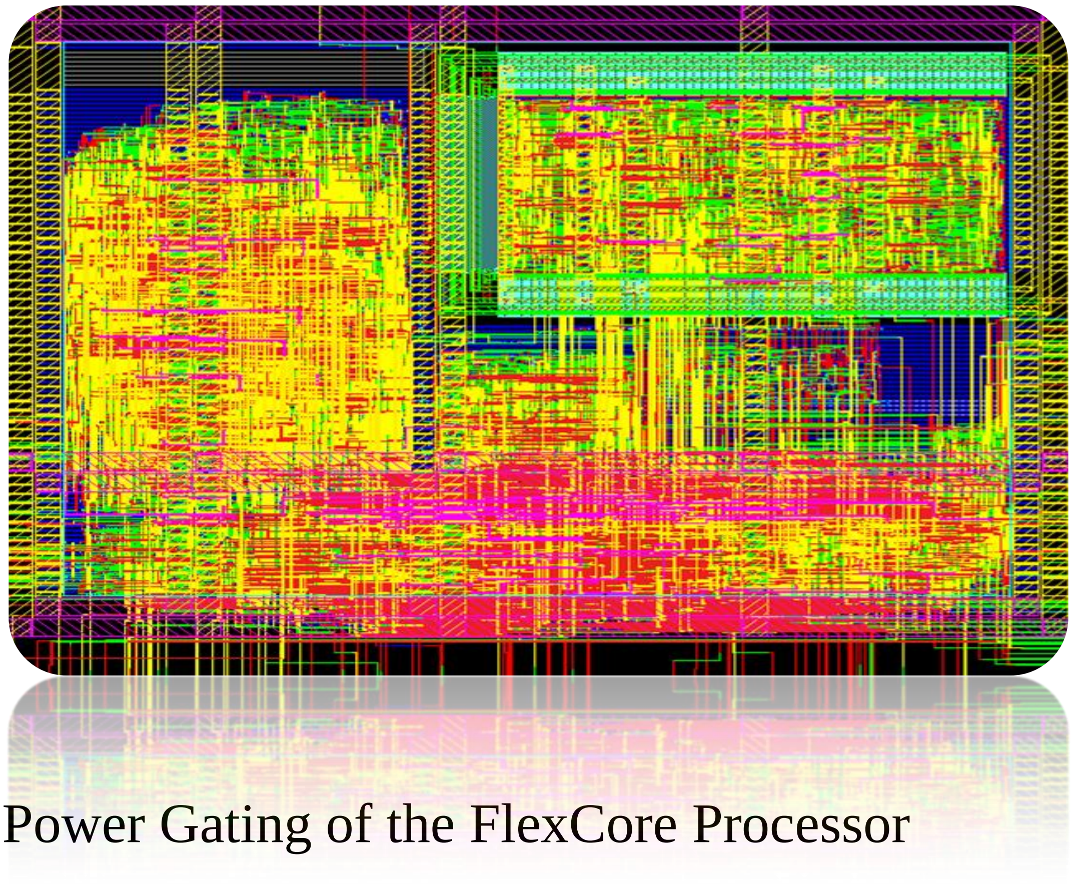


Power Gating of the FlexCore Processor

Master of Science Thesis in Integrated Electronic System Design

Vineeth Saseendran
Donatas Siaudinis

VLSI Research Group
Division of Computer Engineering,
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2010

The Authors grants to Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Authors warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors has signed a copyright agreement with a third party regarding the Work, the Authors warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Power Gating of the FlexCore Processor

Vineeth Saseendran and Donatas Siaudinis

© Vineeth Saseendran and Donatas Siaudinis, June 2010.

Examiner: Per Larsson-Edefors

VLSI Research Group
Department of Computer Science and Engineering
Chalmers University of Technology
SE- 412 96, Göteborg,
Sweden

Supervisor: Tung Thanh Hoang

VLSI Research Group
Department of Computer Science and Engineering
Chalmers University of Technology
SE- 412 96, Göteborg,
Sweden

Department of Computer Science and Engineering
Göteborg, Sweden,

Abstract

The aim of this master thesis work is to reduce the leakage power of the FlexCore processor by applying one of the most effective leakage reduction techniques, power gating. The main principle of this technique is inserting transistors named power switches, to cut off voltage supply of the functional units when they are not in use. In the context of this thesis, multiplier unit of the FlexCore processor, a novel architecture for embedded systems, is selected to be power gated. This is because, initial studies show that the multiplier, due to its relatively large size and significant idle time leads to it being a major contributor to the leakage power dissipation. A process of applying power gating onto the FlexCore's multiplier is divided into two parallel branches, software analysis and hardware implementation, and concluded in an integration phase. The software analysis phase, using FlexTools tool-chain, involves profiling of two EEMBC benchmarks and extending of the Native Instruction Set Architecture (N-ISA) to adopt control bits that are enable to activate or deactivate the multiplier unit on demand. The hardware implementation phase focuses on the implementation of the power gating technique by using power specification which is defined in the Common Power Format (CPF) at both RTL level and physical level. In the final phase, the extended N-ISA instruction is applied on the FlexCore processor with power gated multiplier unit to estimate the power reduction at a small area cost. During the physical implementation phase, the optimal power savings were estimated taking in to account the overhead from the switches. For the two examined benchmarks, energy efficiency was shown in range of 4-14%. In real applications, the multiplier is less active than in the benchmarks considered here and thus, it is possible to achieve higher energy reduction.

Table of Contents:

Abstract 3
Table of Contents: 4
List of Figures: 5
List of Tables: 6
List of Abbreviations 9
Tools and Technology 10
1. Introduction 11
   1.1 Motivation 12
2. Power Reduction 14
   2.1 Power Dissipation in CMOS Circuit 14
   2.2 Power Reduction Techniques 16
3. FlexCore Processor 19
   3.1 Background 19
   3.2 Baseline Architecture of the FlexCore 19
4. Power Gating for the FlexCore Processor 21
   4.1 Power gating the Multiplier 24
5. Software Analysis 27
   5.1 FlexSoC Tool-chain 27
   5.2 EEMBC Benchmark 28
   5.3 Localization of the Multiplier Function 29
   5.4 Tools Chain 30
   5.5 Evaluation of Multiplier Behaviour 31
   5.6 N-ISA Extension for Power Gating 38
6. Hardware Implementation 40
   6.1 Common Power Format 40
   6.2 RTL Design and Synthesis 43
   6.3 Power Control Module 46
   6.4 Physical Implementation 48
7. Results and Analysis 52
   7.1 Power Reduction Estimation after RTL Synthesis 52
   7.2 Power Reduction Estimation after Physical Implementation 55
8. Conclusion & Future Work 64
9. Bibliography 65
Appendix – A: Software Analysis Makefile 66
Appendix – B: FlexCore CPF 68
Appendix – C: Power Control Module 71

List of Figures:

FIGURE 1.1: TREND OF DYNAMIC AND LEAKAGE POWER FOR GENERAL PROCESSORS. 11
FIGURE 1.2: EFFECTS OF DIFFERENT POWER REDUCTION TECHNIQUES. 12
FIGURE 3.1: MULTIPLIER-EXTENDED FLEXCORE PROCESSOR 20
FIGURE 3.2: FLEXCORE N-ISA INSTRUCTION 20
FIGURE 4.1: HEADER AND FOOTER SWITCH PLACEMENT FOR POWER GATING A UNIT. 21
FIGURE 4.2: PLACEMENT OF POWER SWITCHES 22
FIGURE 4.3: ISOLATION CELLS INSERTED AT THE OUTPUT OF A POWER-GATED DOMAIN 23
FIGURE 4.4: STATE RETENTION FLIP-FLOP 23
FIGURE 4.5: POWER DOWN SEQUENCE 23
FIGURE 4.6: MULTIPLIER POWER GATED VIEW OF THE FLEXCORE 24
FIGURE 4.7: SWITCH CONTROL ARRANGEMENT 25
FIGURE 5.1: METHODOLOGY FLOW OF THE SOFTWARE ANALYSIS 28
FIGURE 5.2: MULTIPLICATION LOCALIZATION EXAMPLE IN THE AUTCOR APPLICATION C-CODE. 30
FIGURE 5.3: AUTCOR 'PROFILE' FILES. 32
FIGURE 5.4: AN EXCERPT FROM THE AUTCOR 'SHOWCODE' FILE 33
FIGURE 5.5: AN EXCERPT FROM THE AUTCOR 'READABLE' FILE 33
FIGURE 5.6: MULTIPLIER USAGE IN EEMBC BENCHMARK, AUTCOR. 35
FIGURE 5.7: MULTIPLIER USAGE IN EEMBC BENCHMARK, FFT. 36
FIGURE 5.8: N-ISA INSTRUCTION MAPPING 37
FIGURE 5.9: SEVERAL INSTRUCTIONS OF AUTCOR N-ISA CODE 37
FIGURE 5.10: APPLYING POWER GATING CONTROL SIGNAL 38
FIGURE 5.11: EXTENDED N-ISA 39
FIGURE 6.1: HARDWARE IMPLEMENTATION FLOW 40
FIGURE 6.2: CPF SPECIFICATION STRUCTURE 41
FIGURE 6.3: RTL SYNTHESIS FLOW 44
FIGURE 6.4: SCHEMATIC VIEW OF THE SYNTHESIZED NETLIST SHOWING ISOLATION CELLS HIERARCHY 45
FIGURE 6.5: SCHEMATIC VIEW OF THE ISOLATION CELLS IN THE HIERARCHICAL GROUP 46
FIGURE 6.6: NISA EXTENDED BITS FOR POWER CONTROL AND THEIR OPERATION 47
FIGURE 6.7: PHYSICAL IMPLEMENTATION FLOW 49
FIGURE 6.8: PHYSICAL VIEW OF THE MULTIPLIER DOMAIN, SWITCHES, ISOLATION CELLS AND POWER/GROUND NETS 51
FIGURE 7.1: POWER COMPARISON FOR THE MULTIPLIER, BENCHMARK: AUTCOR, LIB: GP-LVT 53
FIGURE 7.2: POWER COMPARISON FOR THE MULTIPLIER, BENCHMARK:AUTOCOR, LIB:GP-SVT 53
FIGURE 7.3: POWER COMPARISON FOR THE MULTIPLIER, BENCHMARK: FFT, LIB: GP-LVT 54
FIGURE 7.4: POWER COMPARISON FOR THE MULTIPLIER, BENCHMARK: FFT, LIB: GP-SVT 54
FIGURE 7.5: OVERALL POWER/ENERGY REDUCTION FROM THE ORIGINAL DESIGN 55
FIGURE 7.6: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:AUTCOR, LIB:GP-LVT, NOMINAL 57
FIGURE 7.7: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:AUTCOR, LIB:GP-LVT, WORST CASE 57
FIGURE 7.8: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:AUTCOR, LIB:GP-SVT, NOMINAL 58
FIGURE 7.9: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:AUTCOR, LIB:GP-SVT, WORST CASE 58
FIGURE 7.10: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:FFT, LIB:GP-LVT, NOMINAL 59
FIGURE 7.11: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:FFT, LIB:GP-LVT, WORST CASE 59
FIGURE 7.12: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:FFT, LIB:GP-SVT, NOMINAL 60
FIGURE 7.13: POWER COMPARISON OF POWER REDUCTION AND SWITCHING OVERHEAD, BENCHMARK:FFT, LIB:GP-SVT, WORST CASE 60
FIGURE 7.14: POWER/ENERGY REDUCTION FOR OVERALL DESIGN AFTER ESTIMATION AT PHYSICAL LEVEL (ACTUAL) 61

List of Tables:

TABLE 5.1: EEMBC APPLICATION PROFILING STATISTICS 29
TABLE 5.2: EEMBC APPLICATION PROFILING STATISTICS INCLUDING IIC 34
TABLE 6.1: POWER DOMAINS AND MODES FOR THE POWER GATED FLEXCORE DESIGN 43
TABLE 6.2: POWER AND GROUND NETS FOR THE POWER GATED FLEXCORE DESIGN 43
TABLE 7.1: OVERALL ENERGY & POWER REDUCTION AFTER RTL SYNTHESIS, BM: AUTCOR & FFT 55
TABLE 7.2: ENERGY CONSUMPTION OF ORIGINAL DESIGN AND REDUCTION ESTIMATIONS 62
TABLE 7.3: POWER CONSUMPTION OF ORIGINAL DESIGN AND REDUCTION ESTIMATIONS (ACTUAL) AFTER PHYSICAL IMPLEMENTATION 62
TABLE 7.4: AREA COMAPRISON OF SPECIAL CELLS AND REST OF THE STANDARD CELLS 63

List of Abbreviations

ALU	Arithmetic and Logic Unit
Autcor	Auto-correlation
ASIC	Application Specific Integrated Circuit
BC	Best Case
CMOS	Complementary Metal Oxide Semiconductor
CPF	Common Power Format
DIBL	Drain Induced Barrier Lowering
EEMBC	Embedded Microprocessor Benchmark Consortium
EIC	Effective Idle Cycles
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GIDL	Gate Induced Drain Leakage
GP	General Purpose
GPP	General Purpose Processor
HDL	Hardware Description Language
IIC	Intermediate Idle Cycles
ILP	Instruction Level Parallelism
ISA	Instruction Set Architecture
LS	Load/Store
LVT	Low Voltage Threshold
MIPS	Microprocessor without Interlocked Pipeline Stages
MMMC	Multi Mode Multi Corner
NMOS	Negative channel Metal Oxide Semiconductor
Nom	Nominal
N-ISA	Native Instruction Set Architecture
PC	Counter
PLA	Programmable Logic Array
PMOS	Positive channel Metal Oxide Semiconductor
RF	Register file
RTL	Register Transfer Level
RTN	Register Transfer Notation
SDC	Synopsys Design Constraints
SoC	System On Chip
SRAM	Static Random Access Memory
SVT	Standard Voltage Threshold
VTH	Voltage threshold
WC	Worst Case

Tools and Technology

Tools:

Software Analysis

- GCC MIPS Cross-Compiler v 4.1.1
- FlexSoC Compiler
- FlexSoC Simulator
- FlexSoC HDL Generator

Hardware Implementation

- Cadence Encounter RTL compiler v 9.1
- Cadence SoC Encounter v 8.1
- Cadence Incisive Design Simulator v 8.2
- Common Power Format v 1.1

Benchmarks:

EEMBC, Telecom Suite

1. Auto-correlation (Autcor)
2. Fast Fourier Transform (FFT)

Technology:

STM 65 nm v 5.2.2

- General Purpose Standard threshold
- General Purpose Low threshold
- Operating Conditions:
    - Nominal: 1.0 V, 25 C
    - Worst Case: 0.9 V, 125 C
    - Best Case: 1.1 V, -40 C
- Special Cells
    - Isolation
    - State retention
    - Header power switch

Other Specifications:

Clock Period – 3500 ps (285 MHz)

Typical Supply Voltage – 1.0 V

1. Introduction

Power consumption is predicted to be increasing with the scaling of the transistor size and is heading to be an important concern in modern design [1] and [2]. One factor contributing to power is the addition of more transistors per chip which contributes to the increasing dynamic power and the other is increase in leakage current or stand-by current due to the technology itself. The dynamic power is efficiently reduced by scaling down supply voltage. But order maintain the circuit speed the threshold voltage has to be reduced at the same time. This adversely affects the leakage power. For example at 65 nm technology the leakage power is already comparable to dynamic power.

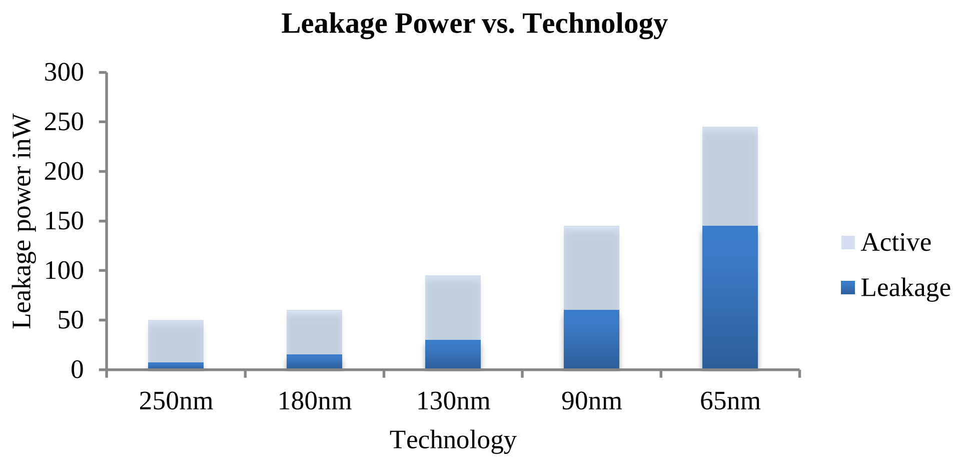


Figure 1.1*: Trend of dynamic and leakage power for general processors.*

Source: Intel:"Power Consumption of Microprocessors".

Another aspect of increasing power dissipation that needs attention is the increasing complexity of embedded applications and the limitation of battery capacity for portable devices. Increased complexity is due to addition of more functionality and thus more units to the processor. This increases both the dynamic and leakage power, while the requirement is to improve the battery life. With deep sub-micron transistor technologies the situation will get worse. Hence there is need for effective power management to reduce both dynamic power and leakage power. Another motivation for power management of embedded processors is that, some major units are idle for most of the operating time and some units when active might not be critical in terms of timing. The first case provides an opportunity to reduce the leakage power and the second case provides an opportunity to reduce the dynamic consumption of non-critical units by voltage scaling.

Several power management techniques have been used to reduce dynamic and leakage power dissipation. Scaling down the voltage is the most effective way of reducing dynamic power due to its square dependency. Figure 1.2 shows a set of power reduction techniques applied to a raw design. Clock gating and voltage islands reduce the active power. Multi threshold (Multi V_{TH}) and power gating are the most commonly used techniques for leakage power reduction. Power gating is highly effective in leakage reduction compared to multi threshold transistor technique. Power gating can reduce leakage power up to 50 times. However, multi-V_{TH} transistor placement is automated, so that there is no timing penalty, whereas power gating can have small timing penalty. Area penalty is also higher for power gating. Although power gating has timing and area penalties, if optimally used can have significant leakage reduction compared to multi-V_{TH}. Another technique to reduce leakage is substrate biasing, which is more complex to implement and has less effect when voltage supply is scaled down for technologies below 65 nm. These techniques are discussed in brief in Section 2.2.

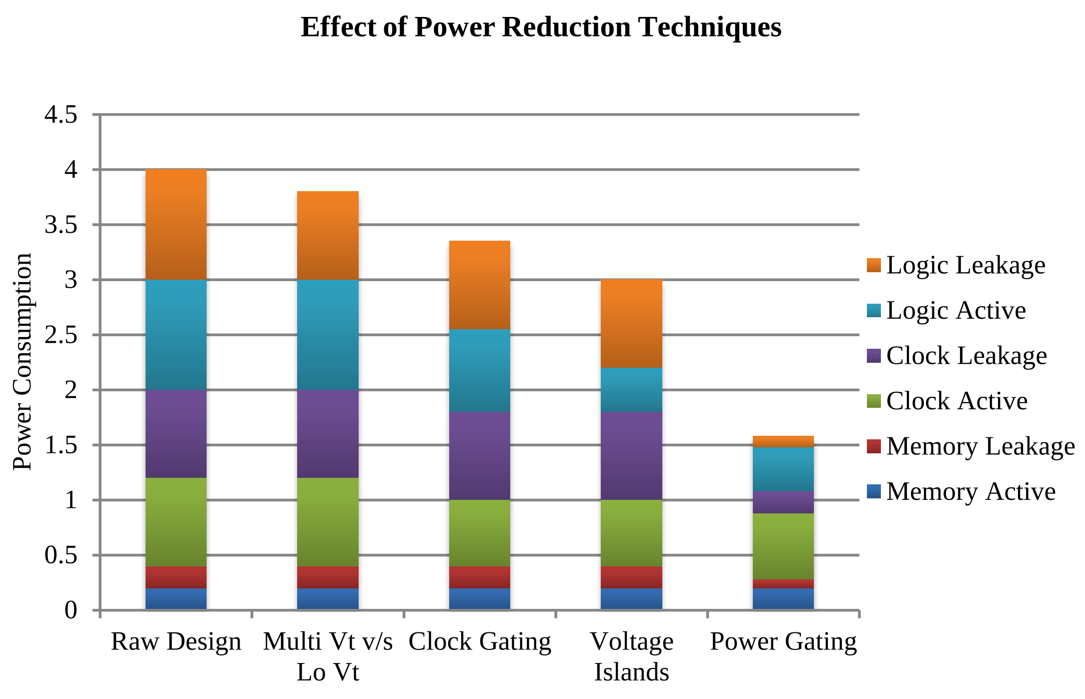


***Figure 1.2**: Effects of different power reduction techniques.*

Source: Chip Design Magazine – "Be Early With Power".

1.1 Motivation

The importance of leakage power reduction and room for leakage reduction in embedded applications mentioned in the previous section are the motivations for this thesis.

The thesis aim is to reduce the leakage power of the FlexCore processor by power gating functional units of the processor. Initial studies show that the multiplier is the most suitable unit on which power gating can be applied since the multiplier is a relatively huge block compared to the rest of the units of the FlexCore and evaluation of several EEMBC benchmarks on the FlexCore shows that it is idle for a large duration. This results in significant leakage power consumption. Power gating is employed to cut down this leakage but this comes with the expense of some area and timing overheads.
There could also be power overhead if the technique is not employed appropriately, which is possible only through an exhaustive software analysis on the applications considered.

This thesis focuses on applying power gating technique to the multiplier unit of the FlexCore. The thesis work is divided into the *software analysis* and the *hardware implementation*. The software analysis phase involves evaluation and modification of the instruction set to provide power gating control and to identify the best instants to turn ON or OFF the unit. The hardware implementation phase focuses on the implementation of the power gated architecture using the common power format at the RTL level and physical level. The final integration phase will apply the information from the software analysis on the new power-gated FlexCore to show the power and energy reduction at the cost of some area overhead.

2. Power Reduction

2.1 Power Dissipation in CMOS Circuit

Power dissipation in a CMOS circuit is contributed by dynamic power, short-circuit power and the static power or leakage power. Dynamic power is a result of switching of the gates when the circuit is operating in an active state. Short-circuit power is a result of current flowing from V_{DD} to ground every time a transistor switches. This occurs for a short duration of the switching time due to finite rise and fall times of the gate signals, which results in both the PMOS and NMOS being ON at the same instant and forming a path from the supply to ground. Static power or leakage is the power consumed by a circuit during stand-by i.e. when the circuit is not in use. The total power consumption of the circuit thus can be written as

$$P_{total} = P_{dynamic} + P_{short-circuit} + P_{leakage} \qquad [1]$$

The dynamic power of the circuit is a function of the switching activity (α), clock frequency (f_{clk}), supply voltage (V_{DD}) and switching load capacitance (C_{load}) as given in Equation 2.

$$P_{dynamic} = \alpha f_{clk} V_{DD}^2 C_{load} \qquad [2]$$

As Equation 2 suggests that dynamic power reduction can be achieved by reducing any of the four factors and reducing the supply has the best efficiency. Techniques such as clock gating, logic restructuring, operand isolation, voltage scaling, dynamic voltage and frequency scaling techniques address one or more of these factors.

2.1.1 Leakage Dissipation

The leakage power is further contributed by four factors, the gate induced drain leakage (GIDL), gate tunnelling leakage, reverse-biased junction leakage and the sub-threshold leakage current [3].

$$P_{leakage} = P_{GIDL} + P_{gate-tunneling} + P_{reverse-junction} + P_{sub-threshold} \qquad [3]$$

Reverse-biased junction leakage current is the same as the reverse saturation current in a diode. The reverse biased diode here is formed between the source or drain and the substrate. The minority carriers near the depletion region and generation of hole-election pairs in the depletion regions form this reverse-biased leakage. Junction reverse-bias leakage components from both the source-drain diodes and the well diodes are generally negligible with respect to the other leakage components.

The gate induced drain leakage (GIDL) is caused by high drain to gate potential and the effect is further increased by high drain to substrate potential. A band-to-band tunnelling occurs in the small overlap region of the gate and the drain. For an NMOS transistor this condition occurs when the transistor is OFF (low gate-voltage) and the drain is at high potential. For a PMOS transistor it occurs when the transistor is OFF (high gate-voltage) and the drain is at a low potential.

The gate leakage flows from the gate through the “leaky” oxide insulation to the substrate. The magnitude of the gate tunnelling current increases exponentially with the decrease in gate oxide thickness (T_{ox}) and increase in the gate supply voltage. Even though the supply voltage is scaled with every technology and that helps reduce the gate tunnelling current, the oxide thickness also has to be scaled for the gate to have effective control over the channel. This again increases the gate leakage current. For an oxide thickness in the range of 2 to 0.5 nm, nearly every 0.3 nm reduction in the thickness for a constant gate voltage results in 10 times increase in the gate leakage current [3]. The gate leakage depends on the gate voltage applied to a transistor. High-k is an effective solution at the technology level.

The sub-threshold leakage is the drain-source current of a transistor operating in the weak inversion region. Unlike the strong inversion region in which the drift current dominates, the sub-threshold conduction is due to the diffusion current of the minority carriers in the channel for a MOS device. The sub threshold current increases exponentially with the linear decrease in the threshold voltage (V_{TH}). As described in [3], the sub-threshold leakage can be written as

$$I_{Sub-threshold} = I_S * 10^{\frac{V_{GS}-V_{TH}}{S}} * (1 - 10^{-\frac{nV_{DS}}{S}}) \qquad [4]$$

Where n is the slope factor between 1-1.5.

I_S is a technology constant current given as

$$I_S = 2n\mu C_{ox}\frac{W}{L} * (\frac{kT}{q})^2 \qquad [5]$$

And S is the sub-threshold swing in the range of 60 mV to 100 mV given by

$$S = n\left(\frac{kT}{q}\right) ln(10) \qquad [6]$$

Further drain induced barrier lowering (DIBL) also causes V_{TH} to reduce in short channel devices. This contributes to huge increase in the sub-threshold current. DIBL is the process of reducing the depletion region near the drain at the influence of the drain voltage. Thereby the threshold voltage near the drain end of the channel reduces. Sub threshold leakage also increases with temperature as suggested by Equation 5.

Overall, the sub-threshold leakage and gate-tunnelling leakage are the main components that contribute to the leakage power in today's transistor technologies. The sub-threshold current is the major contributor to the overall leakage in the 65 nm technology considered in this work. Gate tunnelling leakage will be higher with 45 nm and smaller technologies.

The off state leakage current is the sum of all the above except gate-tunnelling leakage. The gate-tunnelling leakage requires the gate - source - bulk potential to be high.

$$I_{off-Leakage} = I_{GIDL} + I_{sub-threshold} + I_{reverse-biased} \qquad [7]$$

2.2 Power Reduction Techniques

Section 1 and 2.1 show how and where power goes in a chip and some possible techniques to reduce them. This section will discuss these techniques in brief. Among the three components of power consumption as explained in the previous section, dynamic power has been the largest contributor. But the leakage power has been exponentially increasing which smaller transistor technologies (effect of reduced V_{TH}). The quadratic dependence of dynamic power on voltage implies that reduction of voltage will have the highest impact. This has been the largest source of power reduction. The Industry has steadily moved down to lower supply voltage [4]. But reduction in voltage comes as the cost of reduced performance and must also be accompanied with variation of other technology process parameters. Since dynamic power is directly proportional to frequency, a reduction in frequency is suitable for low performance requirements. But the average power consumed per cycle remains the same. In order to reduce the dynamic power the switched capacitance must be addressed. The dynamic power component is reduced either by reduction of the switching activity or by reducing the capacitance or combination of both, like moving a high switching to a node with low capacitance. Leakage power is mainly dependent on the threshold voltage and the drain to source voltage. Higher threshold voltage would decrease the speed of the circuit. One way to address this is to make use of the fact that nearly 80% of a circuit is non-critical with respect to timing [4]. The other technique to eliminate leakage dissipation is to simply disconnect the unit not being used from the supply. This technique is called power gating, which is the technique used in this work to reduce leakage. In the following section few techniques commonly employed for power reducing are discussed.

2.2.1 Dynamic Power Reduction Techniques

- Transistor Sizing: The requirement on performance often leads to up sizing transistors unnecessarily. This is especially true when IC's are custom designed. This over design results in wastage of power. This method of power optimization is concerned with identifying such sources of power wastage and downsizes them. For example transistors on non-critical paths may be up sized for better driving capability but since the overall performance is dependent on the critical path, the up-sized transistors will result in wastage. For synthesized blocks the synthesis tools can automatically identify such sources and downsize them. But for manually designed block it may not be effective and may not be always possible. Tools thus have a great impact. Logic restructuring involves reducing the number of stages wherever possible, so that the total switching activity is reduced. Such techniques are implemented by the modern tools automatically [4].

- Voltage Scaling: Voltage has been the most important parameter for reducing power, although there is some loss of performance. Voltage scaling must be accompanied by reducing the threshold voltage (V_{TH}) to maintain the performance since the delay is approximately inversely proportional to V_{DD}-V_{TH}. If speed is not to suffer excessively V_{DD} must be at least four times V_{TH}. But the problem with such reduced V_{TH} is increase in leakage current. This is more significant in nano-meter technologies. This is a major concern when designing caches, sense-amplifiers, static RAM's and PLA's. Low supply voltage also means that the effect of noise is more and thus reliability is also less.

Voltage Island or multi-supply voltage is a better implementation of the voltage scaling principle where different V_{DD} is given to different blocks depending on the performance requirements. The disadvantage of voltage scaling for the entire design is that the maximum voltage scaling is limited by the performance requirement of the most critical unit. Other units (less critical) might be able to perform with a lower V_{DD}. By voltage islands method, units are separated into islands which operate on different voltage. This technique is more used in SOC designs where there are several functional blocks of varying throughput requirements. Each core has few voltage levels with which it can operate. No islands are needed for blocks operating only at the chip voltage. In voltage island technique level shifters must be added for communication between blocks of different V_{DD}.

Variable V_{DD} or dynamic voltage scaling is another variation of voltage scaling where V_{DD} is dynamically scaled depending on the performance requirements. This is usually employed along with frequency scaling.

- Clock gating and operand isolation are techniques which address the switching activity factor in Equation 2. In clock gating technique clock signal to flip-flops or registers is gated by an enable condition. When these storage elements are not used, the clock is not passed through and unnecessary dynamic activity is reduced. Generally the enable condition is shared with the enable condition for the flip-flops or registers. Similarly operand isolation disables data-path blocks which are not in use by inactivating their inputs through an enable signal. Clock gating and operand isolation are performed by the synthesis tool by enabling certain attributes. The scope for insertion of these is evaluated and inserted during the synthesis step. This step has become an inevitable step in today's design [5]. These techniques help in reducing dynamic power to certain extent. For the FlexCore design the savings from enabling these techniques were small. However as the main focus of this work was on the power gating implementation, the above techniques were not enabled.

2.2.2 Leakage Power Reduction Techniques

- Multiple Threshold Transistors: Multiple threshold transistor design is used to reduce the leakage current which is predominant in the nano-meter technologies. In this technique low V_{TH} transistors are used on the critical path so that the performance is not affected and high V_{TH} transistors are used on non-critical paths so that the leakage power is reduced. Non-critical path means, the path where there is a positive slack. Typically most designs have only about 20% of the total transistors on the critical path, and therefore leakage power can be reduced. Automatic placement by tools ensure that the timing overhead is almost nil.

- Power gating is the most effective technique to reduce leakage power. In this technique a unit which is not in use is disconnected from the supply or the ground so that there is not path for leakage current. It is implemented with the help of low leakage PMOS or NMOS transistors. The gate of the PMOS or

NMOS transistor decides whether the supply or ground respectively is connected to the unit. This method needs additional units to implement correct functionality and thus leads to some area penalty. There could also be a small timing penalty due to switching transition time. But 5% penalty is acceptable. This technique is best suited for units with large idle times. In this case the overhead is acceptable for the leakage gain that can be achieved. But to achieve optimal leakage reduction, the application profile has to be understood for effective control of the unit. Section 3 will explain about power gating in more detail.

- Other techniques at circuit level include usage of long channel transistors, thereby reducing the effect of drain induced barrier lowering (DIBL) and again this technique is very useful for small channel length transistors. But the increased channel length means more delay but this is compensated by making it wide, which gives high area overhead. This method is especially useful for SRAM's where delay is not important and its power consumption is mainly due to static power. Parking states technique forces gates or block to low leakage logic state when they are not in use. This method requires finding the input vectors that puts the unit to least leakage logic. The technique might be suitable for units that have higher non-idle time.

- Substrate biasing is a similar technique and an improvement over the MTCMOS, where the back bias voltage of the transistor is altered so that V_{TH} changes. The substrate bias voltage can be varied dynamically depending on the requirement. The threshold voltage varies on the substrate bias as

$$V_{TH} = V_{TH0} + \gamma(\sqrt{V_{SB} + 2\varphi_F} - \sqrt{2\varphi_F}) \qquad [8]$$

Where φ_F is the Fermi-potential and V_{TH0} is threshold at zero bias. As seen from the Equation 8 the threshold is square root function of the substrate source bias voltage. As the technology goes to smaller transistor sizes the voltage is scaled too. Hence the voltage V_{SB} can only be changed by a limited extent, which leads to only a small change in V_{TH}. Hence it might not be effective with future technologies. There is also extra routing overhead.

Altera's 40nm Stratix 4 FPGA's use programmable substrate biasing technique to increase V_{TH} on blocks which are on the non-critical path. In normal FPGA's all paths are optimized for high speed, whereas in Stratix 4 only blocks on critical path are optimized for high speed, and others are either give a back bias or if they are not in use they are isolated from supply by using power gating techniques [6].

3. FlexCore Processor

3.1 Background

FlexCore embedded processor is the platform on which power gating is applied in order to reduce leakage power [7]. FlexCore was developed as the first exemplar within the FlexSoC project by the VLSI Research Group at the Chalmers University of Technology. The FlexSoC approach moves away from the hard-coded instruction set architecture (ISA) by introducing a reconfigurable interconnect which is governed by a *wide control word*. In the FlexSoC framework, conventional methods to provide application performance are replaced with fine-grained control. Though FlexCore is based on the traditional five-stage pipeline architecture, it is a unique processor designed to be flexible and eventually extensible depending on the application specifics. Most importantly, the core stands out with its ability to have a reconfigurable datapath depending on the application, within the same architecture and compilation framework [7]. Subsequently, such flexible datapath configuration for each specific application without affecting the baseline processor architecture results in improved performance

3.2 Baseline Architecture of the FlexCore

Original design of the processor is based on a simple MIPS R2000 processor's architecture in order to maintain full general-purpose processor (GPP) functionality. Therefore, FlexCore baseline architecture includes functional units to support the full GPP programmability such as Program Counter (PC), Load/Store (LS), Register File (RF), and Arithmetic and Logic Unit (ALU) (Figure 3.1). Two buffers are added which allows data to be directly restored or taken by any functional unit for more efficiency than writing data back to RF or memory. What distinguishes FlexCore processor architecture from the GPP architecture is that instead of using a dedicated interconnect, all functional units communicate through a flexible, fully connected interconnect. Although reconfigurable interconnect incurs power and area overhead, it can be utilized through trimming communication links under application profiling [7].

3.2.1 Multiplier Extension

FlexCore also distinguishes itself with a feature which allows its architecture to be extended by adding more ports to switch-boxes of interconnect and extending the *wide control word* to include control signals for the new units [7].

The core has been extended by adding a multiplier to the baseline architecture in order to increase efficiency of executing multiplication-based embedded applications, such as the Fast Fourier Transform (FFT). This efficiency comes at the expense of two factors. First, the size: the multiplier unit is three times larger than the adder unit. Second, not all applications use multiplier as often as the FFT application. Therefore multiplier being a relatively large unit and used for a small fraction of the time results in considerable amount of leakage power dissipation. Hence, multiplier unit is the focus in this work which it is intended to be power-gated in order to reduce leakage power dissipation of the FlexCore.

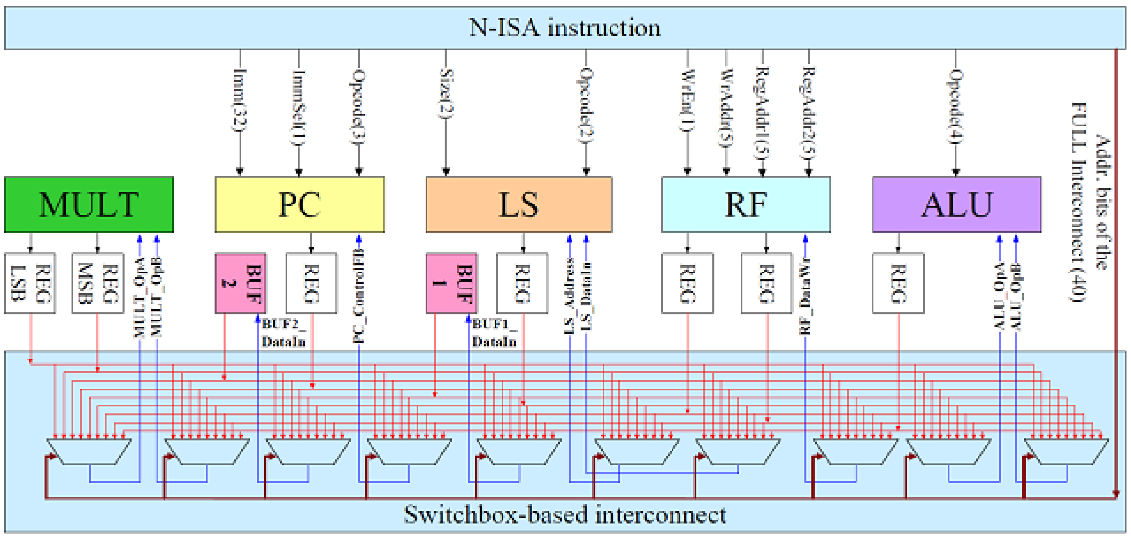


***Figure 3.1**: Multiplier-extended FlexCore processor*
(The enable signals of the datapath units are not shown)

3.2.2 Flexible Interconnect

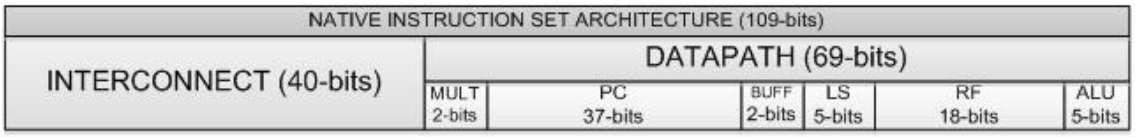
Switch-box based interconnect with 90 possible links is a key difference of the FlexCore from other application-specific embedded processors. Together with another key component, exposed datapath, this full interconnect provides freedom for data to be routed among any function units of the processor [8].

3.2.3 Native Instruction Set Architecture (N-ISA)

While GPP instructions are hard-coded, datapath and interconnect operations are exposed to the compiler in terms of a *wide control word*, named as Native-ISA (N-ISA). Having this *wide control word* of an exposed datapath, the FlexCore framework allows modelling various architectures and subsequently the N-ISA instruction permits the compiler to have more opportunities to efficiently schedule the instructions [9].

Structure of the N-ISA instruction highly depends on the number of datapath units and interconnect configuration. The N-ISA is mainly divided into two sections: datapath control bits and interconnect addresses bits (Figure 3.2). Starting from the least significant bit at the datapath control, the section consists of 5 bits for ALU, 18 bits for register file, 5 bits for load/store unit, 1 bit each for two data buffers, 37 bit for program counter, and 2 bits for multiplier. Overall, 69-bits are required for the datapath control. N-ISA interconnect has total 40 bits in where 4 bits are required for every switch-box dedicated for each functional unit.

Using long N-ISA comes to the cost of instruction bandwidth and a large memory footprint, but it can be resolved through instruction compression [8].



NATIVE INSTRUCTION SET ARCHITECTURE (109-bits)						
INTERCONNECT (40-bits)	DATAPATH (69-bits)					
	MULT 2-bits	PC 37-bits	BUFF 2-bits	LS 5-bits	RF 18-bits	ALU 5-bits

***Figure 3.2**: FlexCore N-ISA instruction*

4. Power Gating for the FlexCore Processor

Power gating or power shut-off is a technique to reduce the leakage power of functional units or modules in a design. The unit considered for power gating is shut-off or deactivated when not in use and thereby reducing unnecessary leakage current. Shut-off involves disconnecting the unit or a gate from the supply or the ground using header or footer switches respectively. The gating is implemented with the help of either a header or a footer cell which are low leakage PMOS or NMOS transistors respectively. Figure 4.1 shows the header and footer switch placement. In case of header switch implementation the supply net to the CMOS circuit that connects to the switch is referred to as virtual V_{DD} or in case of footer cell implementation the internal ground net is referred to as virtual ground.

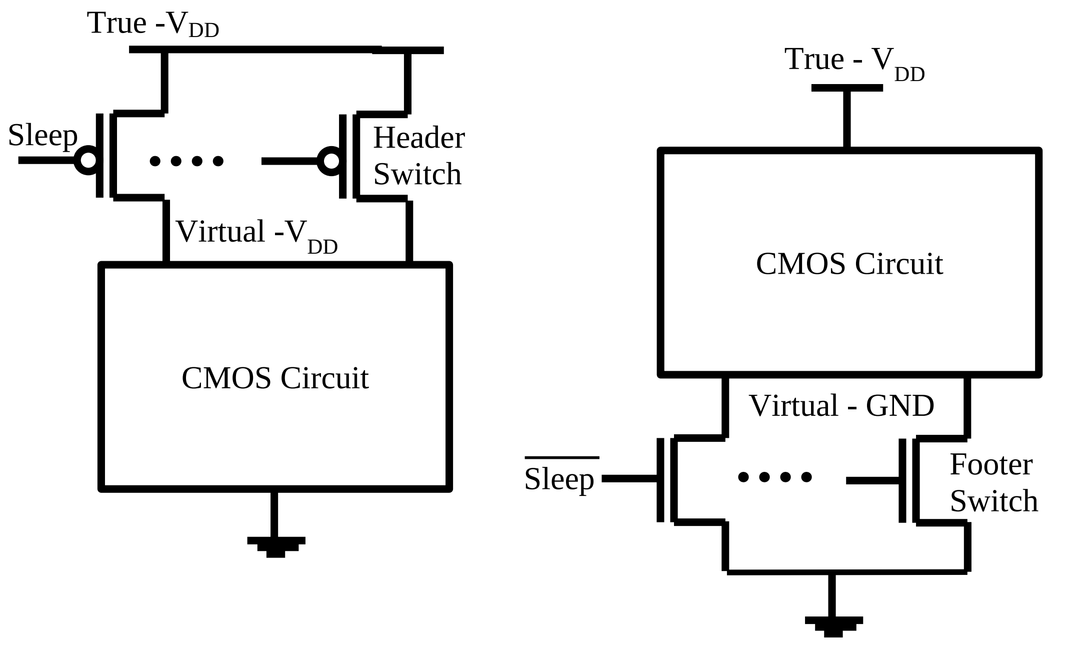


***Figure 4.1**: Header and footer switch placement for power gating a unit.*

The size of the switch depends on the maximum current consumption of the circuit and the capacitance on the switched supply. In a practical implementation the switch size is fixed and multiple switches are placed in parallel depending on the current requirement. The number of power switches that are required for a design has to be determined considering factors such switch area, leakage reduction and voltage drop. Small size leads to larger voltage drop due to smaller resistance and this will impact the performance [9].

The implementation of these switches can be performed either in a coarse-grained or fine-grained fashion [5]. In a coarse-grained implementation switches controls an entire block, where as in a fine-grained implementation several switches control smaller sections of the unit like cells or gates. In coarse grained implementation although the area overhead is small, switching capacitance is high and there will high rush currents. In the fine grained implementation since several switches address smaller units, the switching capacitance is low but the area overhead is much higher. This could also result in some leakage from the large number of switches. Coarse grained is suitable if all smaller units of the circuit being considered operate simultaneously. Fine grained is suitable if units have different operating profile. Also the coarse grained technique is easier to place and route.

Further the switches can be placed in a ring or column fashion [5]. In a ring fashion as the name suggests switches are placed around the unit being power gated. In a column fashion they switches are placed in columns through the length of the unit. Ring placement is more efficient in terms of routing, but area overhead is slightly higher as compared to the column placement technique. Figure 4.2 shows the placement of these two types.

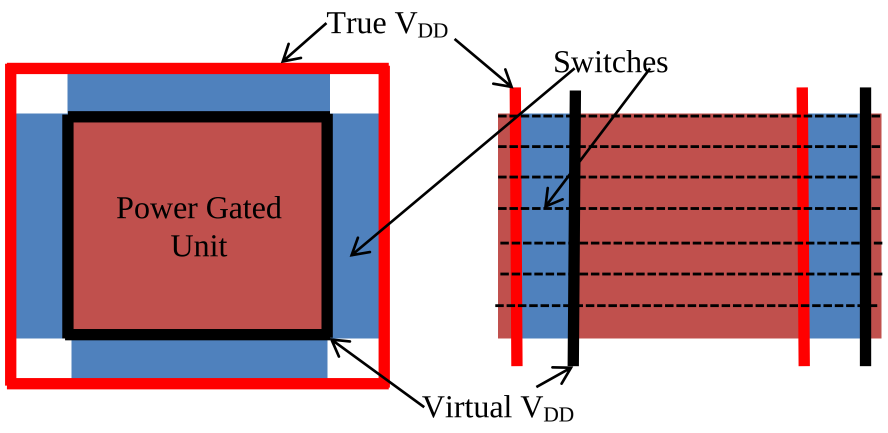


***Figure 4.2**: Placement of power switches*
Ring (left), Column (right)

When power gating, another important issue to be considered is the floating states of the outputs of the power-gated unit. Also flip flops if present in the design will lose their state. Hence special cells to eliminate these problems are needed. Isolation cells are used to prevent floating outputs and state retention cells to save flip-flop states [5]. The isolation and state retention cells are showed in Figure 4.3 and Figure 4.4. Isolation cells are simple two input AND/NAND or OR/NOR gates with an isolation enable signal as one input and data as the other input. AND or NAND work with an active low signal for *isolation* and OR or NOR work with the active high signal for *isolation*. AND or NOR gate drives the output to 0 when isolated and the OR or NAND gate drives the output to 1 when isolated. Isolation can also be achieved with D latches in which case the output will be forced to the last output of the power-gated unit prior to shut down. State retentions cells operate on the virtual supply when the power down unit is in normal operation and switches to the global supply when the unit is shut down.

The cells can share the same enable signal. But there must be definite power up and power down sequence for these cells. A power up of the switches has to happen before the isolation and state retentions are disabled. If the power up occurs later then purpose of preventing floating states and preserving states of flip flops is damaged. The sequence followed in power down is isolation, state retention and power down of the switch and the reverse sequence during power up. This either automatically performed by the tool or manually performed by insertions of buffer cells. Figure 4.5 depicts the required power up/down sequence. The isolation, state retention cells are specified during the synthesis and place and route phase respectively. The switches are either automatically or manually placed during physical implementation. The Cadence SoC

Encounter tool used in this design performs the placement of the switches either in ring or column fashion through special commands.

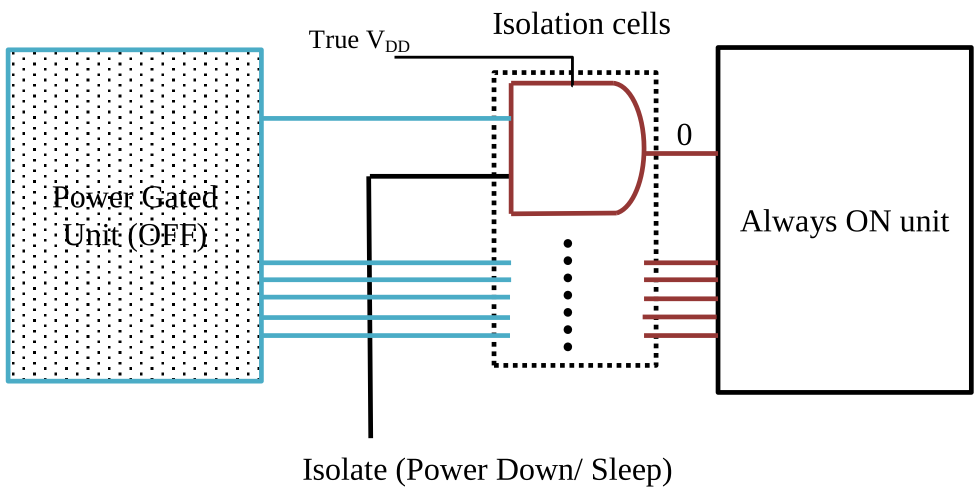


***Figure 4.3**: Isolation cells inserted at the output of a power-gated domain*

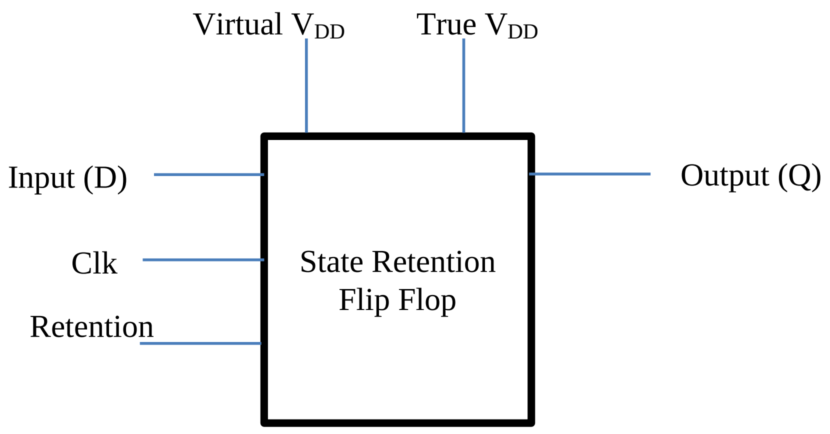


***Figure 4.4**: State retention flip-flop*

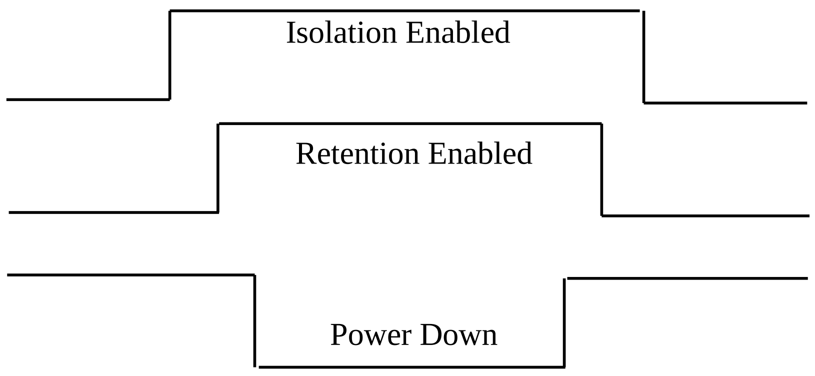


***Figure 4.5**: Power down sequence*

4.1 Power gating the Multiplier

Figure 4.6 shows the block diagram of the Flex-Core with the multiplier power-gated. A header switch is used to power gate the multiplier in this work[1]. A coarse grained power gating is implemented. In a coarse grained implementation the switch(s) perform the power shut-off for the entire unit by switching the virtual supply between the true V_{DD} (TV_{DD}) and the off-state voltage. The off-state voltage is not at zero potential and is usually a value close to the threshold of the header switch. The switches are placed in a ring fashion since placement and routing are easier during the physical implementation[2].

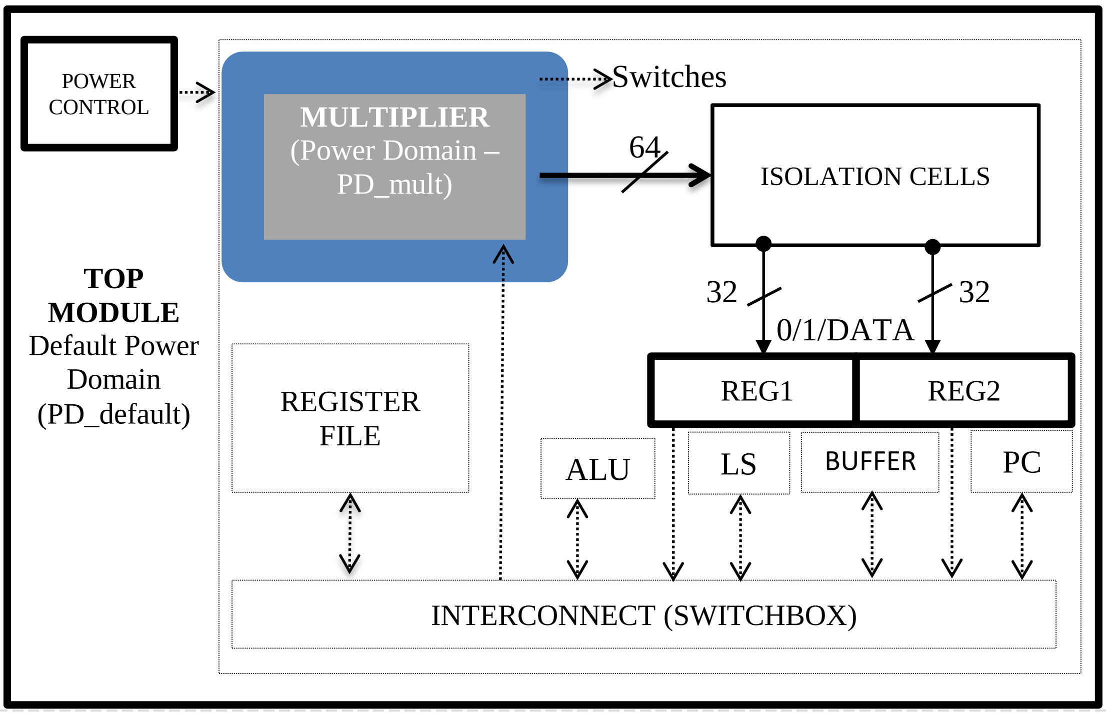


***Figure 4.6**: Multiplier power gated view of the FlexCore*

The multiplier unit which is considered for power gating is defined in a different power domain and the domain is termed as *PD_mult*. A power domain is a region of the design having specific power architecture different from rest of the design region. All units in a specific domain will follow the rules of power architecture specified for that domain. The rest of the FlexCore modules are defined in the in the default power domain *PD_default*. All modules and units unless specified to be in a specific domain will be placed in the default power domain.

The technology libraries used in this work provides two types of cells for the switch implementation, the switch control cell and the PMOS header switch. The control cell receives the switch enable signal along with signals to control the transition current consumption and signal to enable detection of valid states. The switch ON and OFF of the multiplier is performed by PMOS header switches. The gate signal of these switches called the "*switch enable*" defines the turn-ON and turn-OFF of the multiplier. The switch control unit drives the *switch enable* signal of the PMOS switches. The switch

[1] The STM65 v5.2.2 library used here is provided with header switch only.
[2] The aim of this work is focused more towards estimating the power reduction rather than electrical impact of the switche.

control cell provides more flexibility in terms of switch transition time by controlling the current consumption to transit from the off-state supply to the true global supply, i.e. It provides an option to have a more smoother transition at the expense of wake-up time. The switch control cell also generates signals to indicate the detection or switching to a certain valid state. A signal to indicate transition of the internal V_{DD} or the virtual V_{DD} to the true global V_{DD} is generated always. Other signals are generated only by enabling the '*detection ON*' for the switch control. Figure 4.7 shows the switch arrangement for the multiplier.

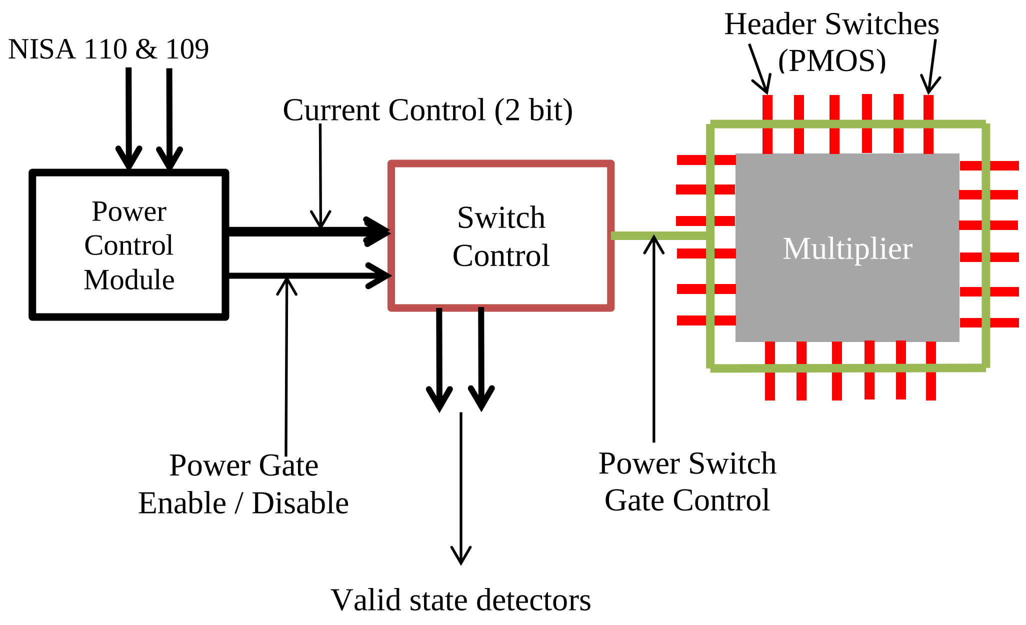


Figure 4.7: *Switch control arrangement*

The switch control unit consists of two *internal switches*, two *current controlled sources* and two *detectors*, one each for the virtual VDD and the PMOS gate control signal. There is also *control logic* to generate different control signals based on the input to the unit. The *current control* input signals to the unit controls the current output of the *current controlled source*, which in turn decides the switching transition duration. This unit has a dimension of 99.2 µm × 24 µm (2380.8 µm^2).

The off-state multiplier will have outputs in floating state, which can affect the functionality of other units that depends on these outputs. In order prevent the propagation of these floating signals to other units, the multiplier outputs must be isolated from units that depend on it. The only unit connected to the multiplier output is the interconnect unit, through the multiplier registers. The actual multiplier unit in the FlexCore hierarchy consists of the multiplier logic and the registers shown outside the *PD_mult* domain in the Figure 4.6. By restricting the shut-down domain to the logic and keeping the registers outside will eliminate the need for state retention cells. There will be no gain in power but only a small increase in area if state retention cells were to be used . Another option is to still define the registers inside the power gated domain and include the D-latch isolation cells at the output of the registers. However in this work only isolation cells with AND or NAND gates will be inserted in final design.

The *switch enable* signal can be either controlled directly by the software via the NISA binary instruction or by a dedicated "*power control module*". The power control module can be either an on chip or off chip controller. In this work an on-chip controller is used. The power control module sends the power control information to the switches either by in-built power-control logic or passing the information from the NISA instruction. The power control module can operate in three modes which are explained in Section 6.3.
Power control for the multiplier is best achieved by software via the extended NISA instruction. Hardware control is inefficient for this processor. Hardware control was used only during the initial stages to verify functionality and to estimate power reduction. Under software control the only important function to be performed by hardware is to delay the power down by some cycles till the output is stable, which is difficult to be implemented on the software.

5. Software Analysis

5.1 FlexSoC Tool-chain

The FlexCore processor, targeting embedded systems, has been developed by VLSI Research group at Chalmers in the context of FlexSoC project. The FlexCore processor combines the advantages of power efficiency and high performance in Application-Specific Integrated Circuits (ASICs) and flexibility and programmability of General Purpose Processors (GPPs). A reconfigurable interconnect allows extensions to the datapath as well as flexible routing of data between datapath units.

In order to support features of the FlexCore processor, a FlexSoC tool-chain (FlexTools) was also developed as follows:

1. Software analysis

    - FlexSoC Compiler – compiles MIPS-assembly code into Register Transfer Notation (RTN) code of the FlexCore processor.

    - FlexSoC Simulator – generates instruction and data binary codes from RTN code which are used to verify and estimate power-performance of FlexCore processor.

2. Physical implementation

    - FlexSoC HDL Generator – generates RTL code for an instance of the FlexCore processor with the specific datapath and *interconnect* configurations.

    - EDA tools for synthesis, place and route, and verification.

In addition, a Makefile was created to chain up all tools to evaluate the properties of the FlexCore processor from C-code applications to physical implementation. In this section, the focus is software analysis whose methodology flow is presented in Figure 5.1.

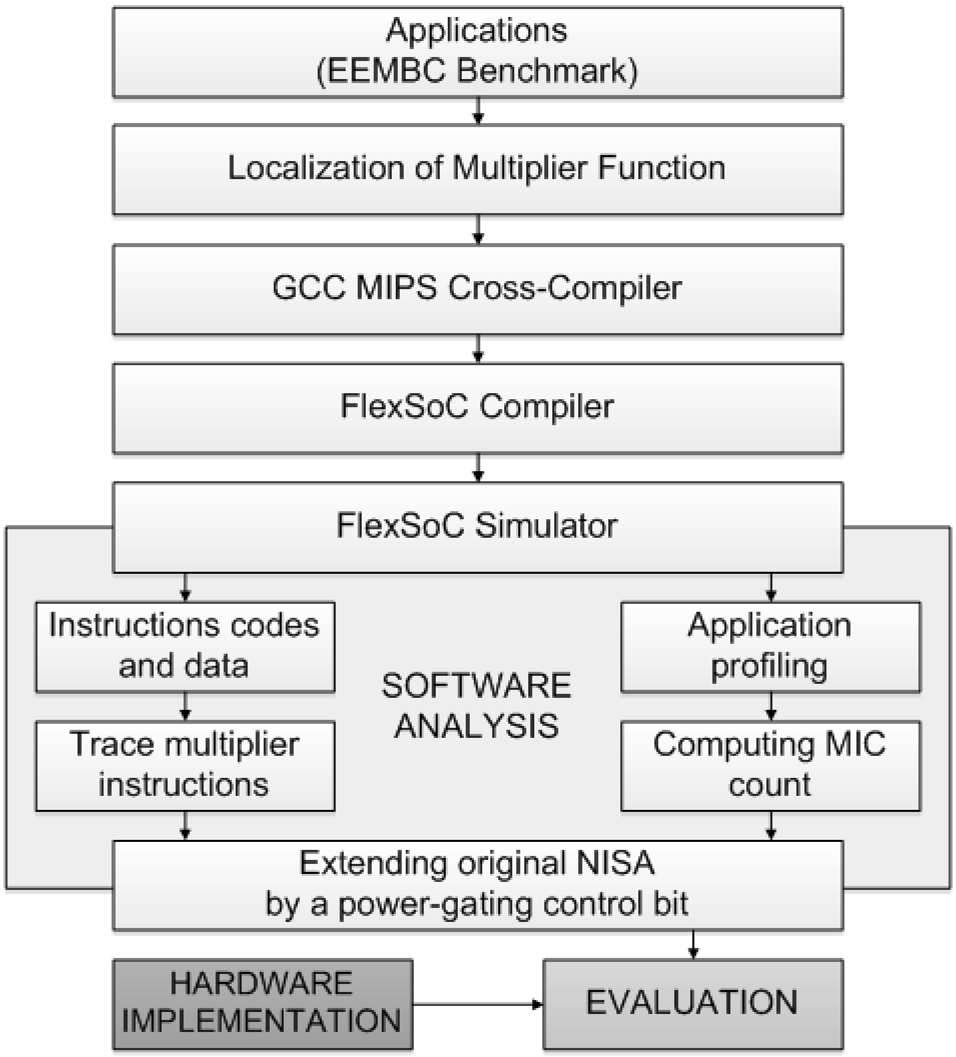


Figure 5.1*: Methodology flow of the software analysis*

5.2 EEMBC Benchmark

In order to examine performance of the FlexCore processor for a diverse range of the applications, there are 10 benchmarks available from the Embedded microprocessor benchmark consortium (EEMBC). They are from three suites, *aifirf*, *canrdr*, *bitmnp* of the Automotive suite; *rgbcmy*, *rgbhpg*, *rgbyiq* of the Consumer suite, and *Autcor*, *conven*, *viterb*, *fft*, of the Telecom suite. All these benchmarks are integer and no-division applications because at the moment the FlexCore processor does not support floating-point computation and hard-divider in its datapath [8].

Out of 10 available EEMBC benchmarks, we selected two benchmarks that are Autcor (Auto-correlation) and FFT (Fast Fourier Transform) from the Telecom suite, for evaluating the impacts of the power gating technique in the scope of our thesis. Autcor and FFT are chosen because they are different in size and use the multiplication operation in different ways. In detail, FFT benchmark is one of the largest among EEMBC applications, with 162,967 cycles in total. Autcor uses 19,553 cycles, a number fairly similar to the other provided EEMBC application.

The multiplication property for two selected benchmarks is traced and reported in Table 5.1 It is clear that there is not a significant diversity in usage of the multiplier in both applications. However, the cycle count of the FFT benchmark is 8.33x higher than Autcor, making the FFT's multiplier utilization one of the largest among the other available EEMBC benchmarks as determined in the initial pre-study.

Table 5.1: *EEMBC application profiling statistics*

Benchmark	Total cycles count	MULT-only	
		Cycles	% of total
AUTCOR	19553	400	2.05
FFT	162967	10240	6.28

In this multiplier-only usage (MULT-only) computation, the intermediate cycles between all two consecutive multiplier activations were not included, only the cycles during which multiplier was active.

Intermediate idle cycles (IIC) of the multiplier are significant factor when applying power gating technique. Every benchmark has different values of IIC which changes during its execution period. If the multiplier is power gated without considering IIC, there would be a large number of switch transitions, which would lead to power overhead. On the other hand, considering IIC in a way that makes multiplier to be ON or OFF for a significant period of time would result in no power savings. In order to achieve best power savings, the most optimal IIC (or *effective idle cycles*, Section 7.2) must be determined. Multiplier's IIC computation and results are presented further down the front-end flow in Section 5.5.1.

5.3 Localization of the Multiplier Function

After selecting applications to apply power gating technique in the FlexCore processor, it is necessary to take into account the behaviour of the multiplier in the C-code of both, Autcor and FFT benchmarks. Since multiplier unit of the FlexCore processor is chosen to be power-gated, the active-cycle count and the idle-cycle count between two consecutive multiplications are important features which need to be extracted through application profiling by using FlexTools. In order to do so, the multiplication instructions are localized in terms of a separate function with two or more input parameters and this function is called within the main program. As a result, FlexTools now are able to provide essential information which allows to us to estimate how often the multiplier to be used as well as how long the multiplier to be activated or de-activated in terms of a cycle count. An example of localizing multiplication instruction is shown in Figure 5.2.

Multiplication in original C code:
```
Accumulator += ((e_s32) InputData[i] * (e_s32) InputData[i+lag]) >> Scale;
```

Localize multiplication instruction in the separate function:
```
MultFunc(InputData[i], InputData[i+lag]);
```

Call multiplication instruction in the main program:
```
Accumulator += (*mult_result >> Scale);
```

***Figure 5.2**: Multiplication localization example in the Autcor application C-code. MultFunc is a name of the multiplication function, mult_result is an output of the MultFunc*

Notice that localization of the multiplication instructions must not introduce errors in the functionality of benchmarks. This is performed by verifying the re-organized C-code to guarantee that it is exactly executed as the original benchmark. Afterward, the verified C-codes are ready to be compiled. The goal of re-organizing the original C-codes is to help FlexTools to identify individual multiplication functions and provide application profiling information specifically related to the multiplier unit.

5.4 Tools Chain

5.4.1 MIPS Cross-compiler

GCC MIPS Cross-compiler is an open source tool that FlexTools relies upon. Since the datapath of the FlexCore processor share almost the same functional units with the conventional GPP processor (MIPS-lite datapath), GCC MIPS Cross-compiler is used to compile the C-code into the MIPS assemble code which is provided to FlexSoC compiler. Therefore, in the following step of the front-end flow, a re-organized C-code together with the other required library’s files are translated into MIPS assembly by a GCC MIPS Cross-Compiler. In this study, the GCC-4.1.1 version was used to assemble the C-code as a stable version. The other newer version of GCC might not be compatible with the FlexSoC compiler, thus, they need to be pre-tested. The execution of the MIPS Cross-compiler is controlled by rules in the Makefile (Appendix-A).

5.4.2 FlexSoC Compiler (FlexComp)

Next, a MIPS-assembled code is compiled by the FlexSoC compiler, named FlexComp, which is a backbone of the FlexTools. As soon as MIPS assembly files are available, FlexSoC compiler can produce RTN code for the FlexCore processor with a specific datapath and interconnect configurations.

During the FlexComp compiling, MIPS assembly instructions are scheduled and expressed in a single, long RTN instruction which makes it possible to achieve high instruction level parallelism (ILP). It is clear that using a reconfigurable interconnect, the datapath operation of the FlexCore processor is exposed to the compiler which is exploited to gain ILP against to the GPP processor.

5.4.3 FlexSoC Simulator (FlexSim)

The main target of this step is to generate all data which are required for application profiling as well as hardware verification. FlexSim, a cycle-accurate simulator, takes RTN code as inputs, simulate and generate data and instruction code in terms of Native Instruction Set Architecture (N-ISA) for back-end phase. Furthermore, application profiling can be performed through using outputs provided by FlexSim. Due to the fact that FlexSim simulator provided an exposed N-ISA code, it is possible to trace of the operation of the individual functional units in a cycle-by-cycle manner. Several necessary options of FlexSim used for applications profiling are listed as follows:

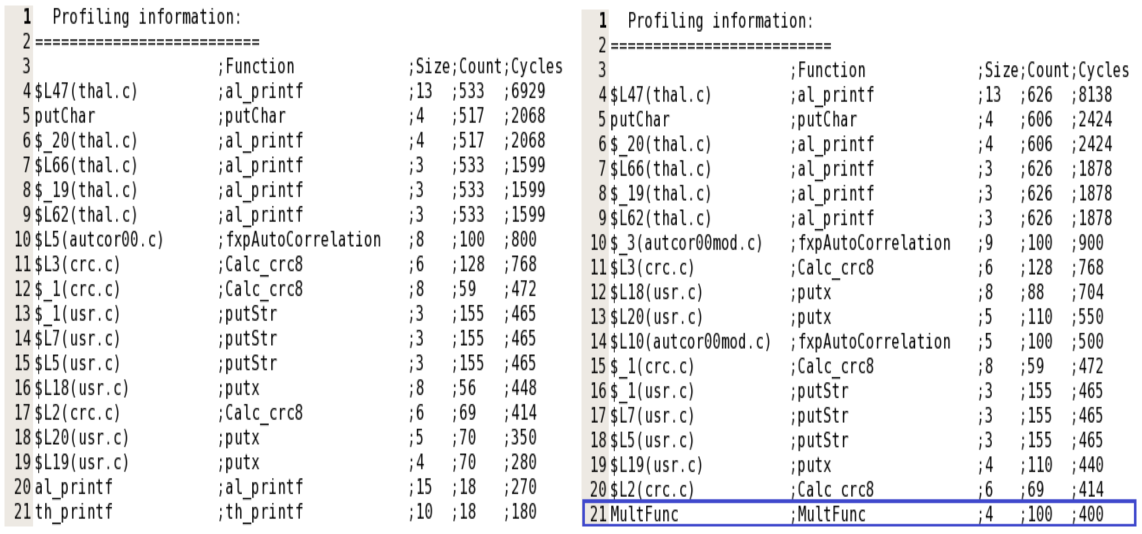
-PROF	Application profiling. Cycle-count and frequency for individual functions used within applications.
-TRACENISAD	Tracing and debugging. Mixed binary/RTN instruction format for debug.
-TNISA	Showing program as timed N-ISA instructions in binary format.
-TRINARYTNISA	Showing program as timed N-ISA instructions in hexadecimal format.
-SHOWCODE	Showing program after static scheduling.

All output formats can be dumped into files for post-processing. As the FlexSoC simulator generates required information for software analysis and, subsequently, to apply power gating technique, the flow continues to the next stage of understanding content of the output files.

5.5 Evaluation of Multiplier Behaviour

5.5.1 Computing IIC Value

The FlexSim is used with an option –PROFF to generate a file, named 'Profile', which provides information related to cycle-count of individual function. Snapshots of profile files for the original and re-organized C-code of the Autcor benchmark are shown in Figure 5.3.



```
1  Profiling information:
2 =========================
3                       ;Function           ;Size;Count;Cycles
4 $L47(thal.c)          ;al_printf          ;13  ;533  ;6929
5 putChar               ;putChar            ;4   ;517  ;2068
6 $_20(thal.c)          ;al_printf          ;4   ;517  ;2068
7 $L66(thal.c)          ;al_printf          ;3   ;533  ;1599
8 $_19(thal.c)          ;al_printf          ;3   ;533  ;1599
9 $L62(thal.c)          ;al_printf          ;3   ;533  ;1599
10 $L5(autcor00.c)      ;fxpAutoCorrelation ;8   ;100  ;800
11 $L3(crc.c)           ;Calc_crc8          ;6   ;128  ;768
12 $_1(crc.c)           ;Calc_crc8          ;8   ;59   ;472
13 $_1(usr.c)           ;putStr             ;3   ;155  ;465
14 $L7(usr.c)           ;putStr             ;3   ;155  ;465
15 $L5(usr.c)           ;putStr             ;3   ;155  ;465
16 $L18(usr.c)          ;putx               ;8   ;56   ;448
17 $L2(crc.c)           ;Calc_crc8          ;6   ;69   ;414
18 $L20(usr.c)          ;putx               ;5   ;70   ;350
19 $L19(usr.c)          ;putx               ;4   ;70   ;280
20 al_printf            ;al_printf          ;15  ;18   ;270
21 th_printf            ;th_printf          ;10  ;18   ;180
```

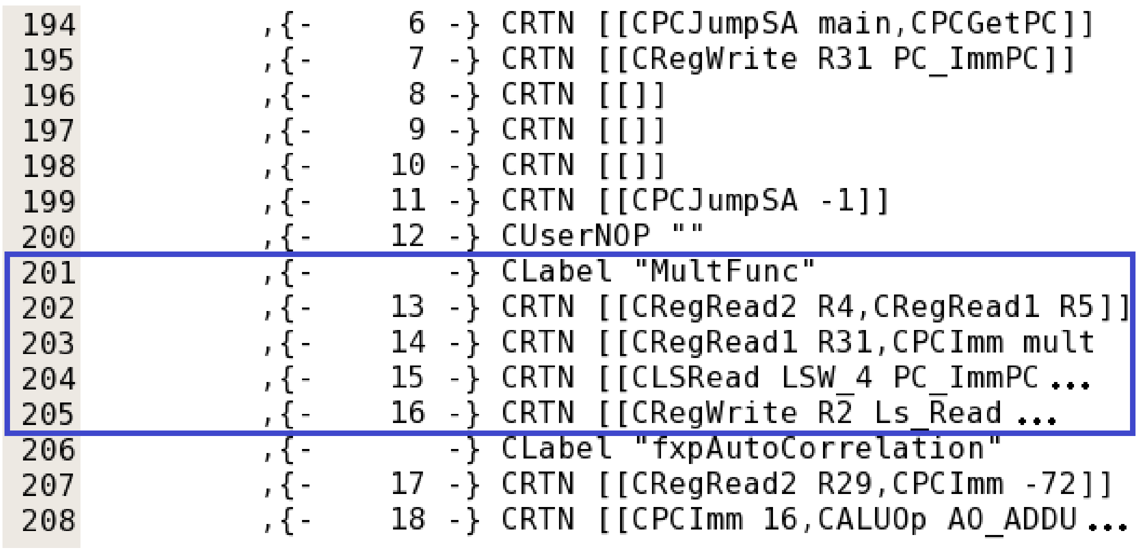
```
1  Profiling information:
2 =========================
3                       ;Function           ;Size;Count;Cycles
4 $L47(thal.c)          ;al_printf          ;13  ;626  ;8138
5 putChar               ;putChar            ;4   ;606  ;2424
6 $_20(thal.c)          ;al_printf          ;4   ;606  ;2424
7 $L66(thal.c)          ;al_printf          ;3   ;626  ;1878
8 $_19(thal.c)          ;al_printf          ;3   ;626  ;1878
9 $L62(thal.c)          ;al_printf          ;3   ;626  ;1878
10 $_3(autcor00mod.c)   ;fxpAutoCorrelation ;9   ;100  ;900
11 $L3(crc.c)           ;Calc_crc8          ;6   ;128  ;768
12 $L18(usr.c)          ;putx               ;8   ;88   ;704
13 $L20(usr.c)          ;putx               ;5   ;110  ;550
14 $L10(autcor00mod.c)  ;fxpAutoCorrelation ;5   ;100  ;500
15 $_1(crc.c)           ;Calc_crc8          ;8   ;59   ;472
16 $_1(usr.c)           ;putStr             ;3   ;155  ;465
17 $L7(usr.c)           ;putStr             ;3   ;155  ;465
18 $L5(usr.c)           ;putStr             ;3   ;155  ;465
19 $L19(usr.c)          ;putx               ;4   ;110  ;440
20 $L2(crc.c)           ;Calc_crc8          ;6   ;69   ;414
21 MultFunc             ;MultFunc           ;4   ;100  ;400
```

Figure 5.3*: Autcor 'profile' files.*
Based on the original C-code (left-side), based on the C-code with localized multiplier (right-side).

Profile of the benchmark shows information which concerns size, count, and consequently, cycles of each function. In order to retrieve such multiplier related information from the Profile, localization of the multiplier function is required to be accomplished. Through the content of Profile, information of multiplier behaviour can be collected which is the main reason for re-organizing C-code of benchmarks before compilation and simulation. Right-side of Figure 5.3 shows multiplication is localized as the separate function (MultFunc) in C-code benchmark (Line 21).

Furthermore, also from the profile, it is known that multiplier takes 4 cycles to execute its function and is used 100 times in Autcor benchmark. However, it is not clear yet when and how often exactly the multiplier is active in the benchmarks. In order to determine the cycle index when multiplication is executed and finished, we need a mapping step between several output files provided by FlexSim. For the sake of simplicity, we should have a look to understand how the multiplication execution is represented in RTN format.

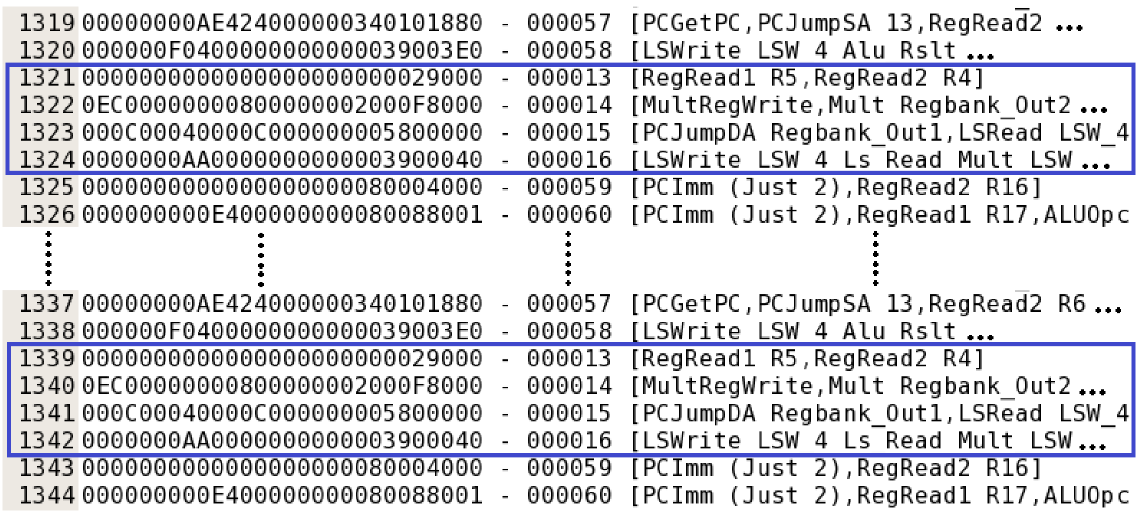
Option –SHOWCODE in the runmips command statically schedules the benchmarks and writes to a file, called *Showcode* file. This file presents every function of a benchmark, their execution representation in RTN format, and equivalent program counter (PC) value sorted in a numerical order. A short excerpt of the *Showcode* is shown in Figure 5.4.



```
194            ,{-     6 -} CRTN [[CPCJumpSA main,CPCGetPC]]
195            ,{-     7 -} CRTN [[CRegWrite R31 PC_ImmPC]]
196            ,{-     8 -} CRTN [[]]
197            ,{-     9 -} CRTN [[]]
198            ,{-    10 -} CRTN [[]]
199            ,{-    11 -} CRTN [[CPCJumpSA -1]]
200            ,{-    12 -} CUserNOP ""
201            ,{-       -} CLabel "MultFunc"
202            ,{-    13 -} CRTN [[CRegRead2 R4,CRegRead1 R5]]
203            ,{-    14 -} CRTN [[CRegRead1 R31,CPCImm mult
204            ,{-    15 -} CRTN [[CLSRead LSW_4 PC_ImmPC ...
205            ,{-    16 -} CRTN [[CRegWrite R2 Ls_Read ...
206            ,{-       -} CLabel "fxpAutoCorrelation"
207            ,{-    17 -} CRTN [[CRegRead2 R29,CPCImm -72]]
208            ,{-    18 -} CRTN [[CPCImm 16,CALUOp AO_ADDU ...
```

***Figure 5.4**: An excerpt from the Autcor 'showcode' file*

In Figure 5.4, multiplier function is labelled as 'MultFunc' (Line 201). From the Profile file, it is known that multiplier takes 4 cycles to execute multiplication function. Here, it is shown a representation of those 4 cycles in RTN format (Lines 202-205). Each cycle has an equivalent PC values that, in MultFunc case, are 13-16. These numbers are helpful to be known in cycle-accurate analysis, which is done by using FlexSim with an option, -TRACENISAD to generate a file, named Readable, simply because this file includes human-readable debug information. Figure 5.5 depicts a part of the Readable file of the Autcor benchmark where multiplier activity is presented.



```
1319 00000000AE4240000000340101880 - 000057 [PCGetPC,PCJumpSA 13,RegRead2 ...
1320 000000F0400000000000039003E0 - 000058 [LSWrite LSW 4 Alu Rslt ...
1321 0000000000000000000000029000 - 000013 [RegRead1 R5,RegRead2 R4]
1322 0EC00000000800000002000F8000 - 000014 [MultRegWrite,Mult Regbank_Out2 ...
1323 000C00040000C000000005800000 - 000015 [PCJumpDA Regbank_Out1,LSRead LSW_4
1324 0000000AA000000000000003900040 - 000016 [LSWrite LSW 4 Ls Read Mult LSW ...
1325 0000000000000000000080004000 - 000059 [PCImm (Just 2),RegRead2 R16]
1326 00000000E40000000080088001 - 000060 [PCImm (Just 2),RegRead1 R17,ALUOpc
 ⋮               ⋮                  ⋮                  ⋮
1337 00000000AE4240000000340101880 - 000057 [PCGetPC,PCJumpSA 13,RegRead2 R6 ...
1338 000000F0400000000000039003E0 - 000058 [LSWrite LSW 4 Alu Rslt ...
1339 0000000000000000000000029000 - 000013 [RegRead1 R5,RegRead2 R4]
1340 0EC00000000800000002000F8000 - 000014 [MultRegWrite,Mult Regbank_Out2 ...
1341 000C00040000C000000005800000 - 000015 [PCJumpDA Regbank_Out1,LSRead LSW_4
1342 0000000AA000000000000003900040 - 000016 [LSWrite LSW 4 Ls Read Mult LSW ...
1343 0000000000000000000080004000 - 000059 [PCImm (Just 2),RegRead2 R16]
1344 00000000E40000000080088001 - 000060 [PCImm (Just 2),RegRead1 R17,ALUOpc
```

***Figure 5.5**: An excerpt from the Autcor 'readable' file*

In Figure 5.5, the first 4-integer values on the left represent the cycle number. There are 19,553 cycles in total for Autcor benchmark. Second column consists of N-ISA instructions in hexadecimal format. Then, the third column depicts PC value and the fourth one – equivalent RTN format of each cycle.

As PC values of the multiplier function are 13-16, in Readable file they are traced as 000013-000016 values (Lines 1321-1324 & 1339-1342). As it can be noticed from this figure, here multiplier is active 2 times (out of 100 in Autcor application).

Consequently, multiplier IIC cycles are computed by summing all the non-multiplier cycles in between active multiplier cycles. In this example, there are 14 IIC cycles (Lines 1325-1338). However, in other time-domain positions (see, Figure 5.5), Autcor has 35 IIC cycles. Table 5.2 presents application profiling statistics including IIC data.

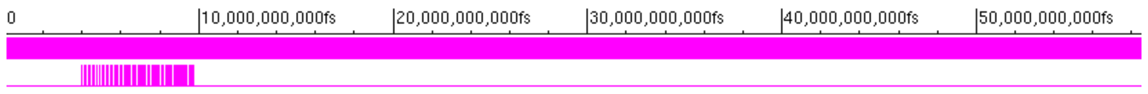
***Table 5.2**: EEMBC application profiling statistics including IIC*

Benchmark	Total cycles	MULT only		IIC	Total IIC	MULT with IIC	
		Cycles	% of total			Cycles	% of total
AUTCOR	19553	400	2.05	14, 34	1531	1931	9.88
FFT	162967	10240	6.28	1, 22, 47, 93, 2950	72148	82388	50.55

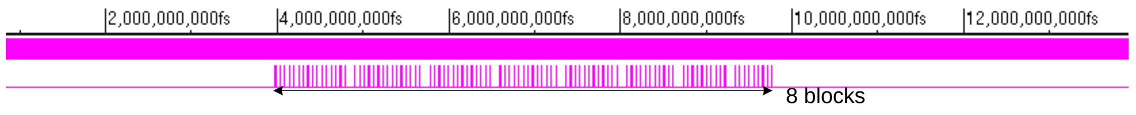
Activity of the multiplier including IIC consumes up to 1931 cycles, or 9.88% of total benchmark cycles. It is determined by summing all the cycles from the first multiplier activation to the last one. It also can be computed by adding total IIC of the multiplier to actual multiplier activity cycles. In FFT case, summation of the IIC and multiplier-only usage cycles (82,388) boosts the total multiplier usage up to 50.6% of total application cycles. A diverse usage of a multiplier in Autcor (9.88%) and in FFT (50.6%) benchmarks is one of the main reason of the selection of these applications. However, there is another important factor that differentiates Autcor and FFT – a number of multiplier's IIC which is irregular in both benchmarks.

Table 2 presents a range of IIC values. Instead of having one constant IIC value and, hence, knowing an exact number of idle cycles between each multiplier execution, there is an irregular number of IIC. Such irregularity of IIC complicates automatic application of power gating technique which would require taking into consideration switching power overhead. Without concerning switching power overhead, IIC parameter would not affect power gating complexity. Therefore, further analysis of both benchmarks related to irregular IIC is required in order to apply the most optimal power gating.

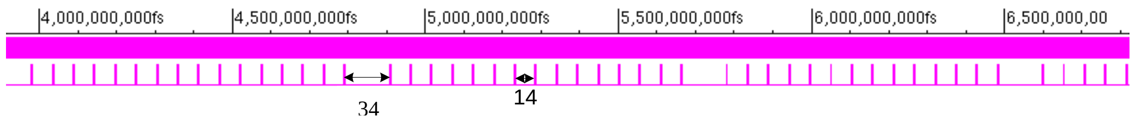
Figure 5.6 presents a graphical view of Autcor benchmark and its multiplier activity. Subfigure A shows the full span of the application with total cycles showed in upper line. The multiplier usage is presented in the lower line. Multiplier activity is illustrated according its enable signal being '1' for one-cycle in the N-ISA instruction while multiplier actually requires 4 cycles to complete its function. In subfigure B, it is shown all the multiplier executions that are 'separated' in 8 blocks, which few of them are zoom in and presented in subfigure C. Lastly, in subfigure D, one of the blocks is zoomed and depicted.



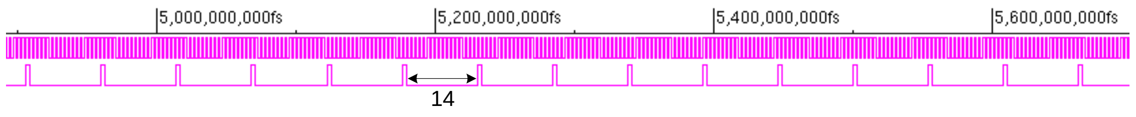
(A) Full view



(B) Multiplier zoom 1



(C) Multiplier zoom 2

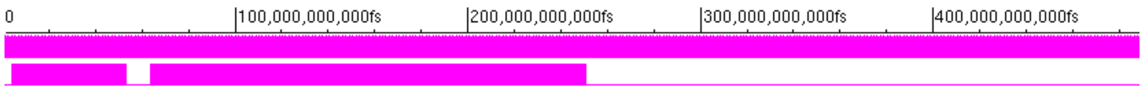


(D) Multiplier zoom 3

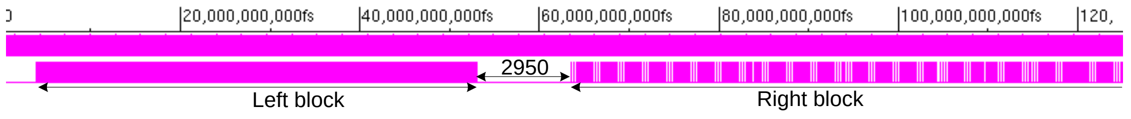
***Figure 5.6**: Multiplier usage in EEMBC benchmark, Autcor.*
Upper line in each subfigure represents cycles and lower – multiplier activity

Notice that there are two IIC in Autcor benchmark (Figure 5.6.C). The larger ‘distance’ between two multiplier executions is 35 cycles, which is what is meant by ‘separating’ the multiplier activity in 8 sub-blocks. The smaller distance, which is also shown in Figure 5.6.D, is 14 cycles – the minimum IIC of the benchmark.

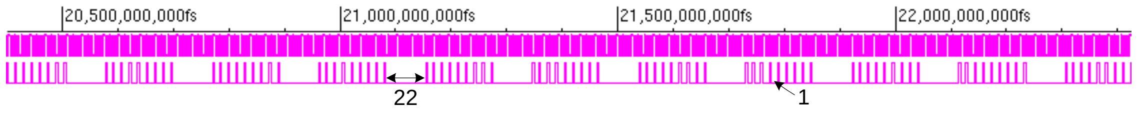
The graphical view of the FFT application is shown in Figure 5.7. The entire multiplier usage in FFT (50.6%) is presented in subfigure A. FFT multiplier activity is ‘divided’ into two blocks that are closer zoomed in subfigure B to show the difference between them. Each block is presented in further subfigures: the left block in subfigure C and the right one – in D.



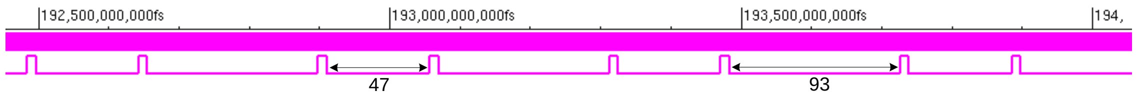
(A) Full view



(B) Multiplier zoom



(C) Left block zoom



(D) Right block zoom

Figure 5.7*: Multiplier usage in EEMBC benchmark, FFT.*
Upper line in each subfigure represents cycles and lower – multiplier activity

Most attention in FFT multiplier usage requires the difference between the left (Figure 5.7.C) and the right (Figure 5.7.D) blocks. Each block has two different IIC which makes FFT benchmark to have 5 different IIC. The fifth one is the largest that separates the blocks and it contains 2950 cycles (Figure 5.7.B). The left block holds IIC of 1 and 22. The right block contains 47 and 93.

5.5.2 Tracing Multiplier Instructions

Method 1

In parallel with computing the IIC count based on *Profile*, *Showcode*, and *Readable* files, a trace of multiplier instructions is performed using N-ISA code file generated by the FlexSoC simulator. Multiplier instructions can be traced only using hexadecimal N-ISA instruction. The original N-ISA consists of 109 bits of control signals for each functional unit in the FlexCore. Figure 5.8 shows the mapping of the N-ISA.

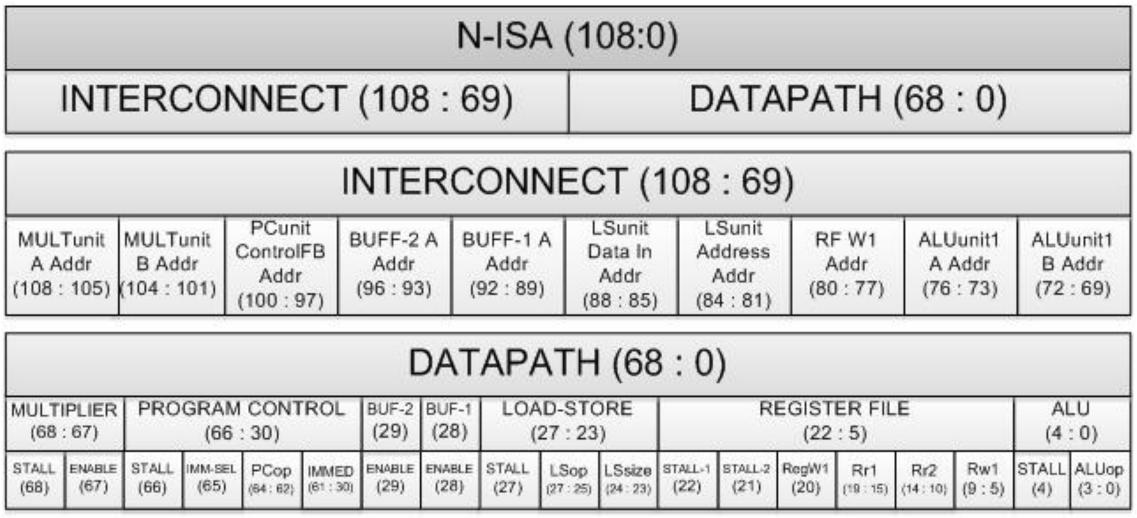


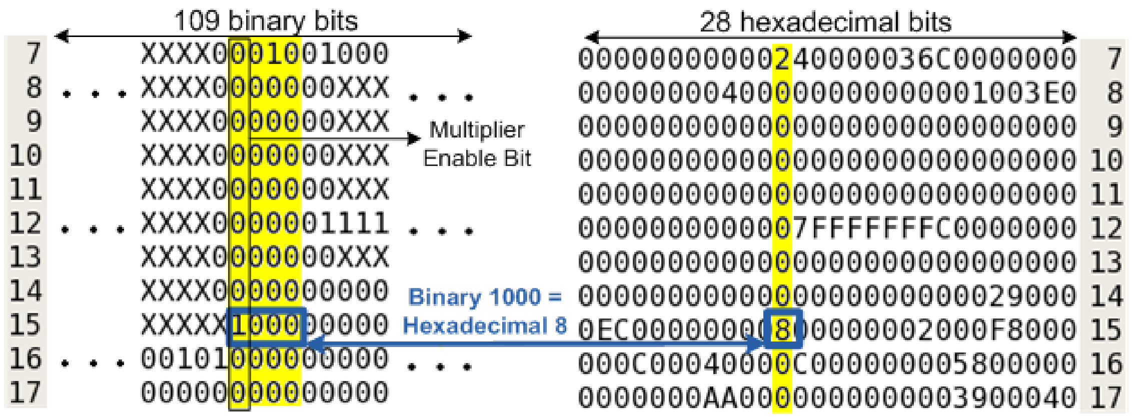
***Figure 5.8**: N-ISA instruction mapping*

As mentioned previously, N-ISA instruction consists of address bits for interconnect and control bits for datapath units. The last 8 LSB are reserved for the multiplier units' addresses (108:101) in the *interconnect*. In the *trinary* N-ISA format, the only instruction, that has other than X value in multiplier address, is on Line 15 where multiplier executes its function (Figure 5.9). Datapath has two bits related to the multiplier functional unit: stall (68) and enable (67). Enable signal is a key point of this multiplier instruction tracing method.

Every time the multiplier is active, enable signal is triggered to '1' in trinary format, otherwise, it is set by ‘0’. Therefore, according to the enable signal of the multiplier, activation of the multiplier can be traced by only using the N-ISA instruction.

Method 2

FlexSim generates N-ISA instructions in two formats: in hexadecimal and in trinary codes. Data files for both formats are identical. Trinary format consists of 109 bits of binary 0 and 1, and X (don't care) values. Hexadecimal code requires less storage space by consisting of 28 bits of hexadecimal values (0-9 and A-F). Examples of both formats of the same N-ISA instruction are presented in Figure 5.9.



```
 7      XXXX0001001000        000000000002400000036C0000000  7
 8 . . . XXXX0000000XXX . . .  00000000400000000000001003E0  8
 9      XXXX0000000XXX        0000000000000000000000000000  9
10      XXXX0000000XXX        0000000000000000000000000000 10
11      XXXX0000000XXX        0000000000000000000000000000 11
12 . . . XXXX0000001111 . . .  00000000000007FFFFFFC0000000 12
13      XXXX0000000XXX        0000000000000000000000000000 13
14      XXXX000000000000      0000000000000000000000029000 14
15      XXXXX10000000000      0EC00000000080000002000F8000 15
16 . . . 0010100000000000 . . . 000C00040000C000000005800000 16
17      0000000000000000      0000000AA0000000000003900040 17
```

***Figure 5.9**: Several instructions of Autcor N-ISA code*
Trinary (left) and in hexadecimal formats (right).

N-ISA instructions in both formats are not completely identical as instructions in the Readable file (Figure 5.5, 2nd column) per cycle. Here, N-ISA instructions are wrapped and do not include repetitive cycles in loops. Therefore, it is impossible to detect all multiplier activities using only N-ISA code. For example, in Autcor N-ISA instruction, multiplier is instructed to be activated only once (Figure 5.9, Line 15) while the benchmark actually uses multiplication function 100 times. The N-ISA instruction in Line 15 is traced from the Readable file where it is identical to the instructions in Lines 1321 & 1340 (Figure 5.5). It means that multiplier performs its function at Line 15.

N-ISA instruction is a compact version of the Readable file excluding PC values and RTN-format notations in order to avoid an excessive amount of instruction bandwidth and a large memory footprint. Subsequently, because of its lower bandwidth, hexadecimal N-ISA instruction format is standard input to the RTL level power estimation where N-ISA is converted to the trinary format. Trinary N-ISA code is also useful in another method of tracing multiplier instructions.

5.6 N-ISA Extension for Power Gating

In order to introduce a multiplier power gating control bit and a power gating mode control bit, the instruction is required to be extended to 111 bits. Purpose of the power gating mode control is to selected either software or hardware power gating control mode (more information in Section 6.3). The flow of the procedure in order to activate a power-gated multiplier by extending the N-ISA instruction is shown in Figure 5.10.

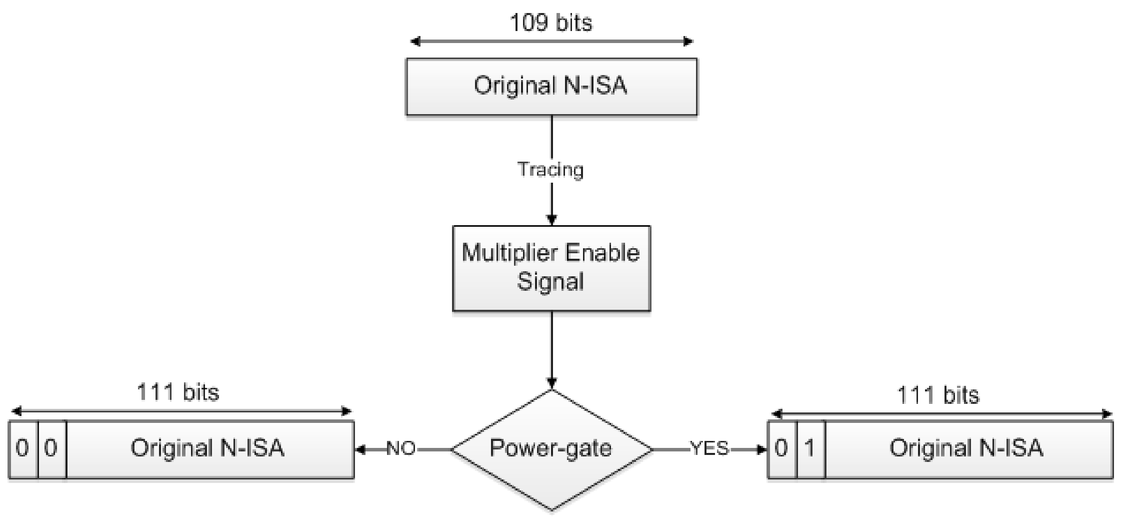
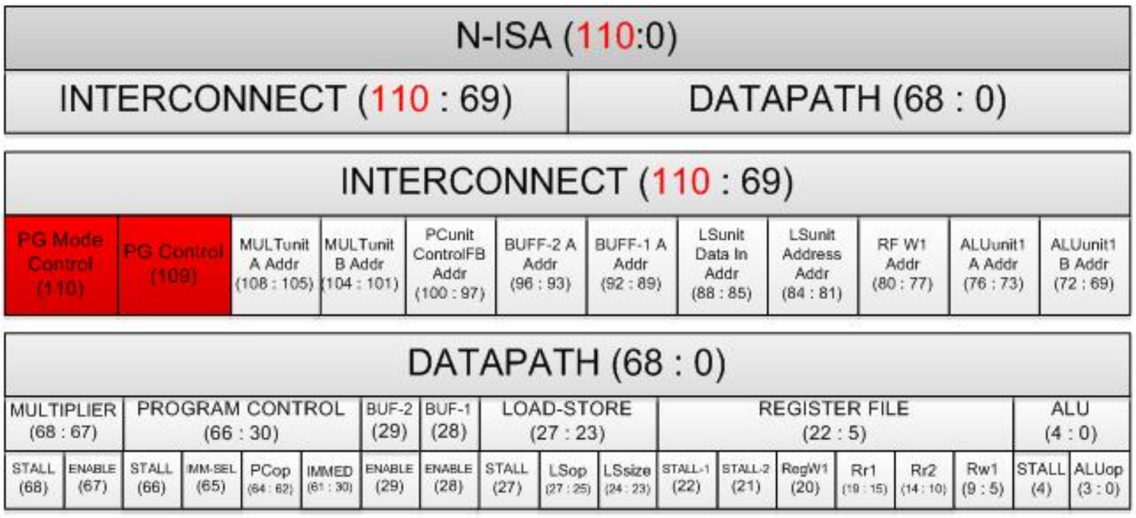


***Figure 5.10**: Applying power gating control signal*

Hence, a structure of the new extended N-ISA instruction with extra power gating bits is shown in Figure 5.11.



N-ISA (110:0)	
INTERCONNECT (110 : 69)	DATAPATH (68 : 0)

INTERCONNECT (110 : 69)											
PG Mode Control (110)	PG Control (109)	MULTunit A Addr (108 : 105)	MULTunit B Addr (104 : 101)	PCunit ControlFB Addr (100 : 97)	BUFF-2 A Addr (96 : 93)	BUFF-1 A Addr (92 : 89)	LSunit Data In Addr (88 : 85)	LSunit Address Addr (84 : 81)	RF W1 Addr (80 : 77)	ALUunit1 A Addr (76 : 73)	ALUunit1 B Addr (72 : 69)

DATAPATH (68 : 0)																					
MULTIPLIER (68 : 67)		PROGRAM CONTROL (66 : 30)				BUF-2 (29)	BUF-1 (28)	LOAD-STORE (27 : 23)			REGISTER FILE (22 : 5)						ALU (4 : 0)	
STALL (68)	ENABLE (67)	STALL (66)	IMM-SEL (65)	PCop (64 : 62)	IMMED (61 : 30)	ENABLE (29)	ENABLE (28)	STALL (27)	LSop (27 : 25)	LSsize (24 : 23)	STALL-1 (22)	STALL-2 (21)	RegW1 (20)	Rr1 (19 : 15)	Rr2 (14 : 10)	Rw1 (9 : 5)	STALL (4)	ALUop (3 : 0)

***Figure 5.11**: Extended N-ISA*

6. Hardware Implementation

Hardware implementation flow begins with the given RTL code for the original FlexCore design (RTL-FlexCore). RTL code for the power control module and the RTL-FlexCore are defined in another top-level module. The power intent in a low power design is specified through the common power format (CPF) file. The synthesis tool takes both RTL and the CPF files and produces the new netlist in which the required low power methods are applied. Certain aspects of the architecture, such as insertion of power switches will be added in the next step of physical implementation. The function of the power-gated design is verified after synthesis and physical implementation with the testbench used for the original design and the power gating functionality is verified with Cadence Incisive Simulator. Figure 6.1 gives an overview of the flow. This section gives a brief about the common power format, the RTL synthesis and the implementation phases.

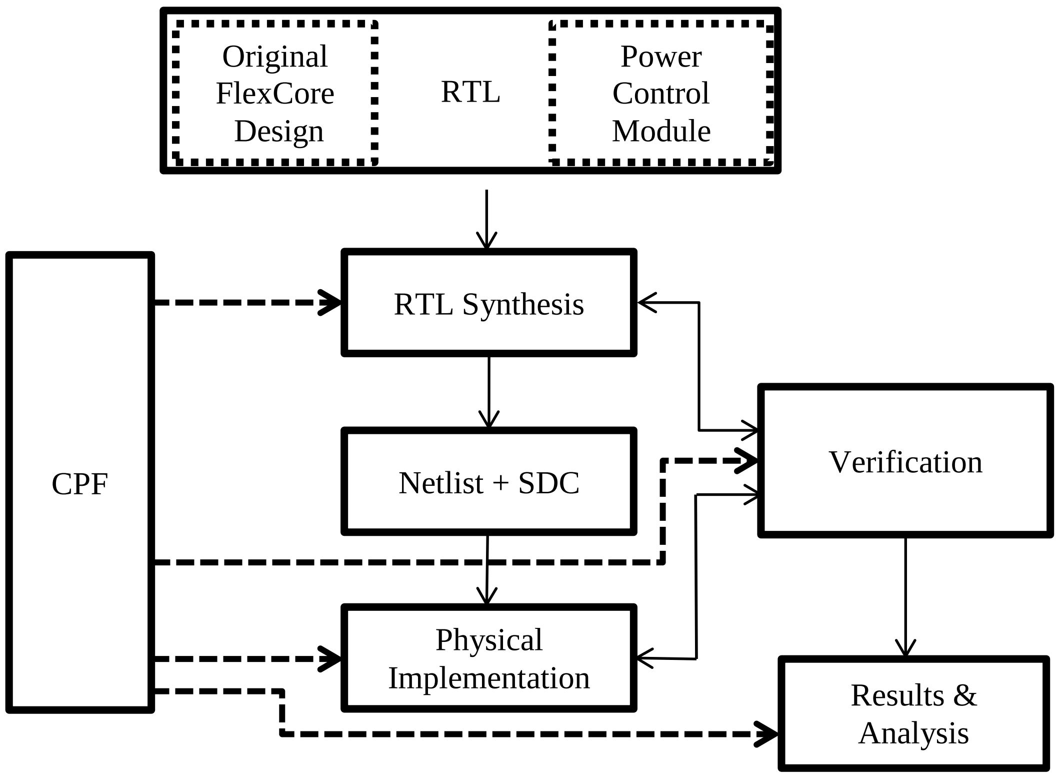


Figure 6.1*: Hardware implementation flow*

6.1 Common Power Format

The power intent of a design is specified through the CPF file [5]. It is a Si2 supported format for specifying the power architecture of a design. The early version of CPF was designed by Cadence. A single CPF is used throughout a digital design flow right from the RTL synthesis to physical implementation and by various verification and estimation tools. CPF defines the specific power architecture that has to be applied to a design such as *multi threshold, power gating, multi supply* or all of these. No modifications are required to be made to the original design. Tools read the original design and its CPF to perform the required changes at the synthesis and physical implementation phase. The CPF file structure begins with specifying the libraries and then the nominal conditions and rule for usage of special cells. The structure is shown in Figure 6.2 and briefly discussed as follows [10]:

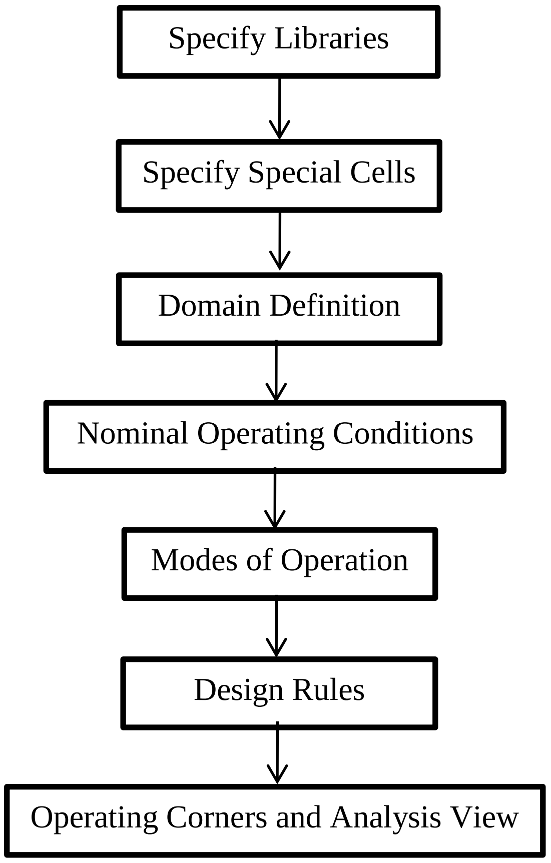


***Figure 6.2**: CPF specification structure*

1. Specifying Libraries: The set of libraries used for design are specified here. The library set specified includes all the libraries for the normal synthesis step and special cells (power switch, isolation, state retention cells) to be used with the chosen low power technique. A library set also contains multiple threshold libraries which are used in implementing multi-V_{TH} technique. Each library set is given a name and referred later in the file to associate a set of rules. Library sets for different operating corners can also be specified and each set is referred with their unique name.

2. Specifying Special Cells: All specials cells to be used in the design like the power switch cells, isolation cells, state retentions cells and level shifters are defined here. The tools searches for the specified cells from the libraries defined in the previous step. For power switch definition, parameters such as associated power nets, the type of the switch cell (header or footer) and the switch enable pin are specified. Similarly for isolation and state retention cells their enable pins and their valid placement locations are specified.

3. Domains: A domain is a group of instances that operate on the same supply voltage or is switched off in a power gated design. A default domain has to be specified. All instances unless specified to be in defined domain, will belong to this default domain. Section 4.1 explains the power domain split for the power

gated FlexCore design. Only two domains are present in the CPF for this work, one is the default and the other domain is for the multiplier. A shut-off condition must be associated with a domain definition which is used for power gating.

4. Nominal Operating Conditions: The possible operating voltages that different domains operate at are defined as the nominal operating conditions. In multi supply or dynamic voltage scaling design the chip or domains operates at different voltage levels which are defined as nominal conditions. In a power gated design there are only two nominal conditions i.e. when a domain is turned ON (operate at the chip voltage) and turned OFF (0).

5. Modes of Operation: This step defines stable modes of operation for the design in which each domain operates at one of the nominal operating conditions specified in the previous step. A default mode has to be defined similar to a default domain. The design operates in the mode until conditions for other modes are satisfied. In a power gated design, the off state nominal condition need not be specified as a 0 voltage level. Domains that are not associated with any nominal operating condition in a mode are considered to operate at 0 V. Table 6.1 shows the domains and modes defined for this power gated FlexCore design.

6. Design Rules: The design rules specify how the special cells have to be used. The specification includes, the type of special cell that have to be used from the different special cells defined in step 2. For isolation cells the condition for isolation, the domains associated and their location and for their insertion is defined. Isolation cells can be inserted either at the input domain or the output domain. For the power switches the associated domain and nets are defined. State retention rules specify the registers that have to be replaced with the state retention cells (retention flops).

7. Operating corners & Analysis view: Operating corners specify a set of process, voltage and temperature values under which the design must operate successfully. Analysis view associates these operating corners with a mode of operation (step 5). The set of active views represent the different design variations that will be timed and optimized through a multi-mode multi-corner (MMMC) analysis.

Apart from the above main steps few other specifications are also present in the CPF.

- Power and ground nets considered for the design are defined. In this power gated design 3 power nets and 1 ground net is defined, which are shown in Table 6.2

- Global connections for pins to the power and ground nets are defined. V_{DD} Pins in the *PD_mult* domain are connected to the Virtual V_{DD} (VDD) power net and the V_{DD} pins for the rest of the design (PD_default) are connected to the True V_{DD} (TVDD). The gate inputs of the PMOS switches and the gate control single from the switch control cell are connected to the SWVDD power net (Table 6.2).

- Rules for transition between the modes specified at step 5 can be defined when a dynamic voltage scaling is to be implemented.

- Timing constraints and activity information for each modes of operation is defined. The timing constraints are specified through the SDC file.

- Dynamic and leakage power targets can also be specified.

- Defining '*Always ON*' cells in a power gated domain: If any cell in the power gated domain is required to be ON while rest of the domain is turned OFF, then it is to be defined as an '*Always ON'* cell. During the physical implementation these cells are connected to the True V_{DD} instead of the Virtual V_{DD} of the domain through special routing. Too many such cells can lead to complexity in placement and routing, in which case the power architecture must be reconsidered.

Table 6.1*: Power domains and modes for the power gated FlexCore design*

Power Mode	**Power Domain**	
	PD_default	PD_mult
PM_default	1.0	1.0
PM_mult_off	1.0	0.0

Table 6.2*: Power and ground nets for the power gated FlexCore design*

Net	**Type**	**Description**
TVDD	Power	Global or true V_{DD}
VDD	Power	Virtual or PD_mult domain V_{DD}
SWVDD	Power	PMOS switches gate control
VSS	Ground	Global ground

6.2 RTL Design and Synthesis

The only RTL design required for the implementation of the power gated FlexCore design is the power control module. No modifications are made to the original design, except for addition of isolation enable (input), power switch enable (input) and transition complete (output) signals to the port list of the top level FlexCore HDL. For evaluation purpose the current control bits for the *switch control* cell are defined as external inputs. Similarly the test signal associated the *switch control* cell are also defined as external signals.

The synthesis is carried out using Cadence RTL Compiler. The synthesis flow has few additional steps to read the CPF file and apply the power intent to the design. The flow used in this work is based on the recommended low power design flow in [11]. The flow is shown in Figure 6.3. Synthesis is performed for the two library types (standard V_{TH} and low V_{TH}) of the provided 65 nm technology. For each library type synthesis is performed for and typical and worst case operating conditions[3].

[3] Refer section "Tools and Technology"

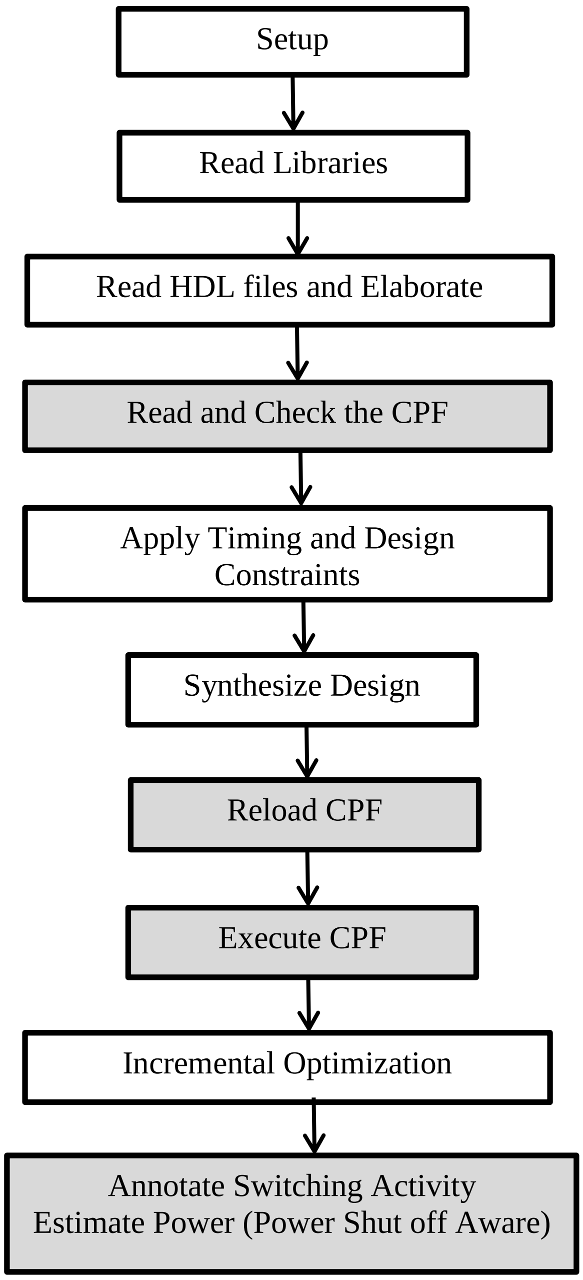

Setup
Read Libraries
Read HDL files and Elaborate
Read and Check the CPF
Apply Timing and Design Constraints
Synthesize Design
Reload CPF
Execute CPF
Incremental Optimization
Annotate Switching Activity
Estimate Power (Power Shut off Aware)

***Figure 6.3**: RTL synthesis flow*

The flow begins with the setup of library locations, HDL locations and setting attributes required for the synthesis. It is followed by the normal steps of reading the libraries, reading the HDL files and then elaborating them. Read and Check CPF step reads the specified CPF file for the top level module and verifies the syntax and definitions. The libraries specified in the CPF are read and the successfully read libraries are reported. The domains specified are created in the hierarchy and instances are grouped to their respective domains. Domains are associated with their respective libraries as defined in the CPF. The timing constraints and activity information if specified in the CPF are read. Isolation, state retention and library set definitions successfully read are reported. Power switch insertion is not carried out in the RTL compiler and hence the power switch definition is report to be unsupported. The report also indicates the number of usable isolations cells, retention cells and level shifters[4] for the design. It also lists the unusable cells. Finally a list of isolation rules or retention rules that have to be applied is created.

The timing and other design constraints are applied and the actual synthesis step is carried out. Cells from the libraries are mapped to their corresponding domain. During synthesis, the state retentions cells replace the specified registers. The CPF file is reloaded to ensure that isolation cells are considered as specified in the CPF file. Synthesis step can add new nets or pins to the netlist and isolation rule might be applicable to some of these nets or pins. The CPF is reloaded to ensure that all locations are considered. When the CPF is executed the isolation cells are inserted in the netlist. The number of isolation cells and their locations (input or output nets/pins) are reported. The report can be checked to verify isolation cells are inserted as the user intended in the CPF. In this design, isolation cells are inserted between the outputs of the multiplier and the isolation outputs are connected to the registers. A total of 64 isolation cells are inserted, which includes two enable signals for the registers (2*32 bit + 2). The registers work on an active-high enable signal and hence a '0' isolation output for the enable signals are required so that the floating values at the multiplier output when it is shut down, is isolated from the output registers. An AND type or NOR type isolation cells are thus needed. All isolations cells having the same enable signal, from and to domain, and of the same cell and library type are grouped into a hierarchical instance.

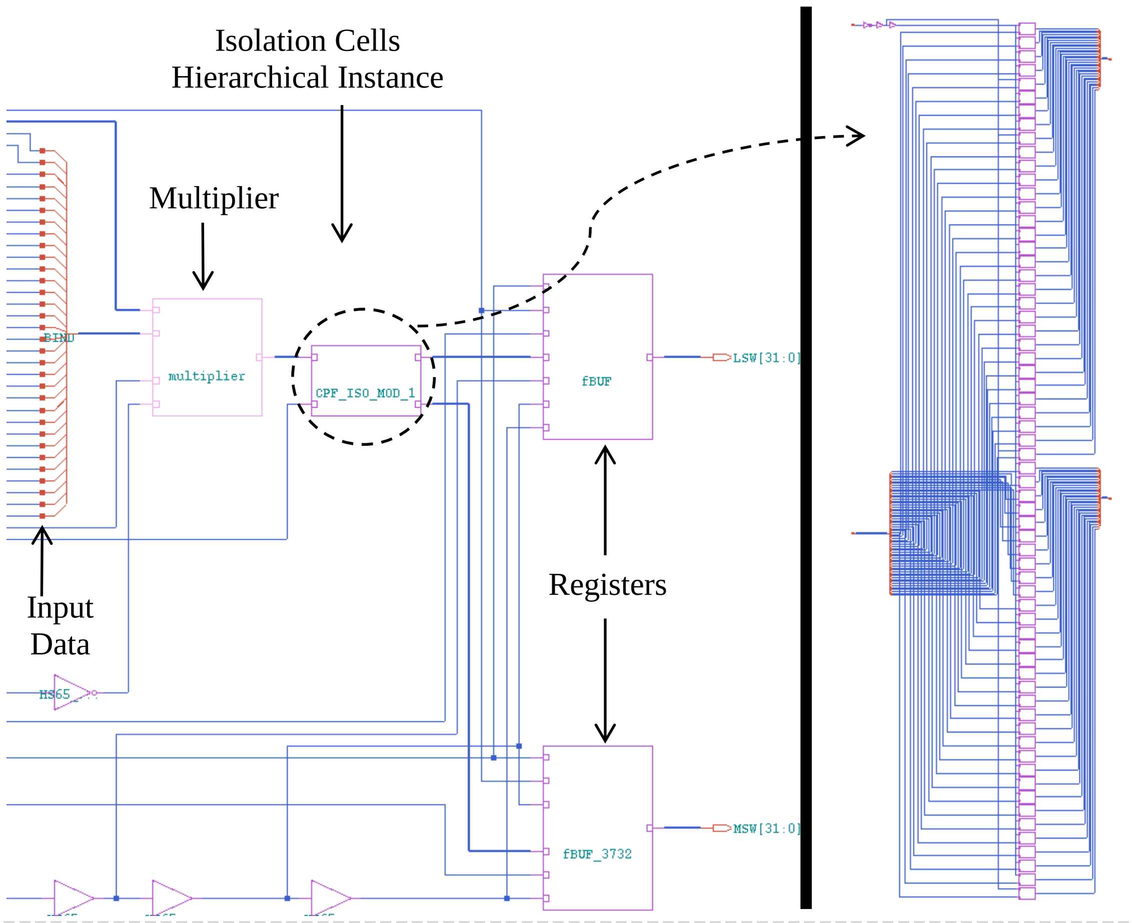


***Figure 6.4**: Schematic view of the synthesized netlist showing isolation cells hierarchy*

[4] There is no level shifter or retention rule specified in the design and hence no definitions or usable level shifters cells are reported.

Figure 6.4 shows the hierarchical instance of isolation cells inserted between the output of the multiplier and the input of the registers. All 64 isolation cells inserted in this design and grouped into this hierarchy. Figure 6.5 shows the isolation cells inside the hierarchy and their common enable signal. Level shifters if need (multi supply voltage design), are also defined in this step similar to isolation cells.

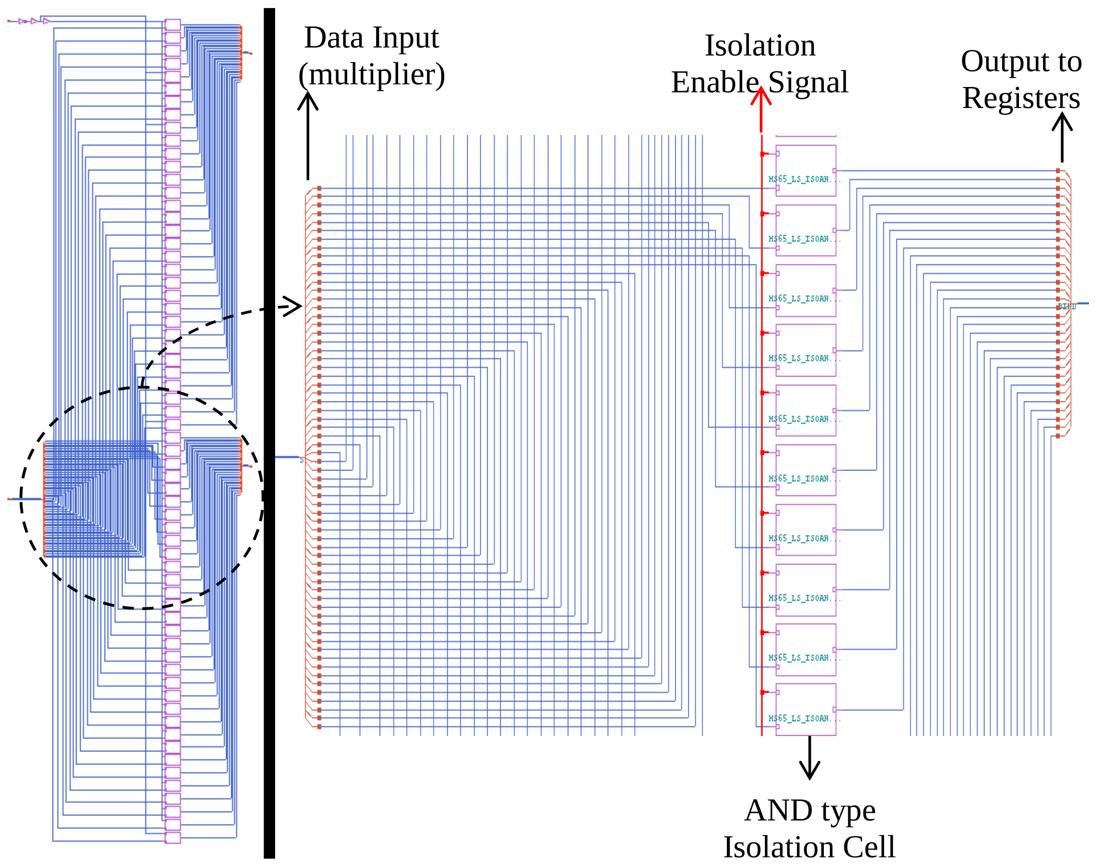


***Figure 6.5**: Schematic view of the isolation cells in the hierarchical group*

Power estimation is reported after reading the activity file for the chosen benchmarks in each case of library type and operating condition combination. The estimation is same as in any design flow except that an attribute has to be set to indicate that a power shut off estimation is made. Also attribute has to be to set indicate if the activity is power shut-off aware or not. The tools estimates the power considering the power down domain and the activity for the wake-up and shut-off is derived from the power domain shut-off condition specified in the CPF domain definition.

6.3 Power Control Module

An additional on-chip power control module was designed for the existing flex-core. The power control module can operate in three different modes which can be controlled from the NISA binary instruction. The 110 and 109 bits of NISA are used to set the operation of the power control module to the required mode (Figure 6.6).

NISA - Extended 110 Mode Control	**NISA - Extended 109** Switch Control in Mode 0	**Original NISA 108:0**

0X – Software Control – Mode 1
10 – Software + Hardware Control – Mode 2
11 – Hardware Control – Mode 3

Figure 6.6*: NISA extended bits for power control and their operation*

Mode 1 (01/00): In this mode the power control for the multiplier is directly received from the NISA. Each NISA instruction will have the ON or OFF state for the multiplier. The 111th bit is '0' in this mode and the bit 110th bit is either '1' or '0' signalling the '*turn-ON*' or '*turn-OFF*' of the switch. The NISA power gating bit is set to "*switch enable =1*" one cycle before the actual multiplication. The hardware takes care of the '*turn-OFF*' of the switch by delaying the turn off for the required number of cycles. The Multiplier takes two cycles for multiplication and another cycle of transferring the results to the registers outside the domain *PD_mult*. The hardware delays the "*switch-enable = 0*" signal condition four cycles after the 109th bit of NISA goes to '0'. For this mode to operate efficiently, evaluation and modification of the input NISA instructions is required based on the application profile, which is main aim of the software part of this work.

Mode 2 (10): In this mode the power control module enables and disables the switches based on some stored values. The first option is to store one or more threshold cycle. By threshold cycles we mean the minimum idle cycles between consecutive usages of the multiplier such that there is an optimal power gain or in the worst case there is no overhead in the power. The power control module counts the number of idle cycles of the multiplier, once the counter exceeds the threshold cycle the switch is disabled. If the multiplier is to be turned ON immediately after the threshold cycle then this mode is not effective[5]. But for the benchmarks considered here such a case does not appear. The value stored must be a higher than the maximum threshold of a set of applications considered to be running on the processor, which will be very inefficient if the idle cycle profile of these applications differ to a great extent. If it is possible to choose an optimal threshold value that is in good agreement with the entire application set considered, then a run-time software analysis on the idle cycle is not required. The second option is to receive the threshold value from the NISA instruction. The compiler must have additional functionality to analyse idle cycles of the application being executed and send the threshold cycle to the power control module either at the beginning of the execution. This data can be received through some serial transaction protocol, after setting the mode to 10.

Mode 3 (11): In this mode the power control module generates the switch enable signal directly checking the multiplier enable bit from the NISA instruction input. Which

[5] In Mode 01/00 i.e. software control, idle cycles up to threshold value is not lost. Analysis on the entire idle cycle profile is performed prior to the execution and hence power gating bit is enabled at the beginning of multiplier idle cycle.

means that the multiplier is turned ON as and when its required. But the switch needs to operate with the least possible transition time. The switches used here have a transition delay of 1 ns in the best case. So this mode is suitable for the 3.5 ns clock period considered here. But the rush current will be high. Similar to the software mode, during shut down the switch is disabled 4 cycles after the multiplier enable bit goes low. A 4 cycle delay is to ensure that he multiplier output is successfully read by the registers. But no effort is made to check the idle cycles. This mode is only suitable for testing and verification. This mode was initial power control operation designed before software support was available. The mode was preserved after the power control module was redesigned to have the above two modes.

6.4 Physical Implementation

The physical implementation of the power gated design of the FlexCore processor has mainly three additional steps comparing to the conventional flow. Figure 6.7 shows the flow of the entire physical implementation.

As highlighted in Figure 6.7 three additional steps involved in the physical implementation phase of a low power design are

- Placement of special cells
- Defining special power network
- Routing special nets.

These steps are explained in detail along with the brief about the other steps.

- The flow begins with a general setup, which includes reading the netlist, timing constraints, power architecture (CPF) and libraries. In the floor planning step instances under the same module are grouped into regions and placed similar to the block diagram view in Figure 4.6. The two domains defined are clearly visible and all instances of the multiplier module are automatically placed in the *PD_mult* domain. Partitions can be defined for the separate domain (PD_mult). This will help carrying out the further steps separately for the two domains. The isolations cells are grouped into a fenced block and placed close to the *PD_mult* domain[6]. During floor planning it is ensured that enough space is available around the multiplier domain to place the power switches and hence a halo of required dimensions are defined for the *PD_mult* domain.

- Placement of Special Cells: Specials involved in a power gating design are the power switches, switch filler cells, switch corner cells, isolation and state retention cells. Isolation and state retention cells are added to the netlist in the synthesis phase as explained in 6.2. The actual placement of these cells is carried out during the step of "Placement of Standard Cells"[7].

[6] Floor planning for level shifters are also carried out similarly

[7] Level shifters are also inserted into the netlist during synthesis phase and are placed along with other standard cells similar to isolation cells.

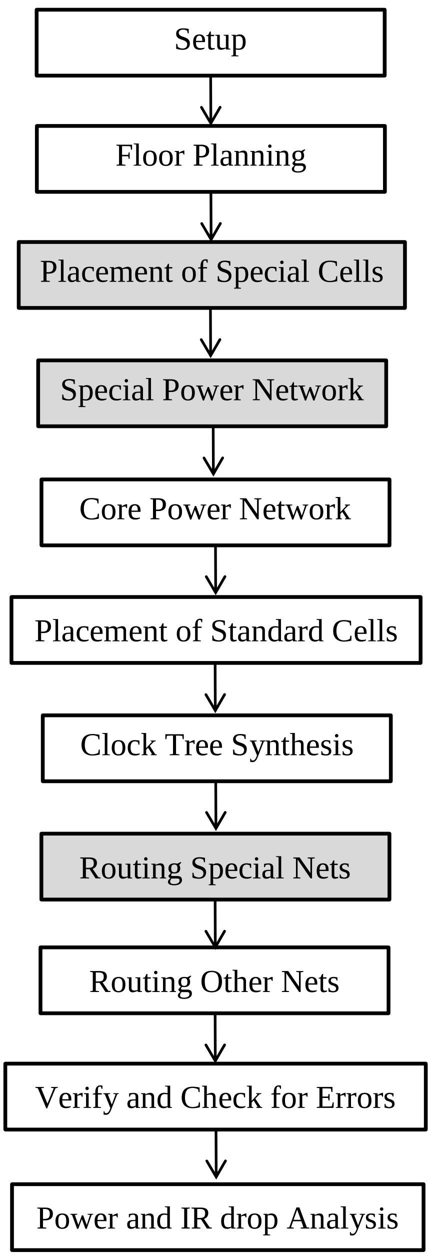


***Figure 6.7**: Physical implementation flow*

- The power switching is made of two types of cells as explained before in 4.1. One is the switch control cell (VDD-CTRL) and the other is the actual PMOS header switch (VDD-SWITCH) cell. They are placed with two separate '*addPowerSwitch'* commands. Switch filler and corner cells are automatically placed with the same '*addPowerSwitch'* command. The connection of cell pins to their corresponding nets is also specified with the same command. The switches are arranged in a ring fashion. VDD-CTRL (switch control) is placed on the left side of the domain and the VDD-SWITCHES (PMOS header switches) are placed on the remaining sides. The switch control can also be placed as a separate instance outside the ring and allow switches to be arranged on all sides of the domain. But for power network simplicity the switch control is placed within the ring.

Additional isolation or level shifters can be placed if required. The switch control cell provided by the library typically operates at 1.2 V and the rest of the cells at 1.0 V. In this case level shifters should be placed for the switch enable, control and valid state indication signals. However in this work the operating voltage of the switch control cell is limited to 1.0 V as it can operate between 0.72 V – 1.43 V.

- Special Power Network: This step defines power rings and stripes around the *PD_mult*. The rings are defined for the power connection to cells within the domain and also ring type switches placed around it. In total 7 rings of 4 different power/ground nets are added. A virtual V_{DD} (V_{DD}) and ground (V_{SS}) are added for connection to cells in the domain. Global V_{DD} or true V_{DD} (TVDD), virtual V_{DD} and two V_{SS} rings[8] are added for the ring switches. As explained before the switch control cell controls the gate terminal of the PMOS switches (VDD-SWITCHES), which is also defined as a power net and a hence ring is added (SWVDD) for this connection. This also defined in the CPF file and shown in Table 6.2.

  The rings placed on the switches are automatically connected to the metal lines on the VDD-SWITCH and FILLER cells by the tool. To connect the pins of the VDD-CTRL block, the special route command is used (*sroute*). Blockages must be added appropriately to avoid unnecessary routing, which can lead to routing congestion in the later steps.

- In the next step the core power network is defined as in a regular design by adding supply (TVDD) and ground (VSS) rings around the core and stripes with regular spacing. The TVDD stripes are limited to the *PD_default* domain. Placement of standard cells is carried out in the usual manner, with pre place and in place optimizations. The isolation and state retention cells are inserted into their corresponding fence blocks. Clock Tree Synthesis is the next step and carried out in the same manner as performed for the original design.

- Routing Special Nets: The enable signals for the switching and isolation cell along with other control cells have to be routed first and fixed, as they are of high priority. The nets include the isolation enable signal and the enable, current control bits and testing control bits for the VDD-CTRL cell. After applying a special route on these nets, their property is change to “fixed”.

- Remaining nets are routed and the design is checked for errors. The RC extraction is performed, followed by power and IR Drop analysis. The netlist at this step is extracted and verified with the chosen benchmarks using the Cadence Incisive Simulator similar to the verification after RTL synthesis.

[8] The second VSS ring is to connect the Virtual VSS and SWGND pins of the switches and control cell. These pins are used only if a footer cell is to be used and thus not considered in this design.

Figure 6.8 shows the view of the multiplier domain, the switches and the control cell around it (ring) and the special power nets routed around it.

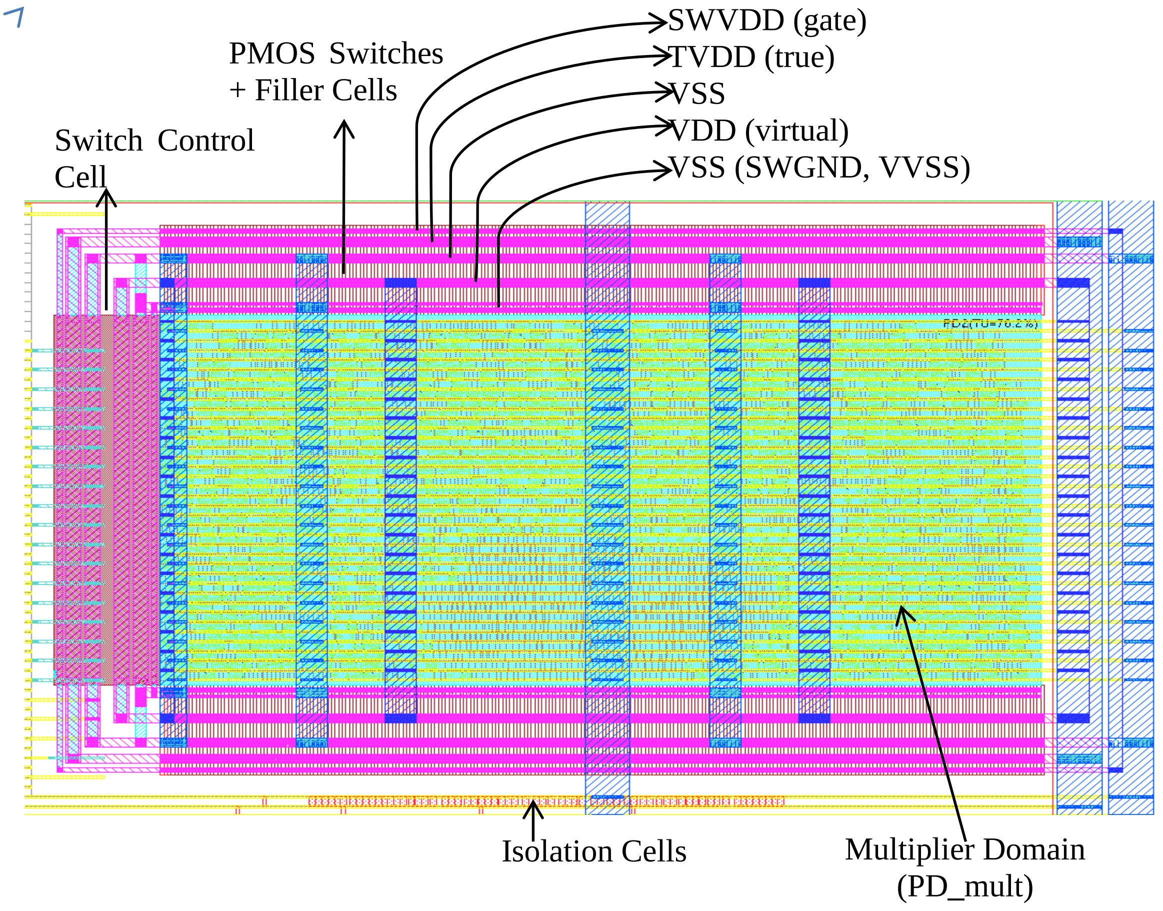


***Figure 6.8**: Physical view of the multiplier domain, switches, isolation cells and power/ground nets*

7. Results and Analysis

This section discusses the results that were obtained at the RTL synthesis and physical implementations for two different libraries and two operating conditions. The results presented are for the *Autcor* and *fft* benchmarks as explained in Section 5.2.

The libraries used are:

- *GL* - *General* Purpose *Low* Voltage Threshold
- *GS* - *General* Purpose *Standard* Voltage Threshold

Operating conditions are denoted throughout as

- *Nom* - Nominal or Typical: 1.0 V | 25 C
- *WC* - Worst Case or Maximum Timing: 0.9 V | 125 C

The results section begins with the early power saving analysis from the power gating of the multiplier after the RTL synthesis step. Further as estimates of the power switches are obtained after the physical implementation the actual possible power savings are estimated. The entire design is simulated at a clock period of 3500 ps, which is approximately 285 MHz.

7.1 Power Reduction Estimation after RTL Synthesis

The power reduction estimation after RTL synthesis is mainly performed as an early evaluation of how much power can be saved with the chosen power reduction technique. In the case of power gating technique, if the average switching overhead is known beforehand, the savings estimated would be a closer approximate to the practical case. However the switch overhead is not known prior to this work. The post RTL synthesis power reduction estimations serve the purpose of a comparison with the later estimations.

This section shows the maximum power reduction that can be achieved with power gating of the multiplier unit of the FlexCore. This power reduction is estimated without considering the dynamic or switching power of the switch control cell (VDD-CTRL). Hence these results can be considered as ideal. The results give a picture of how much reduction can be achieved with different libraries and operating conditions. Power comparison for two types of libraries is shown. Further power for each library in the best, nominal and worst case conditions are estimated. The low V_{TH} libraries have the best power reduction when considering that the dynamic power of the unit is much smaller compared to the leakage. The leakage power also increases with the temperature i.e. for the worst case.

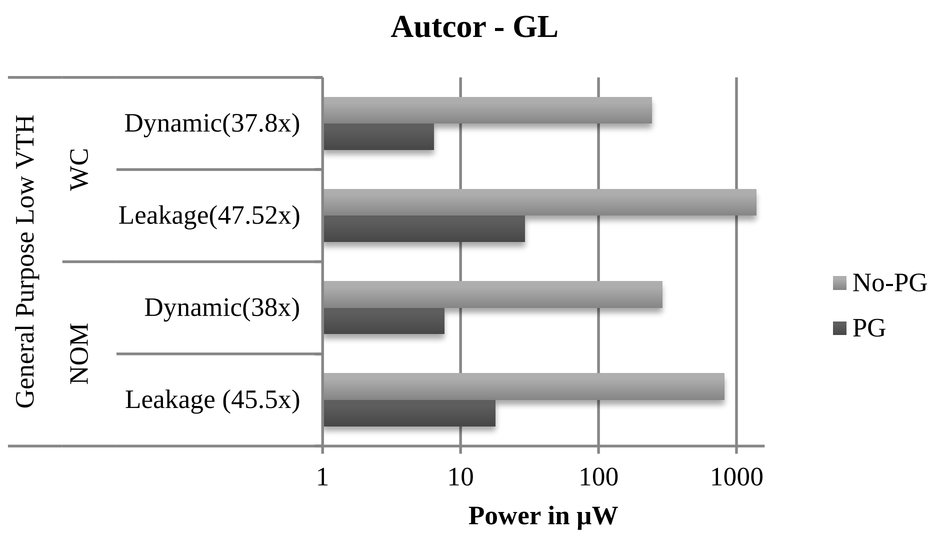


***Figure 7.1**: Power comparison for the multiplier, Benchmark: Autcor, Lib: GP-LVT*

The leakage and dynamic power estimated for the multiplier unit after RTL synthesis as compared to that of the non-power gated design for the Autcor benchmark and low V_{TH} libraries is shown in Figure 7.1. The vertical axis shows the dynamic and leakage power for nominal and worst case conditions. The numbers within the parenthesis indicate the factor by which dynamic and leakage is reduced from the original design. The horizontal axis gives the power in µW (log scale).

Figure 7.2 to Figure 7.4 shows the same results for Autcor benchmark with low V_{TH} library and FFT benchmark for both libraries.

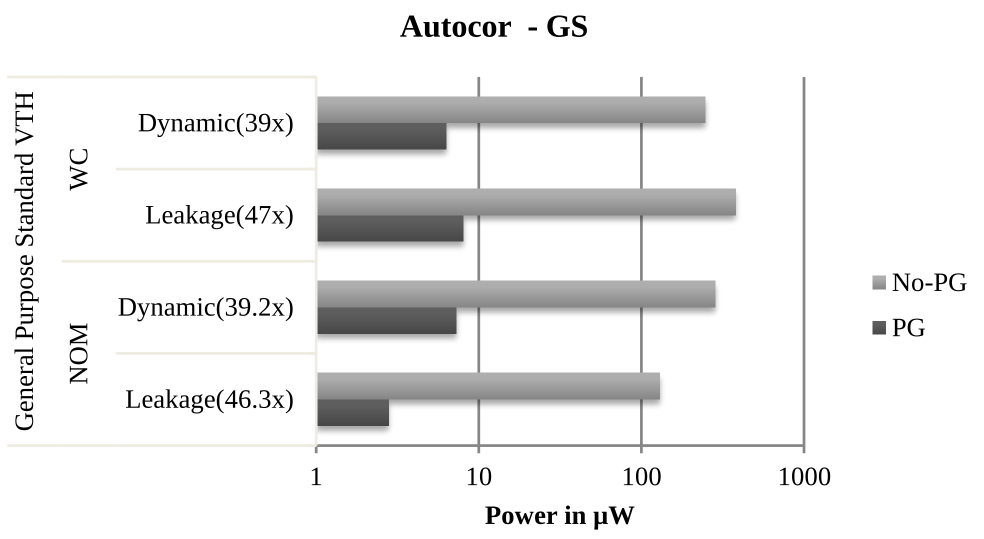


***Figure 7.2**: Power comparison for the multiplier, Benchmark:Autocor, Lib:GP-SVT*

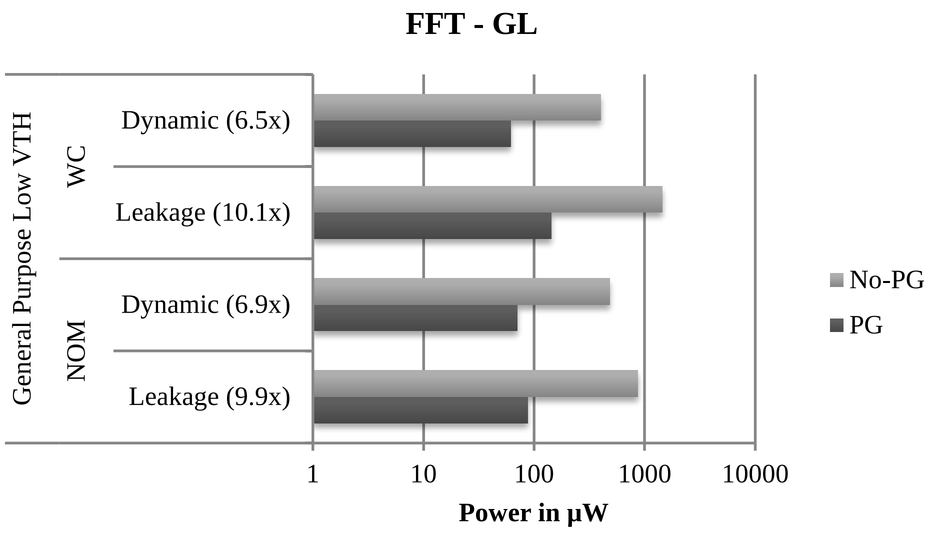


Figure 7.3: *Power comparison for the multiplier, Benchmark: FFT, Lib: GP-LVT*

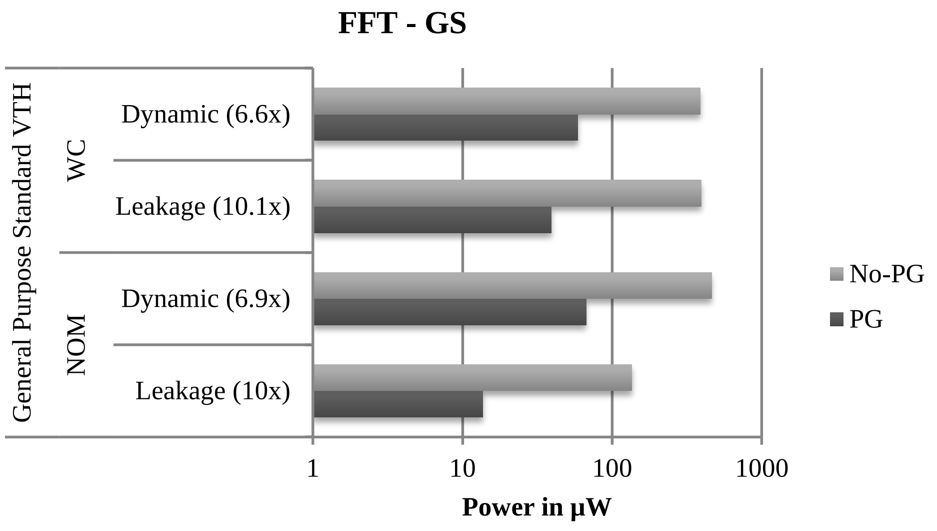


Figure 7.4: *Power comparison for the multiplier, Benchmark: FFT, Lib: GP-SVT*

It is clear from all the above cases (Figure 7.1 to Figure 7.4) that with increase in temperature the leakage will increase rapidly. This is mainly due to the dependence of temperature on the sub-threshold leakage which is evident from Equations 4 to 6. The sub-threshold leakage has a complex exponential dependence on the temperature and is the major source of leakage in the 65-nm technology. Hence the total leakage can roughly vary by the same extent as the sub-threshold leakage. For a temperature increase from 25 C (nominal) to 125 C (worst case) the total leakage approximately increases by a factor 1.7 for the low V_{TH} library and by a factor of 2.9 for the standard V_{TH} library.

Figure 7.5 summarises the overall power reduction estimation for the entire FlexCore design by power gating the multiplier. For the standard V_{TH} library overall reduction of 6% – 9% is estimated and for the low V_{TH} library overall reduction of 11% – 15% is estimated. Table 7.1 shows the overall energy and power reduction for all the benchmark, library, and operating condition combinations.

Power/Energy Reduction (RTL Synthesis)

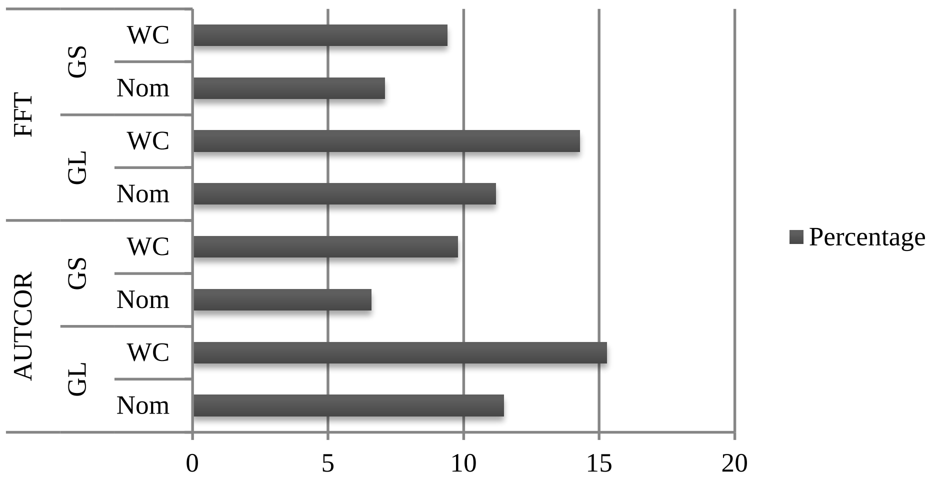


Figure 7.5*: Overall power/energy reduction from the original design*

Table 7.1*: Overall energy & power reduction after RTL synthesis, BM: Autcor & FFT*

Clock Period 3500 ps	**AUTCOR**				**FFT**			
	GL		GS		GL		GS	
	Nom	WC	Nom	WC	Nom	WC	Nom	WC
Cycle Count	19553				162967			
Energy Reduction in nJ	73.4	108.8	27.2	41.4	681.4	936.65	291.8	387.7
Power Reduction in µW	1072.85	1589.24	397.44	605	1194.68	1642.14	511.6	678.95

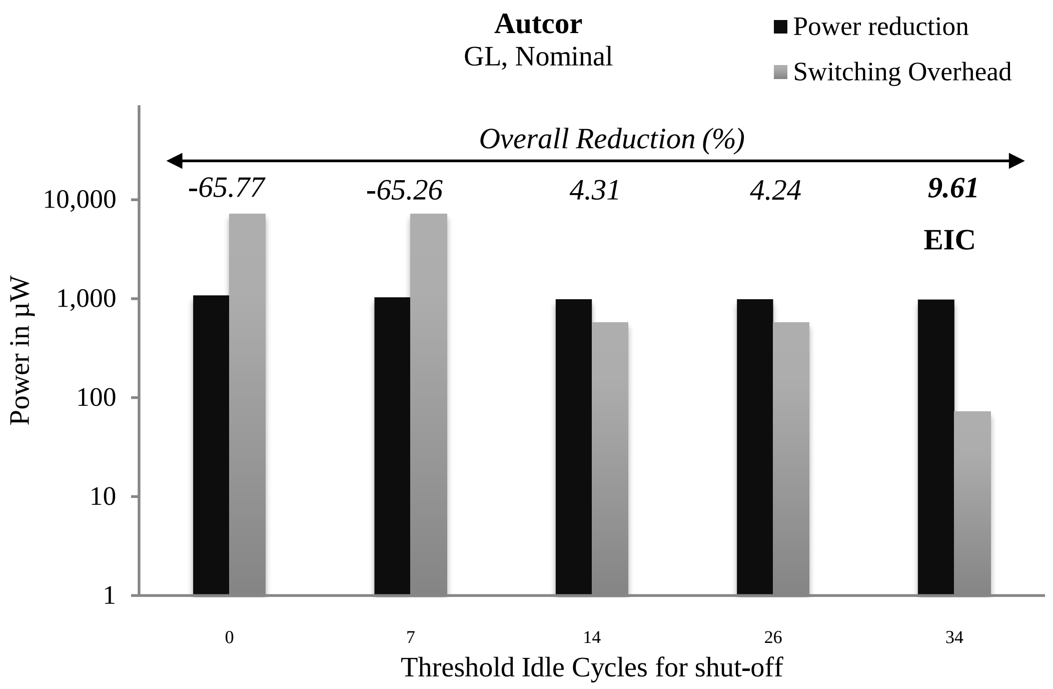
$$Energy_{reduction} = Power_{reduction} * Clock\ Period * Cycle\ Count$$
[9]

7.2 Power Reduction Estimation after Physical Implementation

The power estimations after physical estimation consider the switching overhead. An application will have different idle cycles between two consecutive usages of the multiplier. In applications having looping multiplier operations, if the multiplier was the be turned ON at every instant it is needed and turned OFF at every instant it is not needed there would be a large activity on the switches. This would lead to significant switching power and the gain achieved from power gating is lost. Thus a definite idle cycle called the ***threshold idle cycles*** must be considered to decide if the unit is to be turned OFF i.e. if the idle cycles between consecutive multiplier usage is less than the threshold cycles, then the enable signal of the multiplier unit is retained. This leads to a reduced switching overhead, however there will be a small decrease in the power savings. For the switches and benchmarks used here the change in overhead is significant compared to changes in the power savings. Input from the software analysis is used for best threshold cycles evaluation which is called the ***effective idle cycles***. The *effective idle cycles* corresponds to the best possible reduction that can be obtained. Software analysis provides the different benchmark codes (Autcor and FFT) with all the possible threshold idle cycle considerations. The power reduction and switch overhead is estimated at the physical implementation level using the Cadence SoC Encounter tool for all the cases provided by the software.

The evaluation is performed for all the library and operating condition combinations and for the Autcor and FFT benchmarks as with the estimations in Section 7.1. Figure 7.6 to Figure 7.13 shows the comparison of power reduction and switching overhead with different threshold idle cycle considerations. The power reduction implies the reduction for the entire design which is same as the power reduction from the multiplier[9]. The horizontal axis represents the different threshold idle cycles considered for the benchmark. These threshold cycles considered are obtained from the benchmark profile[10] and are not random values. The vertical axis gives the power in µW (log scale). A threshold idle cycle of 0 on the horizontal axis means than the multiplier will be either turned ON or OFF at every instant as explained in the previous paragraph. In Figure 7.6 a threshold idle cycle of 7 means that multiplier unit remains ON, unless the idle cycles is 7 or more. If the idle cycles is more than 7 then the multiplier is disabled.

For each threshold idle cycles the overall power reduction is shown at the top. A negative value indicates that the switching overhead is higher than the savings and hence the overall power will increase. Figure 7.6 shows the effective idle cycle evaluation for the Autcor benchmark with GL[11] library in a nominal operating condition. A threshold cycle of 34 gives the best power reduction of 9.61 %. Threshold idle cycle of 14 and 26 would also give a power gain, but the best reduction is obtained for 34 idle cycles and hence it is the ***effective idle cycle (EIC)*** for this case. This is indicated by the term *EIC* in the figure. Threshold idle cycle of 14 where some power gain is present is called the ***minimum threshold idle cycle***. The standby power and the leakage power of the switch cells are insignificant compared to its switching power and hence the switching overhead is relatively constant with operating condition. The switching power mainly depends on the number of transitions.



[9] No other power reduction technique is implemented and reduction achieved is only due to power gating of the multiplier hence the overall reduction will be same as the reduction for the multiplier.
[10] The threshold cycles are not multiples of any number nor do they have any relation with adjacent values.
[11] For explanation of the term GL and GS refer to the start of this section (7).

***Figure 7.6**: Power comparison of power reduction and switching overhead, Benchmark:Autcor, Lib:GP-LVT, Nominal*

As observed in Figure 7.7 the EIC remains the same for the worst case condition for the same Autcor benchmark and GL library as the previous case. However the overall reduction is higher in the worst case as compared to the nominal case which is expected as the leakage would increase with temperature (Section 7.1).

The break-even point is the number of idle cycles for which the overhead and leakage reduction are equal. For the cases in Figure 7.6 and Figure 7.7 the break-even point would lie between threshold idle cycle value of 7 and 14 where the overall reduction is 0% (overhead = leakage reduction).

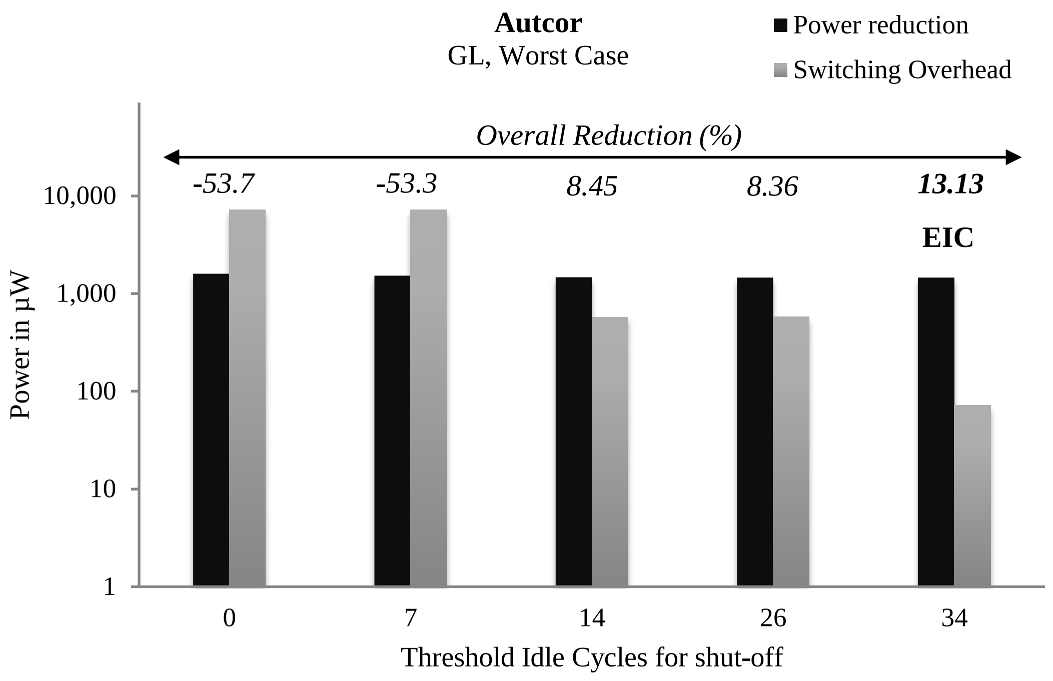


***Figure 7.7**: Power comparison of power reduction and switching overhead, Benchmark:Autcor, Lib:GP-LVT, Worst Case*

In Figure 7.8 and Figure 7.9 it can be observed that there is no power gain for threshold idle cycles of 14 and 26 compared to the previous two cases for the GL library. For the Autcor benchmark the EIC remains the same for the two libraries (GL and GS). For a different application it is possible that the EIC also changes with change in the library (especially the threshold voltage), which must be taken in to consideration on the software side.

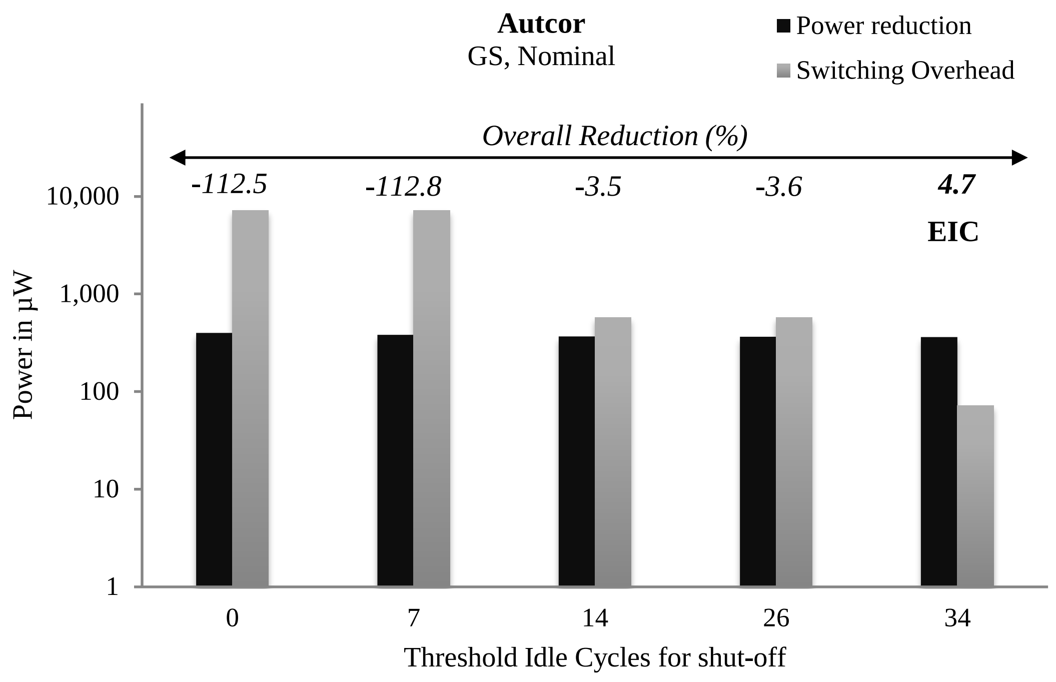


Figure 7.8: *Power comparison of power reduction and switching overhead, Benchmark:Autcor, Lib:GP-SVT, Nominal*

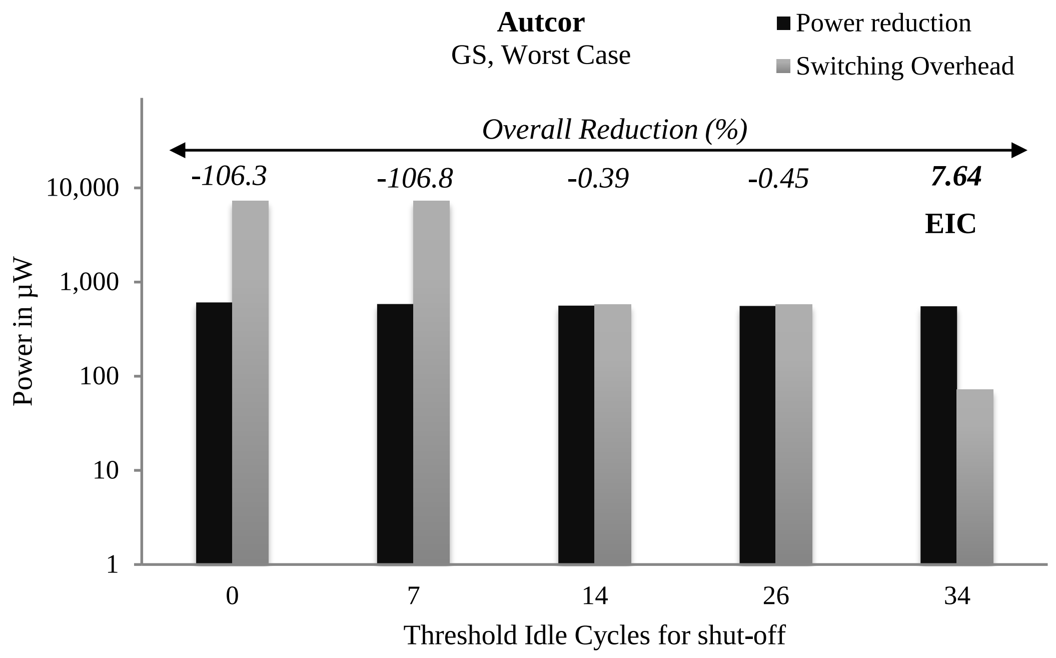


Figure 7.9: *Power comparison of power reduction and switching overhead, Benchmark:Autcor, Lib:GP-SVT, Worst Case*

The FFT benchmark is a bigger application compared to the Autcor and has more threshold idle cycles to be considered (Section 5.5). In Figure 7.10 one can observe that the EIC is 127 which is also the *minimum threshold idle cycles* for this case[12].

[12] The EIC and minimum threshold idle cycles were different for the Autcor benchmark with GL library (Figure 7.6 and Figure 7.7)

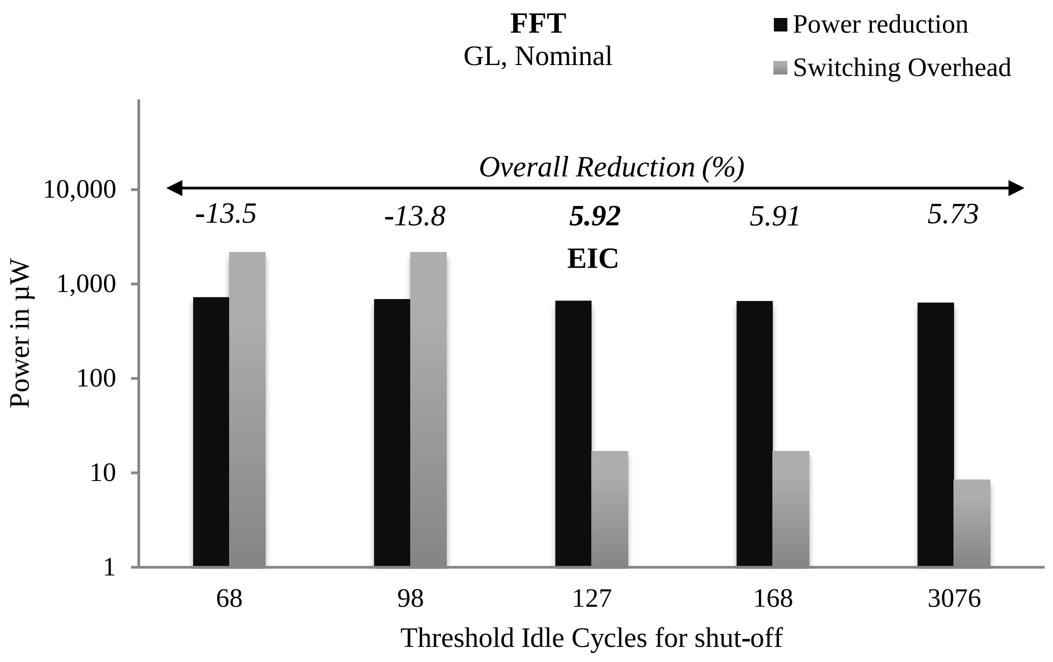


Figure 7.10*: Power comparison of power reduction and switching overhead, Benchmark:FFT, Lib:GP-LVT, Nominal*

The power gain is reduced for threshold idle cycle of 3076 (compared to 127) which means that the decrease in power reduction is larger than the decrease in power switch overhead. For threshold idle cycles lesser than 68 the overhead is very large and are not shown in all the FFT cases.

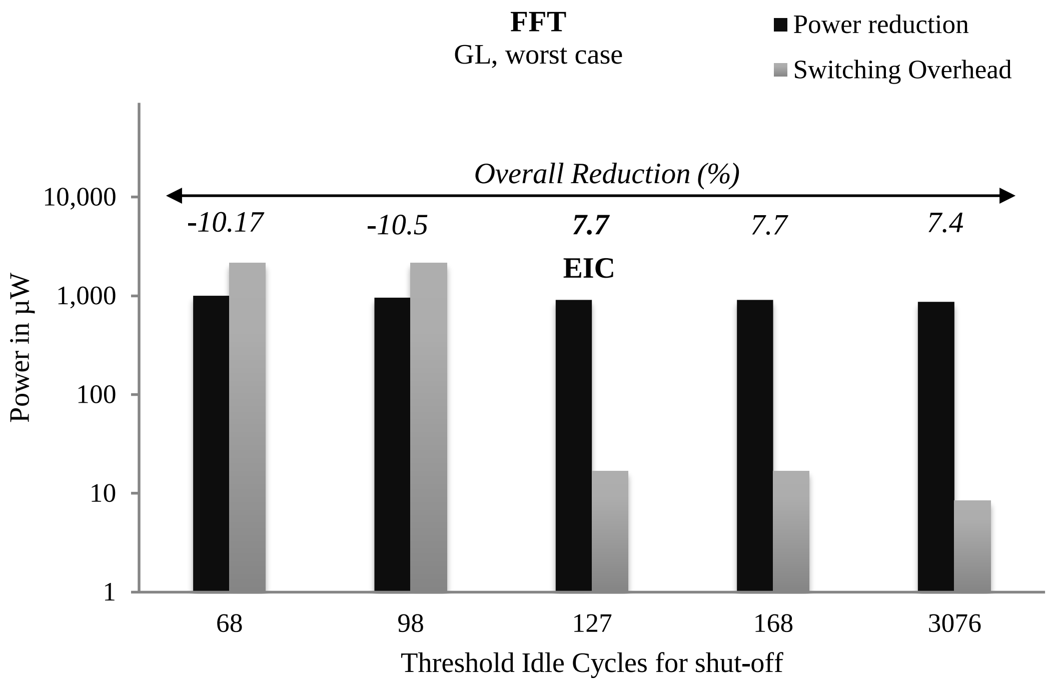


Figure 7.11*: Power comparison of power reduction and switching overhead, Benchmark:FFT, Lib:GP-LVT, Worst Case*

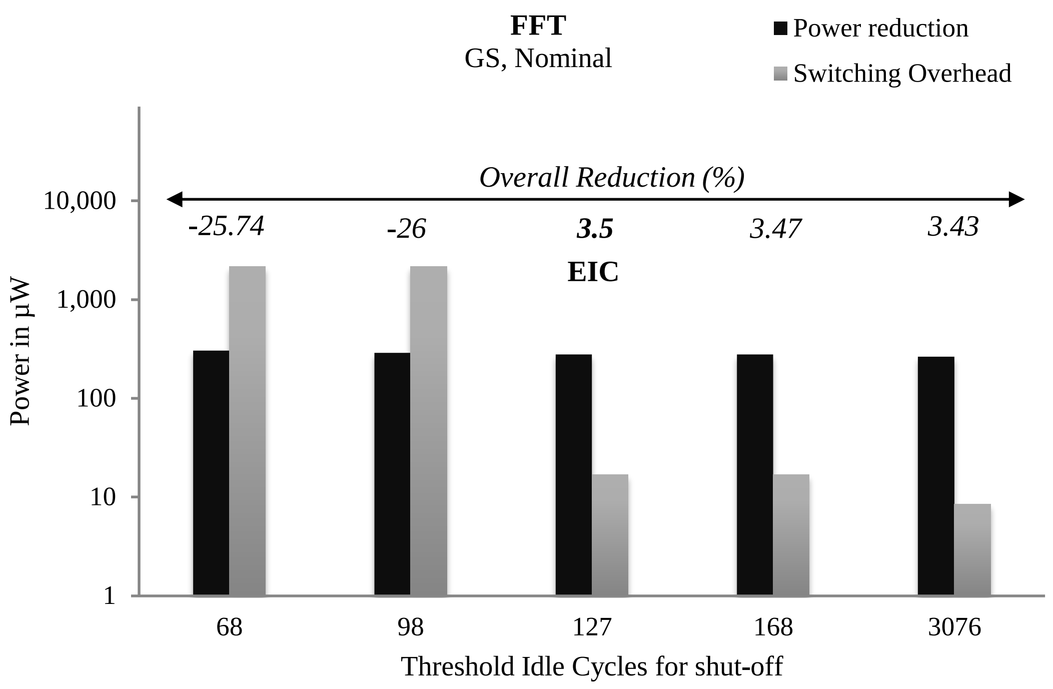


Figure 7.12*: Power comparison of power reduction and switching overhead, Benchmark:FFT, Lib:GP-SVT, Nominal*

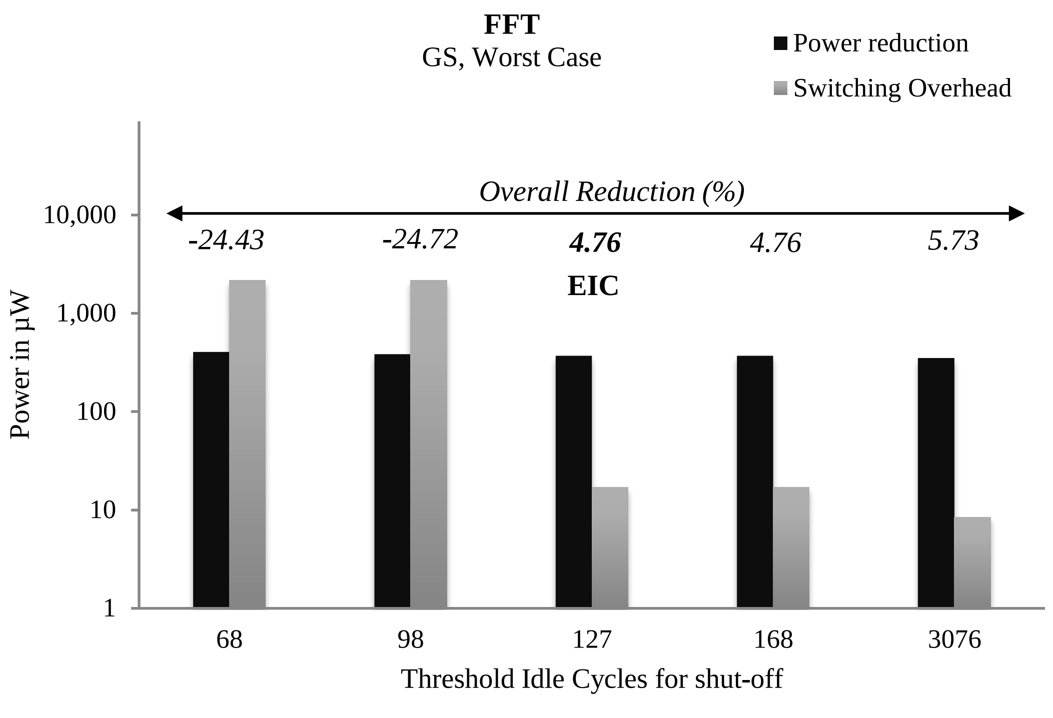


Figure 7.13*: Power comparison of power reduction and switching overhead, Benchmark:FFT, Lib:GP-SVT, Worst Case*

The effective idle cycles can also change with the operating temperature. Consider two threshold cycles ”A” and ”B” (B > A) of application. In the nominal case B is the EIC for this application (B has lesser switching overhead than A). If increase in power reduction (from nominal to worst case) at A is larger than at B and the difference of these increases is larger than the difference of switching overhead at A and B, then the overall gain in the worst case will now be at A.

$$Overhead_{diff} = A_{overhead} - B_{overhead} \quad [10]$$

$$Overhead_{diff} > A_{reduction\_nominal} - B_{reduction\_nominal} \quad \rightarrow EIC\ at\ B \quad [11]$$

$$Overhead_{diff} < A_{reduction\_worst} - B_{reduction\_worst} \quad \rightarrow EIC\ at\ A \quad [12]$$

Equation 11 denotes the initial case in the nominal operating condition. The effective gain at B is more than at A. Equation 12 denotes the case with increased temperature (worst case) in which the effective gain at A is larger than at B. More evaluation is required on this effect by considering suitable applications and will be performed as extension of this work.

Figure 7.14 summarises the overall FlexCore power/energy reduction that are estimated after the physical implementation considering the switching overhead. These savings are the actual possible savings for the design by power the multiplier. For a comparison the overall savings estimated in the previous section is also shown. The estimations for the Autcor benchmark after the RTL synthesis and physical implementation are nearly the same (difference of 1% - 2%), whereas for the FFT benchmark the two estimations differ by some extent (4% - 7%). This is also evident from the benchmark profiles discussed in Section 5.2. FFT has high multiplier usages and thus more activity on the power switches. Autcor has multiplier idle for large fraction of its time. Table 7.2 and Table 7.3 give the comparison of the actual energy and power reductions possible with the previously estimated reductions.

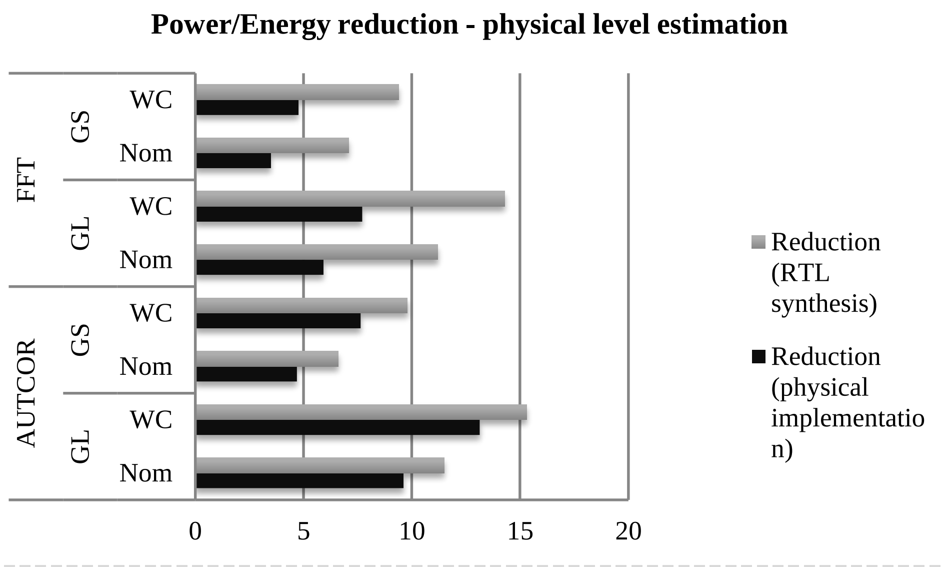


***Figure 7.14**: Power/Energy reduction for overall design after estimation at physical level (Actual)*

The results shown here are for a clock period of 3500 ps. At a lower speed the overall dynamic power would decrease which means the fraction of leakage power of the total power would increase. This would give increased power reduction. The EIC could decrease depending on the benchmark. For example, in the Autcor benchmark (GPLVT), if the dynamic power reduces by more than 5.5% (GL) - 7.2% (GS) then the

EIC will change from 34 to 27 cycles. An EIC change for the FFT benchmark would be more apparent, since it has much higher activity than Autcor.

Table 7.2: *Energy consumption of original design and reduction estimations*

Clock Period – 3500 ps	**AUTCOR**				**FFT**			
	GL		GS		GL		GS	
	Nom	WC	Nom	WC	Nom	WC	Nom	WC
Cycle Count	19553				162967			
Energy of Original Design in nJ	635.9	712.5	412.4	423	6107	6539	4115	4107
Energy Reduction (Ideal) in nJ	73.4	108.8	27.2	41.4	681.4	936.65	291.8	387.7
Energy Reduction (Actual) in nJ	61.15	93.59	19.35	32.34	361.8	503.9	143.9	195.8

Table 7.3: *Power consumption of original design and reduction estimations (Actual) after physical implementation*

Clock Period – 3500 ps	**AUTCOR**				**FFT**			
	GL		GS		GL		GS	
	Nom	WC	Nom	WC	Nom	WC	Nom	WC
Power of Original Design in µW	9292	10410	6026	6181	10707	11464	7214	7201
Power Reduction (Ideal) in µW	1073	1589	397	605	1195	1642	512	679
Power Reduction (Actual) in µW	894	1368	283	473	634	883	252	343

Table 7.4 summarises the area overhead from the power gating implementation. For a comparison the total cell area for all other units of the design except the special cells (switches, isolation) and the power control module is also shown. It is clear that the area overhead increases with the number of switches. Since the switch arrangement is of ring type, the area increase will be constant up to certain number of switches depending on the dimensions of the unit. If the switches were to be placed either on the top or bottom side of the domain a maximum of 247 switches can be placed which will have an overhead of 0.0047 mm^2. Similarly if switches are placed both on top and bottom, a maximum of 484 switches and if placed on the top, bottom and left a maximum of 600 switches can be placed. Switches more than 600 were not considered in this work, since it needs a modification of the floor plan. One can observe that with 600 switches the excess area is nearly 10% of the total cell area. The IR drop for the global supply up to 600 switches was around 0.114 V (Supply drops from 1 V to 0.886 V). Estimation of appropriate number of switches needed for this design will be performed as an extension of this work. Several factors have to be considered to estimate the number of switches as explained in Section 4.

***Table 7.4**: Area comparison of special cells and rest of the standard cells*

Area Comparison		
Switch Control Cell (VDD-CTRL)		0.0024 mm^2
PMOS Header Switches (VDD-Switch)	No. of Switches	
	1 – 247	0.0047 mm^2
	248 – 484	0.0095 mm^2
	484 – 600	0.0122 mm^2
Isolation Cells	64 Cells	232.9 µm^2
Power Control Module		56.2 µm^2
Standard Cell Area		0.05 mm^2

Maximum IR drop of global supply (TVDD) – 0.114 V (11.4%) (Up to 600 Switches)
Tap Current – 25.96 mA.

8. Conclusion & Future Work

Comparison of leakage and dynamic power in Section 7.1 upholds the facts that lead to the motivation of this work. Leakage is comparable to the dynamic power at 65 nm and this will get worse with smaller technologies. This work also substantiates the point that power gating is an effective technique to reduce leakage power dissipation. The previous section shows that with power gating a power reduction of 4-14% can be achieved depending on the application. A promising fact is that in real applications multiplier is used less intensely than the benchmarks considered here, in which case there will be a definite improvement in power reduction. Section 7.2 shows that a good evaluation of the application is required to identify the optimal reduction that can be achieved. The exposed datapath of the FlexCore provides an efficient and easier software control for power gating and also with very less hardware requirement (power control module). Since the operating status of the FlexCore in each cycle is fully exposed to the compiler, there is direct control of the power gating at each cycle and this helps in utilizing the idle cycle profile of an application with ease. The leakage power changes significantly with temperature. If software control is to be effective throughout, the temperature change must be taken into account. We can conclude that power, especially leakage power is an important concern in modern design and techniques such as power gating provided with optimal software control can reduce significant leakage.

The following tasks have to be carried out as future work. The whole step of effective idle cycle evaluation has to be completely automated. Section 7.2 suggested how the effective idle cycles can change with temperature. A detailed evaluation on this possibility has to be performed considering suitable applications. The number of switches to be used for power gating the multiplier unit has to be determined and the changes in IR drop have to be measured. The break-even point for the multiplier has to be determined for a general application.

9. Bibliography

1. ITRS, International Technology Roadmap for Semiconductors, 2007 edition. Available from: http://www.itrs.net/reports.html.

2. Chandra, G, Kapur P, Saraswat K.C. Scaling trends for the on chip power dissipation. *Interconnect Technology Conference, 2002.* no., pp. 170- 172, 2002 *Proceedings of the IEEE 2002 International,* vol.

3. FALLAH F, PEDRAM M. Standby and Active Leakage Current Control and Minimization in CMOS VLSI Circuits. *IEICE Transaction on Electronics*, Vols. E88-C, pp. 509-519, 2005.

4. Tiwari V, Monteiro J, Patel R. Power Analysis and Optimization from Circuit to Register-Transfer Levels. EDA for IC Implementation, Circuit Design, and Process Technology. Mar 2006, ISBN: 978-1-4200-0795-4.

5. Power Forward Initiative (PFI). A Practical Guide to Low-Power Design, User, Experience with CPF, May 2008. Available from: http://www.powerforward.org/

6. Altera Corp. Stratix IV FPGAs: The Lowest Power High-End 40-nm FPGA. Available from: http://www.altera.com/products/devices/stratixfpgas/stratix-iv/overview/power/stxiv-power.html#ppt

7. Thuresson M, Sjalander M, Bjork M, Svensson L, Larsson-Edefors P, Stenstrom P. FlexCore: Utilizing Exposed Datapath Control for Efficient Computing. *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pp.18-25, 16-19 July 2007

8. Hoang T.T, Jälmbrant U, Hagopian E.d, Subramaniyan K.P, Sjalander M, Larsson-Edefors P. Design Space Exploration for an Embedded Processor with Flexible Datapath Interconnect.

9. Rabaey J. Low Power Design Essentials. 1st ed. Springer; 2009.

10. Cadence Design Systems, Inc. *Common Power Format User Guide*, 2007.

11. Cadence Design Systems, Inc. *Low Power in Encounter RTL Compiler,* 2008.

12. Homayoun H, Baniasadi A. Reduction Execution Unit Leakage Power in Embedded Processors. Embedded Computer Systems: Architectures, Modelling and Simulations, 299-308, Springer Berlin /Heidelberg, July 2006.

,

Appendix – A: Software Analysis Makefile

```
#-------------------------------------#
#- Makefile of the Software Analysis -#
#-------------------------------------#

#### Definitions ####

# Directories

WORK         = Work directory
BENCHMARK     = Benchmark directory
TELECOM       = EEMBC, Telecom suite directory
MIPS          = MIPS-assembly directory
FLEX         = RTN-format for FlexCore directory
SHARED        = Shared benchmark files directory
RESULTS       = Results directory
BM_C_CODE     = Benchmark C-code
C_CODE        = C-code of power gating directory

# Input files
BM             = Selected benchmark

# Options
MIPS_OPT      = -mno-explicit-relocs -c -O2 -D NDEBUG -mno-abicalls
FLEX_OPT      = --interconnect=../interconnect.csv \
                --flex=../flex.ini

RUNMIPS_OPT   = -interconnect $(WORK)/interconnect.csv +RTS \
                 -k500000000 -RTS -m 10M -flex $(WORK)/flex.ini
ITERATIONS    = 1

#### Software Analysis of the EEMBC benchmark ####

# Compiling C-code to MIPS-assembly code
cmips:

      @ # diffmeasure.c
      @mips-elf-gcc-4.1.1 $(MIPS_OPT) -I$(TELECOM)/th_lite/mips/al\
      -I $(TELECOM)/th_lite/src -I diffmeasure -I. \
      -I $(TELECOM)/th_lite/mips/al -I$(TELECOM)/th_lite/src -S-o \
      $(TELECOM)/telecom/mgcc/asm_lite/diffmeasure/diffmeasure.S \
      $(TELECOM)/telecom/diffmeasure/diffmeasure.c

      @ # verify.c
      @mips-elf-gcc-4.1.1 $(MIPS_OPT) -I$(TELECOM)/th_lite/mips/al\
      -I $(TELECOM)/th_lite/src -I diffmeasure -I. \
      -I $(TELECOM)/th_lite/mips/al -I $(TELECOM)/th_lite/src -S-o\
      $(TELECOM)/telecom/mgcc/asm_lite/diffmeasure/verify.S \
      $(TELECOM)/telecom/diffmeasure/verify.c

      @ # $(BM).c
      @mips-elf-gcc-4.1.1 $(MIPS_OPT) -I. \
            -I $(TELECOM)/th_lite/mips/al -I $(TELECOM)/th_lite/src\
            -I $(BM)00 -I $(TELECOM)/telecom/$(BM)00/datasets \
            -I $(TELECOM)/telecom/diffmeasure \
            -DDATA_1 -DITERATIONS=DEFAULT \
            -I.-I$(TELECOM)/th_lite/mips/al-I$(TELECOM)/th_lite/src\
            -S -o $(MIPS)/$(BM)_mipsed.S $(BM_C_CODE)/$(BM).c
```

```
	@ # bmark_lite.c
	@mips-elf-gcc-4.1.1 $(MIPS_OPT) -I. \
		-I $(TELECOM)/th_lite/mips/al -I $(TELECOM)/th_lite/src\
		-I $(bm)00 -I $(TELECOM)/telecom/$(BM)00/datasets \
		-I $(TELECOM)/telecom/diffmeasure \
		-DDATA_1 -DITERATIONS=DEFAULT \
		-I.-I$(TELECOM)/th_lite/mips/al-I$(TELECOM)/th_lite/src\
		-S -o $(MIPS)/bmark_lite.S $(BM_C_CODE)/bmark_lite.c


# Compiling MIPS-assembly code to RTN-format for the FlexCore
mipsflex:

	@flexcomp $(FLEX) $(BENCHMARK)/*.S
	@mv $(FLEX)/*.S $(SHARED)

	@flexcomp $(FLEX) $(MIPS)/*.S
	@mv $(FLEX)/$(BM)_mipsed.S $(FLEX)/$(BM)_flexed.S


# Simulating the FlexCore and generating analysis related
information (files)
flextest:

	@runmips  $(RUNMIPS_OPT) -arg -i$(ITERATIONS) -q  \
		-proff           $(RESULTS)/profile.$(bm)  \
		-tracenisad      $(RESULTS)/readable.$(bm) \
		-showcode     >  $(RESULTS)/showcode.$(bm) \
		-trinarytnisaf   $(RESULTS)/trinary.$(bm)  \
		-tnisaf          $(RESULTS)/hex.$(bm)      \
		 $(SHARED)/*.S $(FLEX)/$(BM)_flexed/*.S


# Running all the commands at once
test: cmips mipsflex flextest

	@gcc $(C_CODE)/powergate.c -o $(C_CODE)/powergate.out
	@$(C_CODE)/powergate.out

	@gedit $(RESULTS)/showcode.$(BM)
	@gedit $(RESULTS)/profile.$(BM)
	@gedit $(RESULTS)/hex.$(BM).code
	@gedit $(RESULTS)/trinary.$(BM).tcode
	@gedit $(RESULTS)/readable.$(BM)

# END #
```

Appendix – B: FlexCore CPF

```
##################################################
#           Technology part of the CPF           #
##################################################

set_hierarchy_separator /
set_power_unit uW

#     Specify libraries    #

define_library_set -name lib1d2v -libraries { \
      $loc_switch /ESWITCH65LPSVTHVT_1V2_50A_nom_1.20V_25C.lib \
      $loc_iso/CORI65LPSVT_nom_1./20V_25C.lib \
      $loc_core/CORE65GPLVT_nom_1.00V_25C.lib \
      $loc_corx/CORXT65GPLVT_nom_1.00V_25C.lib \
      $loc_clck/CLOCK65GPLVT_nom_1.00V_25C.lib \
      $loc_prhs/PRHS65_nom_1.20V_25C.lib  }

#     Specify special cells     #

##    Isolation Cells  ##

define_isolation_cell -cells {"HS65_LS_ISOAND*"} -enable PE \
      -valid_location to
define_isolation_cell -cells {"HS65_LS_ISOOR*"} -enable PD \
      -valid_location to

##    Power Switch Cells    ##

define_power_switch_cell -cells \
      "SW65_LH_VDDSWITCH SW65_LH_VDDCTRL" \
      -power TVDD \
      -power_switchable VDD \
      -type header -stage_1_enable MACROEN

##    Retention Cells  ##

      #define_state_retention_cell -cells {""} -restore_function

##    Always ON Cells  ##

      #define_always_on_cell -cells ""


##################################################
#             Design part of the CPF             #
##################################################

set_design FlexCore_top

#     Declare power/ground nets      #

      create_power_nets -nets TVDD -voltage 1.0
      create_power_nets -nets VDD
      create_ground_nets -nets VSS -voltage 0
      create_power_nets -nets SWVDD
```

```
#    Specify power domains    #

create_power_domain -name PD_default -default

create_power_domain -name PD_mult \
      -instances {core/MULTunit/MULTunit} \
      -shutoff_condition {!core/mult_en}

#    Nominal operating conditions   #

create_nominal_condition -name off -voltage 0

create_nominal_condition -name on -voltage 1.0


#    Modes of operation   #

create_power_mode -name PM1 -domain_conditions \
      {PD_default@on PD_mult@on} -default

create_power_mode -name PM2 -domain_conditions \
      {PD_default@on PD_mult@off}


#    Design rules   #

##    Isolation rule   ##

create_isolation_rule -name iso1 -from PD_mult \
      -isolation_condition {core/ise_m} -isolation_output low

##    Power switch rule     ##

create_power_switch_rule -name psr1 -domain PD_mult \
       -external_power_net TVDD

##    State retention rule  ##

#create_state_retention_rule -name st1 -domain PD_mult \
      -restore_edge {!pm_inst.pge_enable[0]} \

##################################################
#             Update libraries                   #
##################################################

# Associate library sets with nominal conditions #

update_nominal_condition -name on -library_set lib1d2v

# Update isolation the rules #

update_isolation_rules -names iso1 -location to \
      -cells {"HS65_LS_ISOAND*"}

# Update powerswitch rules   #

update_power_switch_rule -name psr1 -prefix CPF_PS_ \
      -cells "SW65_LH_VDDSWITCH"
```

```
# Specify timing constraints  #
update_power_mode -name PM1 -sdc_files pm1.sdc

update_power_mode -name PM1 \
     -activity_file Flex_top.tcf -activity_file_weight 100

# Describing power nets  #

create_global_connection -domain PD_default -net TVDD -pins "VDD
vddo"
create_global_connection -domain PD_default -net VSS -pins "VSS
gndo"
create_global_connection -domain PD_default -net VSS -pins "gndi
swgnd"
create_global_connection -domain PD_default -net SWVDD -pins swvdd
create_global_connection -domain PD_mult -net VDD -pins VDD
create_global_connection -domain PD_mult -net VSS -pins VSS


# update Power Domain #

     update_power_domain -name PD_default -internal_power_net TVDD
     update_power_domain -name PD_mult -internal_power_net VDD



end_design

# END #
```

Appendix – C: Power Control Module

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;


entity PowerControl is
    port (
      Clk      : in std_logic;
      Reset    : in std_logic;
      pc_in    : in std_logic_vector(110 downto 109);
             --NISA 111th and 110th bits
      mult_en : in std_logic;
      --NISA 67 , multiplier enable bit
      pse_m   : out std_logic);
      --multiplier switch and isolation control
end;

architecture rtl of PowerControl is
    type state_type is (S0, S00, S01, S10, S11);
    signal state_m,nstate_m: state_type;
    signal m:std_logic;
    signal c_delay_cycles_S00,n_delay_cycles_S00: natural range 0
to 3; --delay counter for mode 0
    signal c_delay_cycles_S11,n_delay_cycles_S11: natural range 0
to 3; --delay counter for mode 11
    signal c_idle_cycles,n_idle_cycles:natural;
    constant threshold_cycles: natural:=100; --threshold cycles for
mode 10
begin

seq:   process
(Clk,Reset,nstate_m,n_delay_cycles_S00,n_delay_cycles_S11,n_idle_cy
cles)
       begin
            if Reset = '0' then
            state_m<=S0;
            elsif falling_edge(Clk) then
            state_m<=nstate_m;
            c_delay_cycles_S00<=n_delay_cycles_S00;
            c_delay_cycles_S11<=n_delay_cycles_S11;
            c_idle_cycles<=n_idle_cycles;
            end if;
      end process seq;

comb_m:     process
(pc_in,state_m,c_delay_cycles_S00,c_delay_cycles_S11,c_idle_cycles)
      begin
               case state_m is

                  when S0     =>          --Starting State
                  n_delay_cycles_S00<=0;
                  n_delay_cycles_S11<=0;
                  n_idle_cycles<=0;
                  m<='1';
                  if pc_in= "00" then
                  nstate_m<=S00;
                  elsif pc_in= "01" then
                  nstate_m<=S01;
                  elsif pc_in= "10" then
                  nstate_m<=S10;
                  elsif pc_in= "11" then
                  nstate_m<=S11;
                  else
                  nstate_m<=S0;
                  end if;


                  when S00    =>    --Mode 0 - Multiplier OFF
                  if pc_in = "01" then
                  n_delay_cycles_S00 <= 0;
                  m<='1';
                  nstate_m<=S01;
                  elsif pc_in= "00" then
                        if c_delay_cycles_S00 < 3 then

                        m<='1';
                        n_delay_cycles_S00 <= c_delay_cycles_S00 +
1;
                        else
                        m<='0';
                        end if;
                  else
                  m<='1';
                  nstate_m<=S0;
                  end if;

                  when S01    =>    --Mode 0 - Multiplier ON
                  m<='1';
                  if pc_in = "00" then
                  n_delay_cycles_S00 <= 1;
                  nstate_m<=S00;
                  elsif pc_in= "01" then
                  nstate_m<=S01;
                  else
                  nstate_m<=S0;
                  end if;

                  when S10    =>    --Mode 10 Threshold Cycles
                  if pc_in = "10" then
                        if mult_en='1' then
                        m<='1';
                        n_idle_cycles<=0;
                        elsif mult_en<='0' then
                              if c_idle_cycles < threshold_cycles
then
                              m<='1';
                              n_idle_cycles<=c_idle_cycles+1;
                              else
                              m<='0';
                              end if;
                        else
                        m<='1';
                        nstate_m<=S0;
                        end if;
                  else
                  m<='1';
```

```
                 nstate_m<=S0;
                 end if;

                 when S11    =>          --Mode 11 (Testing)
                 if pc_in = "11" then
                       if mult_en='1' then
                       m<='1';
                       n_delay_cycles_S11<=0;
                       elsif mult_en='0' then
                             if c_delay_cycles_S11 < 3 then
                             n_delay_cycles_S11 <=
c_delay_cycles_S11 + 1;
                             m<='1';
                             else
                             m<='0';
                             end if;
                       else
                       nstate_m<=S0;
                       m<='1';
                       end if;
                 else
                 nstate_m<=S0;
                 m<='1';
                 end if;

                 when others =>
                       m<='1';
                       nstate_m<=nstate_m;
                 end case;
      end process comb_m;

      pse_m<=m;

end rtl;
```