Comparison of a RT and Behavioral Level Design Entry Regarding Power

Frank Poppen, Wolfgang Nebel

OFFIS Research Institute

Frank.Poppen@Offis.de

ABSTRACT

SoC designer face two main problems nowadays. Firstly, the complexity of ASICs is expected to double every 18 months as Moore’s Law did not loose its correctness yet. Higher levels of abstraction need to be introduced to handle the billions of transistors of future designs and to keep, or even better shorten, time to market. At the same point the designer of these sub micron devices will have to observe the power dissipation of his SoC. The continuous enhancement of battery’s energy capacity does not keep up with the more and more power consuming applications. This is critical to handheld products like cellular phones and PDAs.

RT level abstraction is the entry point in most designflows today. During synthesis a gate level netlist is being generated. If any, power estimations and optimizations usually are performed on these netlists. In the low power community it is well known that at this point most opportunities of higher level power optimizations are wasted. The better flow is the one with a behavioral level entry point and an early focus on power dissipation.

Synopsys offers its Behavioral Compiler which introduces this higher level of abstraction. By entering the algorithms only, the designer does not need to do needlework on scheduling and binding of operations. Behavioral Compiler even offers automated gated clock insertion which should result in reduced power dissipation.

As a design case, this paper introduces a bank of six fourth order IIR filters, which is implemented in VHDL code at RT level as well as behavioral level. The differences of the two specifications are explained and the dissimilar design flows are elucidated. The outcome of both flows is a gate level mapped design which is, then, analyzed in terms of power dissipation. By iterating both flows, while applying power optimizations to the VHDL and power constraints to the synthesis scripts, the synthesized gate level netlists are improved regarding power. The reader will hereby undergo the advantages and limits of each methodology.

1 Introduction

During the early days of software engineering, programmers had to code algorithms in hardware near languages like assembler. The results where fast and efficient little routines for a single processor. But the code was useless for any other architecture. For a new processor the engineer had to start all over again. This resulted in the 1960th software crisis which was overcome with the first compilers. For general purpose IP, it is unthinkable to code nowadays software in assembler. EDA industry reached a similar point where designers have more and more difficulties to handle the fast advancing technologies. “IP reuse” is the buzzword that comes up with most methods of solutions, but like software coded in assembler, today’s designs are not implemented for reuse. This is not only necessarily the designers fault, but also has to be blamed on the designflows that are used. Shifting the entry point to a higher level – from RT to behavioral – might be the solution. Synopsys offers the Behavioral Compiler (BC) for this.
Like programmers had to be convinced that compilers can generate efficient machine code, behavioral synthesis has to prove its advantage to other methodologies. In [1] and [2] it was shown that BC is able to replace a RT level design entry and even improve performance.
Therefore this paper does not handle the aspect of timing and area but focuses on the aspect of power dissipation in architectures generated with BC. Equally, we ignore aspects of testability to reduce the complexity of the design space. The question we want to answer is: “Is BC able to synthesize power efficient designs?”
In the following chapter, we want to give a quick explanation of the used design. We introduce the algorithms and the top level architecture. In two sub chapters, 2.1 and 2.2, we lead into the different coding styles for BC and DC. In chapter 3 the two design flows for each entry, behavioral and RT, are depicted. In chapter 4 we set the basis for a accurate gate level power estimation. We make sure to simulate the design with appropriate input vectors in order to achieve the needed activity information for power estimation. Chapter 5 resembles the core of this paper. Each case study is explained and analyzed. The conclusions and recommendations are summarized in chapter 6. The paper closes with the references in chapter 7.

2 Design Case

The design case we have chosen is a bank of six IIR fourth order gammatone filters to explore the design space. It is an ideal case because it contains typical components of a SoC: controller, a data flow path, a ROM block and a RAM block.
The algorithm of one IIR is shown in Figure 1. We see a data flow that 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 the mentioned ROM and eight previously calculated values (Real1 to 4 and Imag1 to 4) that are kept in the RAM.
We chose to divide the design into two main parts. The *channel* implements the data flow of Figure 1 while the *controller* is described as a separate entity. The *controller* reads the filter constants and the previously calculated values from ROM and RAM. After reading it then serves these to the *channel* and writes back the results into the RAM. The communication between these two entities drives a handshake protocol. Figure 2 shows the quite simple structure of the toplevel.

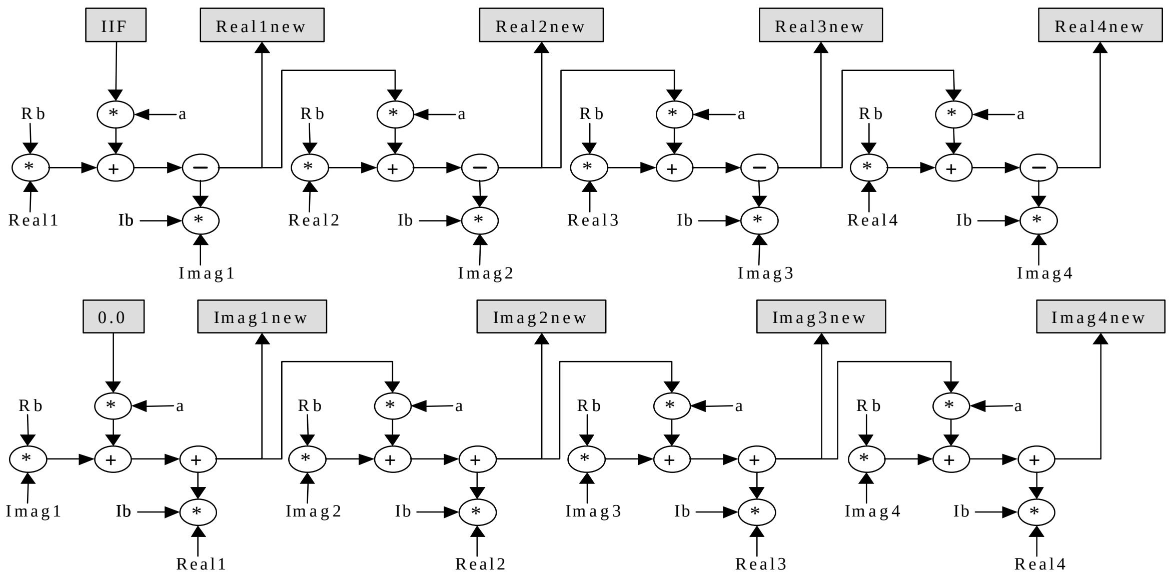

Figure 1: Algorithm of Channel

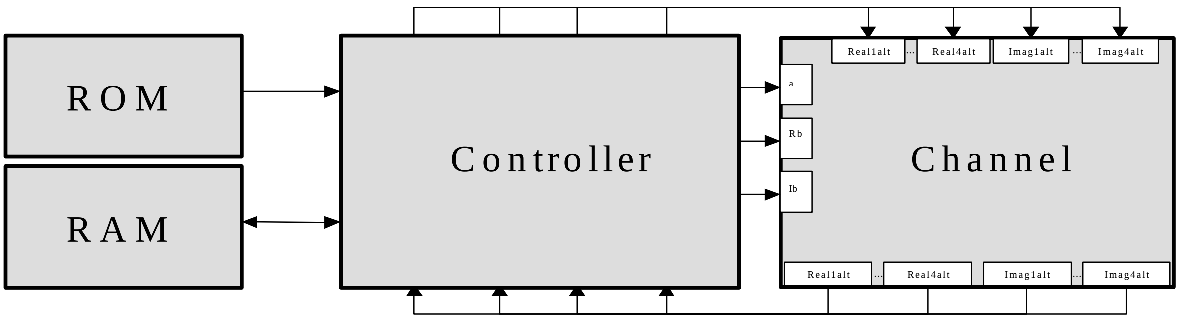

Figure 2: Top Level Block Diagram

2.1. RT VHDL

Since RT level coding is the most common design methodology we handle this topic short. RAM and ROM are instantiated from a package that is generated by the silicon vendor's memory generator. A 16 Bit two stage multiplier and addersubtractor are instantiated from Synopsys' designware foundation library. Processes have the following structure:

```
label : process(clk)
begin
  if clk'event and clk = '1' then
    if reset = '1' then                          --Do reset
    else                                         --Normal operation
    end if;
  end if;
end process;
```

The RT level specification excluding testbench consists of 1280 lines of code.

2.2. Behavioral VHDL

No instantiations where used for behavioral VHDL coding. BC is capable to find the right allocation, binding and scheduling for the given algorithm and its operations. Multiplications and additions or substractions are mapped to Synopsys' designware foundation library automatically. To make the RAM and ROM accessible to BC we used Synopsys' MemWrap and applied the methodology described in [8]. The following code maps variables to the memory cells:

```
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";
```

The structure of the processes is different as well. They consist of two infinite loops – the reset and main loop. Have a look at the following example.

```
p_gfb_kanal : process
begin
    reset_loop: loop
                ...                                  -- Reset all signals
            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_kanal;
```

It is important to know that each '*wait until Clk*' statement followed by an '*exit loop*' defines a so called super state. Super states might take more then one clock cycle to execute. More details can be found in [7].

The behavioral level specification excluding testbench consists of 636 lines of code.

3 Used Designflow

We want to compare two different design methodologies. One starts with a RT level VHDL description of an exemplary design. The other requires a behavioral level VHDL implementation of the same device, which is processed by Synopsys' Behavioral Compiler (BC). Each methodology requires its own design flow. They are drafted in Figure 3.

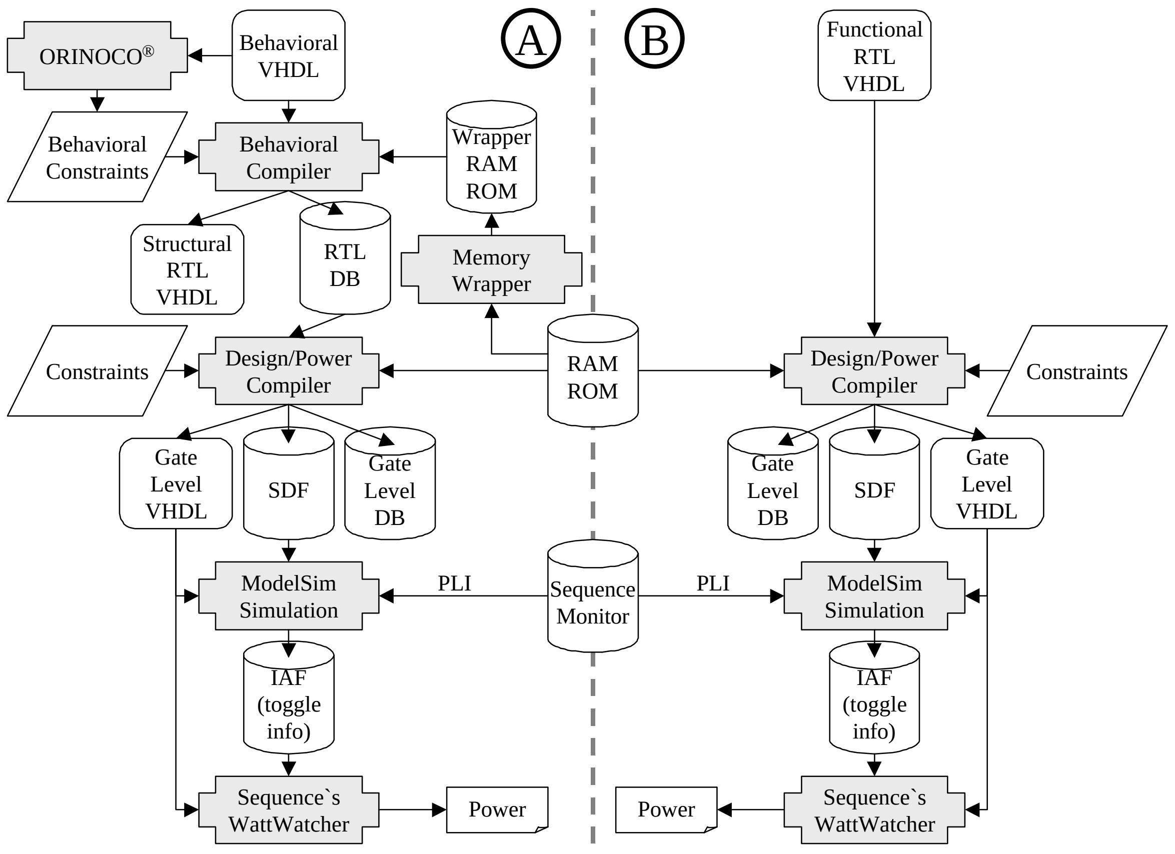

Figure 3: Used Designflow

The left part (A) of Figure 3 shows the behavioral design entry. The input is behavioral VHDL which implements two main loops (compare with 2.2).
Several script files have been written to explore the design space. It is very easy to have BC find an allocation, scheduling and binding for a desired functionality. By changing a few lines in the script file, BC will allocate several multipliers and adders or just one. Our in-house developed tool ORINOCO® supports the designer to find appropriate low power constrains for BC. Chapter 5.8 provides more details on this topic.
BC drives the user defined RAM and ROM with the help of a wrapper generated by Synopsys' tool Memory Wrapper. Reference [8] 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 a functional verification (not drafted in Figure 3). At the same time we save the design in a Synopsys' Data Base (DB) file which serves as input for Design Compiler (DC) and Power Compiler (PC) respectively.
The next step in the flow 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.
The result is a gate level netlist of the design which is written out to VHDL and DB. PC's features clock gating and operand isolation are employed to improve the energy balance. Have a look at [9] for a more detailed description of the latter topic.

Hereafter the signal activity is being traced during a ModelSim VHDL simulation. This is made possible by an entity named *sente_monitor_mit* that is instantiated in the testbench. Via the simulators PLI interface necessary toggle information is written into an Intermediate Activity File (IAF).
The power estimation tool WattWatcher from Sequence finally analyzes the VHDL and annotates the activities to each cell to sum up total power usage. The cell power models are supplied by LSI Logic in an Advanced Library Format (ALF).
The RTL entry design flow – right part (B) of Figure 3 – is equivalent to the back end of the first flow (A).
Finally the estimated power results of each flow are compared and analyzed.

4 How to Simulate

As already mentioned, WattWatcher is depended on activity information to estimate power. A gate level VHDL simulation supplies this data. It is essential for the quality of the estimated power results to simulate a common case. Two questions needed to be solved: What data should be processed by the GFB and how much time should be simulated?

4.1. What to Simulate

The testbench supplies two sinuses with frequencies of 0.28936 KHz and 1.29607 KHz to the filters using the left and right channel of the GFB. The frequencies are properly chosen to generate a characteristic activity in each filter of the GFB. Figure 4 and Figure 5 show the filters' responses to these sinuses. The time axis is quantified with the input sampling frequency which is 16.276 KHz.
The first stimulus excites the first band pass filter most since it is its mid-frequency. The second stimulus comes close to the mid-frequency of the second filter.

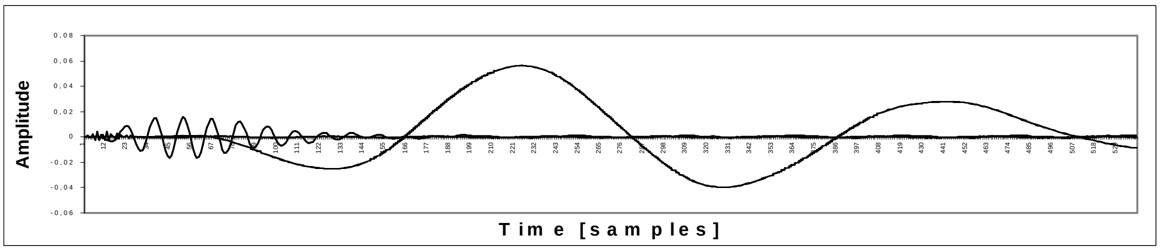

Figure 4: Response of GFB to 0.28936 KHz sinus

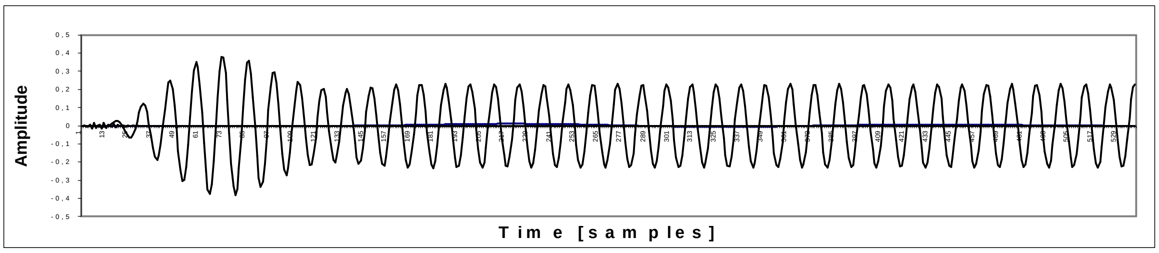

Figure 5: Response of GFB to 1.29607 KHz sinus

4.2. How long to simulate

Our introduced design flow shows the need of gate level simulation. We had to find a compromise between the necessity to simulate a long period of real time and to keep simulation time down. Only when "enough" samples of the input sinus are processed the power estimation will be meaningful. To get hold on the indistinct term "enough" we simulated an exemplary architecture similar to the one introduced in chapter 5.1 – containing ROM, RAM, five 2-stage-pipelined-multipliers, five adders and one subtractor – with different runtimes. The results are listed in Table 1.

Total power combines clock and internal power[1]. While clock power predicts the power dissipation due to driving the clock tree, internal power summarizes the loss in each leaf cell of the design. The second half of Table 1 shows the power usage of groups of such cells that form the structures RAM/ROM[2], multipliers and adders/subtractors.

During the first µs the design is resetting and the RAM is being initialized. This is reflected by the first column of Table 1. It shows a 13.9% higher power dissipation in the memory compared to typical operation mode. Clock power is constant in the various simulation cases. This is not surprising given that clock gating is not applied in the simulated instance.

One can see that the design's total power consumption does not change at all when a period of 0.01 ms or more is being simulated. When selecting the time period with a length of 1.5 ms, we are accepting an error of 4.4% in the adder/subtractor logic compared to the more accurate estimation with 3 ms of toggle information. Judging the share of these logic for total power (0.1%) and used processor time (~ 0.66 h instead of ~ 1.3 h on a SunUltra 440 MHz with 768 MB of RAM) we accepted 1.5 ms simulated real time as "enough".

During this period the design completes a reset and toggles between operational and idle mode 25 times or processes 25 samples respectively.

Simulated Time [ms]	**0.001**	**0.01**	**1**	**1.5**	**2**	**2.5**	**3**
Total Power [mW]	123.00	110.00	110.00	110.00	110.00	110.00	110.00
Clock Power [mW]	12.00	12.00	12.00	12.00	12.00	12.00	12.00
Internal Power [mW]	111.00	97.8	98.30	98.40	98.50	98.50	98.50
Mem Power [mW]	51.65	45.92	45.35	45.35	45.35	45.35	45.35
Mult Power [mW]	20.29	16.85	17.80	17.91	17.95	17.96	17.98
AddSub Power [mW]	0.13	0.13	0.11	0.11	0.12	0.12	0.12

Table 1: Estimated Power for Different Simulation Times

5 Analysis

The tables of Appendix A show the estimated power of exemplary designs. They differ in the number of allocated arithmetic units and applied low power methodologies like clock gating or operand isolation. In the following chapters 5.1 to 5.10 we give a brief description on each design. Then we evaluate and compare the results. We set of from a default implementation.

[1] Usually pad power would be a third source but we did not insert pads in the design.
[2] Actually RAM and ROM are represented by one large vendor cell each.

5.1. Default

We started from a naively implemented VHDL code for behavioral and another for RT level entry without considering any low power aspects. In both flows this resulted in unnecessary memory accesses and therefore high power dissipation. BC implements the architecture using five two-stage-multipliers, five adders and one subtractor. DC is not able to schedule pipelined designware components. For this reason, we had to analyze timing requirements manually and then chose to instantiate one two-stage-multiplier and one addersubtractor for the RT implementation. This makes it understandable that the RT design uses 11.5% less power compared to the behavioral design (see formula 1). It has to be noted that several arithmetic units do not necessarily consume more power then one implementation as will be shown in 5.6 to 5.10.

$$\frac{BAVpower - RTpower}{BAVpower} \cdot 100 \qquad (1)$$

5.2. Optimized RAM access

The controller has been optimized to eliminate unnecessary memory accesses. In the behavioral flow this has been achieved due to modifications to the memory wrapper. Owing to the comfortable GUI of this tool, this took just a few mouse clicks and another run of scheduling and compilation. For the RT level flow the VHDL code of the controller had to be modified. Compared to the change in the behavioral flow, this was quite a complex interference with the VHDL code. In this way we where able to reduce power dissipation in RAM and ROM by 96.2% in the behavioral and by 94.9% in the RT flow (see formula 2).

$$\frac{defaultMEMpower - lowRAMpower}{defaultMEMpower} \cdot 100 \qquad (2)$$

All following cases integrate this optimized controller.

5.3. Extend Latency

This architecture only exists for the behavioral flow. The controller remained unchanged as in 5.2. For scheduling the option *extend_latency* has been activated. This resulted in a channel implementation with one two-stage-mult, one adder and two addersubtractors. This reduced power compared to the default design by 24.5%.

5.4. Set Cycles

Even with the extend latency option set, the behavioral flow creates an unnecessarily fast design. We calculated that – given the sample frequency of 16.276KHz – each filter has a maximum time window of 960ns to complete its calculations. The loop which implements the channel might take up to 48 clock cycles to be executed. Therefore we introduce the command *set_cycles 48 –from_begin ... –to_end* before scheduling the channel. The newly scheduled architecture contains one two-stage-multiplier and one addersubtractor. It is directly comparable to the default design of the RT level flow which implements the same number of arithmetic units. It can be seen that the power dissipation of the resulting designs differs only by 6.7%, leaning towards the favor of the behavioral methodology.

5.5. Binary

In both flows the state encoding style was changed from one-hot to binary. This was achieved by setting the variable *bc_fsm_coding_style = "binary"* in the behavioral flow. In the RT level flow the VHDL code was modified. It might have been possible to use the design compilers ability to extract a state machine from the design but since this is a gate level technique it is applicable to both flows. There is no need for a comparison.

This reduced the width of the state-vectors from 30 to 70 bits to below 10 bits. Gray encoding was preferred but BC does not support this encoding style, which we think is a feature that should be implemented. Power consumption was reduced by 6.1% (behavioral) and 6.3% (RT) respectively as a result of shrinking the bit width of the state registers.

5.6. Mult

For both design flows we replaced the pipelined multiplier dw02_mult_2_stage(str[3]) with a dw02_mult(nbw[4]). In 5.8 we determine this step. It is amazing that the multiplier consumes about 69,5% less power than the pipelined version. It would be interesting to see if the implementation dw02_mult_2_stage(csa[5]) is a better choice. Unfortunately, this could not be examined in this paper. In the following experiments we continued to use the dw02_mult(nbw).

5.7. Clock Gating

We applied power compiler's ability of gated clock insertion to the two flows. In the behavioral flow the synthesis scripts had to be enhanced with the following:

```
set_clock_gating_style -sequential latch -min 4
set_fix_hold SlClk
set_input_delay 0 -clock SlClk { Signal1,...,SignalN }
set_clock_transition 0 SlClk
schedule -gate_clock
propagate_constraints -gate_clock
compile
```

The RT flow used the following:

```
set_clock_gating_style -sequential latch -min 4
set_fix_hold SlClk
set_input_delay 0 -clock SlClk { Signal1,...,SignalN }
propagate_constraints -gate_clock
set_clock_skew SlClk –propagate
compile
```

Please refer to [9] for the denotation of each command. Clock gating reduced power consumption by 31.3% (behavioral) respectively 21.5% (RT). Unexpectedly, the behavioral design is marginally smaller than the one without the extra clock gating logic from 5.6. A possible remedy might be different default variable settings and algorithms within the synthesis flow due to the use of PC.

5.8. Operand Isolation

We applied power compiler's ability of operand isolation to the flows. Employing the same methodology for RT flows is possible but it came out that using instantiations for Designware

[3] Pipelined Wallace tree multiplier synthesis model
[4] None-Booth-recoded Wallace tree synthesis model
[5] Carry-save array synthesis model

components is fatal. As described in [9] PC offers two methods for operand isolation integration. The *pragma isolation method*, that has been used for the behavioral flow, was not appropriate for the RT flow since the VHDL code did not contain binary operations to which the pragma could be assigned. The RT VHDL Code needed to be rewritten with operand inferencing. Unfortunately, the pipelined designware multipliers can only be inferenced by BC. This required another recoding and the usage of none pipelined multipliers (compare with 5.6).
Operand isolation disconnects the inputs of an arithmetic unit by the use of either AND or OR gates. The AND gate technique switches the input vectors to all zeros when an arithmetic unit is unused while the OR gate methodology switches them to all ones.
Table 4 shows that this methodology is not applicable for the design case. Enabling and disabling operand isolation induces extra activity and extra power dissipation. In other scenarios the benefit of operand isolation might outweigh this handicap.

5.9. ORINOCO

In 5.1 we mentioned that using as few arithmetic units as possible is not necessarily the most power efficient design methodology. The references [3] to [5] elucidate the theories behind this statement. Currently we work on the implementation of a behavioral level power estimation tool named ORINOCO® [6], which supports the designer in finding the right scheduling, binding and allocation. ORINOCO®'s suggestion for our exemplary design is shown in Figure 6.

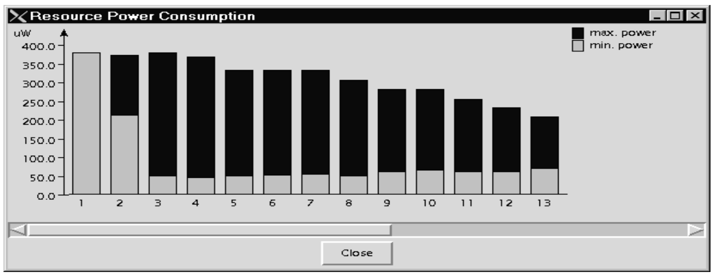

Figure 6: ORINOCO®

To achieve minimum power in the multipliers, ORINOCO® suggests to use three instances with a certain binding of operations to these. The binding is essential for this optimization since a poor one will eliminate the advantage. This can be seen in Figure 6. The three-multiplier-bar with an optimal binding (light gray) indicates a low power design while implementing the worst binding (black) would just result in a waste of area for the extra hardware.
We used the following scheduling command three times to have BC implement the design as proposed by our tool:

```
set_common_resource { \
    p_gfb_kanal/reset_loop/operational_loop/mult_piped_95/mul_24 \
          ...
    p_gfb_kanal/reset_loop/operational_loop/mult_piped_167/mul_24 \
} -min_count 1 -max_count 1
```

It takes high effort to integrate the effect of this command into the RT level flow. This is the reason why we didn't try to generate an equivalent design from the RT level entry.
The result of the power optimization can be seen in Table 5. The three multipliers actually do need less power then the one of the design Mult (0,741 mW compared to 0,855mW). This is an improvement by 13.3%. The theory therefore does show a practical relevance. Unfortunately, total power is slightly higher (1.3%). The explanation is quite simple: The effect of control logic

like multiplexers and finite state machines on power dissipation is the subject of current research and has to be implemented in ORINOCO® yet. The control logic of the new design consumes more power then the one from the design Mult. The favorable balance of the multipliers is inferior to the negative effect of the control logic.

5.10. Worst

We tried the worst possible binding suggested by ORINOCO® as well and again the result matches the theory. Table 5 shows that the same three multipliers consume now 0.934 mW. This corresponds to a degradation of 26.0% compared to the best binding. But again total power does not reflect this result. Amazingly this design consumes marginally less power then the one with only one mult. To draw a conclusion: Examining the arithmetic units only is not enough.

6 Conclusions and Recommendations

In 1 we put the question: “Is BC able to synthesize power efficient designs?” We can answer this with a conservative “yes”. We do so because BC does not offer any special power constraints or optimizations at behavioral level. The applied techniques like e.g. clock gating and operand isolation are not exclusively for BC. They are offered by Synopsys’ tool power compiler which can be integrated in the RT level flow, too. This is the reason why both flows offer good power reduction possibilities and produce nearly similar results. It might seem that BC does generate slightly less power consuming designs but with some manual enhancement to the RT level VHDL code, the results may be the same.

There are three reasons why we still prefer the behavioral level flow:

1. it might not enhance power efficiency but it doesn’t harm either.
2. the development of new tools like ORINOCO® for high level power optimization – once they are no longer prototypes or research studies – promise an advantage of BC over DC in the future.
3. it is much simpler to introduce power optimizations at the behavioral level.

The RT level VHDL code had to be modified several times during the design space exploration, which took some time to do and added the permanent risk of inserting bugs. The behavioral VHDL code stayed unchanged except for two synthesis pragmas. Since pragmas are just comments this does not alter the specification of the design at all. Every other optimization was applied by modifying the scheduling and synthesis scripts.

7 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] 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

[4] Lars Kruse, Eike Schmidt, Gerd Jochens, Wolfgang Nebel, “Low Power Binding Heuristics”, PATMOS’99, pp. 41-50, Kos, Greece, 1999

[5] 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

[6] “http://www.orinoco.offis.de” and “ http://www.lowpower.de” respectively[7] “v2000.11 BehavioralCompiler Modeling Guide”, Synopsys Online Documentation

[8] “v2000.11 BehavioralCompiler User Guide”, Synopsys Online Documentation

[9] “v2000.11 PowerCompiler Reference Manual”, Synopsys Online Documentation

8 Appendix A

Architecture	Default	Optimized RAM	Extend Latency	Set Cycles	Binary
Area [mm²]	4.60	4.62	2.79	2.71	2.62
Internal Power [mW]	98.40	53.00	40.10	38.50	36.30
Clock Power [mW]	12.00	12.00	8.97	8.67	8.02
Total Power [mW]	110.00	65.00	49.10	47.20	44.30
sum of Mults [mW]	17.9100	17.9100	4.7800	2.8000	3.3200
sum of Add&Sub [mW]	0.1457	0.1457	0.1787	0.1120	0.1320
RAM [mW]	45.3000	1.6700	1.6600	1.6500	1.6500
ROM [mW]	0.0530	0.0530	0.0527	0.0523	0.0523
2_stage_Mult 1 [mW]	*4.1000*	*4.1000*	*4.7800*	*2.8000*	*3.3200*
2_stage_Mult 2 [mW]	*3.4800*	*3.4800*			
2_stage_Mult 3 [mW]	*3.5600*	*3.5600*			
2_stage_Mult 4 [mW]	*2.0600*	*2.0600*			
2_stage_Mult 5 [mW]	*4.7100*	*4.7100*			
Add 1 [mW]	*0.0147*	*0.0147*	*0.0523*		
Add 2 [mW]	*0.0173*	*0.0173*			
Add 3 [mW]	*0.0123*	*0.0123*			
Add 4 [mW]	*0.0313*	*0.0313*			
Add 5 [mW]	*0.0311*	*0.0311*			
Sub 1 [mW]	*0.0130*	*0.0130*			
AddSub 1 [mW]	*0.0130*	*0.0130*	*0.0411*	*0.1120*	*0.1320*
AddSub 2 [mW]	*0.0130*	*0.0130*	*0.0853*		

Table 2: Designs Behavioral Entry

Architecture	Default	Optimized RAM	Binary
Area [mm²]	2.80	2.02	1.92
Internal Power [mW]	88.30	41.50	38.90
Clock Power [mW]	9.03	9.05	8.57
Total Power [mW]	97.40	50.60	47.40
RAM [mW]	45.3000	2.3100	2.3100
ROM [mW]	0.0167	0.0167	0.0167
2_stage_Mult [mW]	3.8800	3.8800	3.0500
AddSub [mW]	0.0810	0.0810	0.0810

Table 3: Designs RT Level Entry

	Bahavioral Flow: lowRAM_ ...				RT Level Flow: lowRAM_...			
Architecture	**Mult**	**Clock Gating**	**Isolation (or)**	**Isolation (and)**	**Mult**	**Clock Gating**	**Isolation (or)**	**Isolation (and)**
Area [mm²]	1.86	1.81	1.87	1.88	1.85	1.86	1.81	1.81
Internal Power [mW]	36.70	26.80	37.00	36.60	37.10	30.40	36.60	36.20
Clock Power [mW]	8.31	5.78	8.31	8.31	8.08	6.52	7.88	7.88
Total Power [mW]	45.00	32.50	45.30	45.00	45.2	36.90	44.50	44.10
RAM [mW]	1.6600	1.6600	1.6600	1.6600	2.3100	2.3100	2.3100	2.3100
ROM [mW]	0.0527	0.0527	0.0527	0.0527	0.0167	0.0167	0.0167	0.0167
Mult [mW]	0.9170	0.9520	1.2500	0.9200	1.1300	0.84000	1.6400	1.2500
AddSub [mW]	0.1580	0.1650	0.1540	0.1540	0.1100	0.0806	0.1280	0.1280

Table 4: Clock Gating and Operand Isolation

Architecture	**ORINOCO**	**Worst**
Area [mm²]	2.52	2.51
Internal Power [mW]	39.10	38.00
Clock Power [mW]	8.86	8.58
Total Power [mW]	47.90	46.50
RAM [mW]	1.6500	1.6600
ROM [mW]	0.0523	0.0527
Mult 1 [mW]	0.2680	0.3780
Mult 2 [mW]	0.2580	0.3830
Mult 3 [mW]	0.2150	0.1730
all Mult [mW]	***0.7410***	***0.9340***
Add 1 [mW]	0.0396	0.0429
Sub 1 [mW]	0.0117	
AddSub 1 [mW]	0.0824	0.0754
all Add [mW]	***0.1337***	***0.1183***

Table 5: Applying ORINOCO® Suggestions