding" example induces high activity to the adder because of the large Hamming distance in between the input values. The second binding causes less activity at the adders' inputs. Power dissipation is reduced without having a penalty on timing and area constraints. [4 – 6] handle the topic in detail.
Table 8 shows the power values of architectures with optimized binding. The "power" column contains the power estimates of these architectures after schedule and compile. The "optimized power" column gives the power estimates of the same architecture after it has been recompiled with annotated SAIF and the constraint *set_max_dynamic_power* 0.0 nW. We see that more resources can be used to lower power dissipation. An architecture with four multipliers, seven adders and one subtractor uses 12.54% less power then the same architecture with only two multipliers. That the binding is crucial can be seen if we compare the "good binding" section with the "bad binding". The constraints set by ORINOCO® enhance power efficiency by up to 17%. To stay fair, we also have to point out that the tool failed on the architecture with two multipliers. The "good binding" delivers a 8% worse power dissipation. There are two possible explanations for this behavior. The more resources are allocated the more binding variations exist. Two multipliers might tighten the design

space too much. The other explanation weighs more. ORINOCO® currently does not rate control logic's power dissipation. Complex structures of control logic can outweigh power savings achieved by a good binding. This aspect of ORINOCO® is under development at this point.
Synopsys' PowerCompiler enhances power efficiency by approximately 30%. This is well within the expected range of 20 to 50% (Figure 5). PowerCompiler adds additional cells to the design to reduce power. Please refer to [10] for more information on this topic. The cellcount is nearly tripled. This sounds dramatic but area does increase only by approximately 12%.
For the next test we changed the number representation from two's complement to signed magnitude. The filters of the GFB swing around a neutral axis. Especially for small signals this induces a lot of activity into the design when two's complement is used. We clarify this in Figure 8.



Decimal	Twos Complement	Signed Magnitude
-1	1 1 1 1 1 1 1 1	1 0 0 0 0 0 0 1
1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1
-1	1 1 1 1 1 1 1 1	1 0 0 0 0 0 0 1
1	0 0 0 0 0 0 0 1	0 0 0 0 0 0 0 1

Figure 8: Two's Complement versus Signed Magnitude

The first three lines of Table 9 contain architectures known from Table 7. The column "optimized power" lists the by Power Compiler optimized designs. Power Compiler is able to reduce power dissipation by approximately 30% as seen before.
The mid three lines stand for a signed magnitude implementation. Since Synopsys' DesignWare does not offer arithmetic units for this number representation we had to implement our own library called DWSL. The power estimation results are disappointing on first sight. Compared to the two's complement implementation, power was reduced by 2% only. Optimization with PC then brings the surprise. If we compare the values from the "power" column with those of the "optimized power" column of Table 9, we see that the power dissipation is reduced by up to 41%. We did not spend much effort to implement our DWSL components. It seems they are too inefficiently implemented to transpose the benefit of signed magnitude directly.
In the last three lines of Table 9 it is shown that a "good binding" improves the power balance by another 6%, so that the most power efficient design consumes 25 mW.

6 Conclusions and Recommendations

In this paper we evaluated a behavioral level low power design flow and its applicability based on a design case. In chapter 5.1 we showed that simulation period and simulation stimuli have to be well chosen for the estimation of accurate power values. For our example a period of 3.5 ms and a speech input stream proved to be the best choice between estimation accuracy and simulation performance.
In chapter 5.2 we confirmed our statement from [3]. Synopsys' pipelined DesignWare multipliers are not the first choice when it comes to low power design. The architectures with these components consume 20% to 30% more power then those with a single staged multiplier.

We tried to get the most out of low power techniques in chapter 5.3. We learned that the architectures allocating the least resources do not necessarily consume the least power. We experienced Synopsys' PowerCompiler and OFFIS' ORINOCO® to be an efficient team for low power design. In this context we have to accentuate that the "best binding" low power methodology does not necessarily have any penalty on area. The GFB has weak constraints on timing and latency so that one multiplier resource is sufficient. The "best binding" methodology therefore requires additional hardware. If e.g. another design case requires several resources, the "best binding" can be applied without any extra cost on area.
We see in the wide range – 25mW to 83mW – of power dissipation estimates for our GFB algorithm that awareness of power is important. These values include a factor of 3.3 which punctuates Figure 5. We expect that operand isolation and clock gating will reduce power dissipation below 25mW – widening the range even further.
By moving up one abstraction level, from RTL to behavioral, we also improved the IC designers' productivity. The behavioral specification, be it VHDL or SystemC, implements the algorithm only. Construction of architectures is being automated and is no longer business of the designer.

Appendix A Results

		power [mW]				resources			area	
	simulated time	**3.5 ms**	**7.0 ms**	**10.5 ms**	**14.0 ms**	**adder**	**addsub**	**multiplier**	**cellcount**	**[mm²]**
architecture	**GFB**	42.334	43.355	43.702	43.836	5	1	4	1651	3.52
	pipe2_GFB	59.416	60.731	61.158	61.300	4	1	4	1594	3.68
	pipe3_GFB	56.765	58.111	58.540	58.690	4	1	3	1581	3.30
	pipe4_GFB	57.660	58.844	59.210	59.360	3	2	2	1399	2.95
	pipe5_GFB	59.745	61.058	61.466	61.640	3	2	2	1697	3.05
	pipe6_GFB	65.997	67.142	67.510	67.669	4	1	2	1585	3.07

Table 1: Power Estimates for Sinus Stimulus

		absolute relative error [%]				resources			area	
	simulated time	**3.5 ms**	**7.0 ms**	**10.5 ms**	**14.0 ms**	**adder**	**addsub**	**multiplier**	**cellcount**	**[mm²]**
architecture	**GFB**	3.43	1.10	0.31	-	5	1	4	1651	3.52
	pipe2_GFB	3.07	0.93	0.23	-	4	1	4	1594	3.68
	pipe3_GFB	3.28	0.99	0.26	-	4	1	3	1581	3.30
	pipe4_GFB	2.86	0.87	0.25	-	3	2	2	1399	2.95
	pipe5_GFB	3.07	0.94	0.28	-	3	2	2	1697	3.05
	pipe6_GFB	2.47	0.78	0.23	-	4	1	2	1585	3.07

Table 2: Absolute Relative Error for Sinus Stimulus

		power [mW]				resources			Area	
	simulated time	**3.5 ms**	**7.0 ms**	**10.5 ms**	**14.0 ms**	**adder**	**addsub**	**multiplier**	**cellcount**	**[mm²]**
architecture	**GFB**	44.134	44.201	44.392	44.430	5	1	4	1651	3.52
	pipe2_GFB	62.014	62.147	62.357	62.411	4	1	4	1594	3.68
	pipe3_GFB	59.202	59.281	59.538	59.597	4	1	3	1581	3.30
	pipe4_GFB	59.906	59.944	60.132	60.179	3	2	2	1399	2.95
	pipe5_GFB	62.436	62.339	62.545	62.595	3	2	2	1697	3.05
	pipe6_GFB	68.348	68.339	68.489	68.531	4	1	2	1585	3.07

Table 3: Power Estimates for White Noise Stimulus

		absolute relative error [%]				resources			area	
	simulated time	3.5 ms	7.0 ms	10.5 ms	14.0 ms	adder	addsub	multiplier	cellcount	[mm²]
architecture	GFB	0.67	0.52	0.09	-	5	1	4	1651	3.52
	pipe2_GFB	0.64	0.42	0.09	-	4	1	4	1594	3.68
	pipe3_GFB	0.66	0.53	0.10	-	4	1	3	1581	3.30
	pipe4_GFB	0.45	0.39	0.08	-	3	2	2	1399	2.95
	pipe5_GFB	0.25	0.41	0.08	-	3	2	2	1697	3.05
	pipe6_GFB	0.27	0.28	0.06	-	4	1	2	1585	3.07

Table 4: Absolute Relative Error for White Noise Stimulus

		power [mW]				resources			area	
	simulated time	3.5 ms	7.0 ms	10.5 ms	14.0 ms	adder	addsub	multiplier	cellcount	[mm²]
architecture	GFB	44.130	44.110	44.047	44.014	5	1	4	1651	3.52
	pipe2_GFB	61.983	61.874	61.737	61.654	4	1	4	1594	3.68
	pipe3_GFB	59.311	59.183	59.057	58.977	4	1	3	1581	3.30
	pipe4_GFB	59.906	59.753	59.622	59.562	3	2	2	1399	2.95
	pipe5_GFB	62.493	62.229	62.044	61.951	3	2	2	1697	3.05
	pipe6_GFB	68.354	68.227	68.087	68.026	4	1	2	1585	3.07

Table 5: Power Estimates for Speak Sample Stimulus

		absolute relative error [%]				resources			area	
	simulated time	3.5 ms	7.0 ms	10.5 ms	14.0 ms	adder	addsub	multiplier	cellcount	[mm²]
architecture	GFB	0.26	0.22	0.07	-	5	1	4	1651	3.52
	pipe2_GFB	0.53	0.36	0.13	-	4	1	4	1594	3.68
	pipe3_GFB	0.57	0.35	0.14	-	4	1	3	1581	3.30
	pipe4_GFB	0.58	0.32	0.10	-	3	2	2	1399	2.95
	pipe5_GFB	0.87	0.45	0.15	-	3	2	2	1697	3.05
	pipe6_GFB	0.48	0.30	0.09	-	4	1	2	1585	3.07

Table 6: Absolute Relative Error for Speech Stimulus

			resources				area	
		power [mW]	mult	add	addsub	sub	cellcount	[mm²]
one mult	1-stage-mult	41.087	1	1	1	-	1450	2.31
	2-stage-mult	51.946	1	1	1	-	1445	2.46
	3-stage-mult	52.010	1	1	1	-	1496	2.49
	4-stage-mult	52.934	1	1	1	-	1397	2.46
	5-stage-mult	54.395	1	1	1	-	1475	2.53
	6-stage-mult	51.646	1	1	1	-	1541	2.50
two mults	1-stage-mult	46.874	2	1	1	-	1673	2.86
	2-stage-mult	59.886	2	1	1	-	1705	2.98
	3-stage-mult	59.716	2	2	1	-	1714	3.03
	4-stage-mult	63.941	2	1	1	-	1555	2.96
	5-stage-mult	65.528	2	1	1	-	1493	2.93
	6-stage-mult	65.928	2	2	1	-	1627	3.00
three mults	1-stage-mult	53.267	3	3	1	1	1825	3.36
	2-stage-mult	61.527	3	1	1	-	1779	3.39
	3-stage-mult	66.068	3	1	1	-	1705	3.41
	4-stage-mult	69.889	3	1	1	-	1735	3.46
	5-stage-mult	75.003	3	1	1	-	1645	3.43
	6-stage-mult	82.856	3	2	1	-	1800	3.69

Table 7: Power Estimates for Alternative Architectures

			resources				area		optimized	area	
			mult	add	sub	power [mW]	cellcount	[mm^2]	power [mW]	cellcount	[mm^2]
ORINOCO®	good bind	1-stage-mult	2	7	1	53.769	1894	3.30	37.830	5217	3.71
		1-stage-mult	3	20	4	50.437	1766	3.50	32.895	5127	3.94
		1-stage-mult	3	7	1	46.603	1968	3.49	31.376	5480	3.93
		1-stage-mult	4	7	1	47.025	2019	3.82	30.717	5878	4.25
	bad bind	1-stage-mult	2	8	1	52.260	2165	3.43	34.712	5505	3.84
		1-stage-mult	3	9	2	53.864	2227	3.75	35.802	5744	4.18
		1-stage-mult	3	8	1	54.462	2232	3.77	38.147	5936	4.18
		1-stage-mult	4	8	1	52.930	2346	4.05	36.163	6188	4.49

Table 8: Power Estimates for ORINOCO® Architectures

		resources					area		optimized	area	
		mult	add	addsub	sub	power [mW]	cellcount	[mm^2]	power [mW]	cellcount	[mm^2]
	two's	1	1	1	-	41.087	1450	2.31	29.325	2837	2.63
		2	1	1	-	46.874	1673	2.86	34.846	3770	3.18
		3	3	1	1	53.267	1825	3.36	34.820	4992	3.82
	signed	1	2	-	-	44.858	1537	2.46	28.036	3330	2.82
		2	2	-	-	45.960	1672	2.84	28.215	3723	3.17
		3	2	-	-	45.377	1722	3.17	26.613	4540	3.56
ORINOCO®		2	8	-	-	45.035	1887	3.07	27.412	4523	3.49
		3	3	-	-	43.470	1559	3.12	25.531	4683	3.57
		4	8	-	-	42.543	1695	3.64	24.988	5810	4.10

Table 9: Two's Complement versus Signed Magnitude

Appendix B References

[1] Roberto Ugioli, Emanuele Oreste Zagano, "How to Speed Up Finite Impulse Register With Latest Features of Behavioral Compiler", SNUG Europe, 2000

[2] Dr. Peter Nagel, Martin Leyh, Martin Speitel, Alexander Krebs, "Methodology for using Behavioral Compiler for the Development of an OFDM Demodulator", SNUG Europe, 2000

[3] Frank Poppen, Wolfgang Nebel, "Comparison of a RT and Behavioral Level Design Entry Regarding Power", SNUG Europe, 2001

[4] Lars Kruse, Eike Schmidt, Gerd Jochens, Ansgar Stammermann, Wolfgang Nebel, "Lower Bound Estimation for Low Power High-Level Synthesis", 13th International Symposium on System Synthesis (ISSS 2000), Madrid, Spain, pp.180-185, September 2000

[5] Lars Kruse, Eike Schmidt, Gerd Jochens, Wolfgang Nebel, "Low Power Binding Heuristics", PATMOS'99, pp. 41-50, Kos, Greece, 1999

[6] Lars Kruse, Eike Schmidt, Gerd Jochens, Wolfgang Nebel, "Lower and Upper Bounds on the Switching Activity in Scheduled Data Flow Graphs", International Symposium on Low Power Electronics and Design (ISLPED'99), pp. 115-120, San Diego, California, 1999

[7] "http://www.orinoco.offis.de" and " http://www.lowpower.de" respectively

[8] "v2000.11 BehavioralCompiler Modeling Guide", Synopsys Online Documentation

[9] "v2000.11 BehavioralCompiler User Guide", Synopsys Online Documentation

[10] "v2000.11 PowerCompiler Reference Manual", Synopsys Online Documentation

[11] Anand Raghunathan, Niraj K. Jha, Sujit Dey, "High-Level Power Analysis and Optimization", Kluwer Academic Publishers, 1998

[12] R. Patterson, I. Nimmo-Smith, J. Holdsworth, P. Rice, "An Efficient Auditory Filterbank Based on the Gammatone Function", Appendix B of SVOS Final Report: The auditory Filterbank, APU report 2341, 1987