Power Modelling and Validation

Gheorghe Sârбу
Hughes Hall

Option B Project Report
A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for the degree of
Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: Gheorghe.Sarbu@cl.cam.ac.uk

October 31, 2013
Declaration

I, Gheorghe Sărbu of Hughes Hall, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,585

Signed:

Date: October 31, 2013

This dissertation is copyright ©2013 Gheorghe Sărbu.

All trademarks used in this dissertation are hereby acknowledged.
Acknowledgements

My sincerest thanks to my supervisor, Dr Robert Mullins, and to Daniel Bates, for the central part they have played in the success of my project.

I am grateful to my MPhil colleagues for the trust they placed in me, as their elected representative, and for their support throughout the year.

This step of my life - the MPhil in Advanced Computer Science at the University of Cambridge - would not have been possible without generous financial support of numerous people and institutions, to whom I will be forever indebted.

Last, but not least, I thank the staff of the Graduate Administration Office, whose unwavering attention to details, openness to dialogue and willingness to help in the direst of situations have earned them the utmost respect on my part.
Abstract

Recent years have seen a rise in the number of solutions proposed for chips with high core counts (over 32 cores on a single chip) - the era of manycores is drawing near. The huge design space that architects now investigate suffers from severe limitations because of the utilization wall, so no significant breakthrough can be made without taking into account power constraints.

My work aims to help design space exploration for manycores by providing insight into details of power dissipation in current day chips. I achieve this through power models built for simple units inside a RISC pipeline - typical core in a future manycore called Loki.

This paper presents linear models correlating switching activity to energy consumption and evaluates their cumulated ability to predict energy estimation for the entire pipeline. Evaluations are run for an implementation using a 45nm standard cells library, and find that my modelling methodology is sensible - models are able to provide energy predictions correct to a first order, with an average error of 35%. I discuss possible reasons for this error, in the end.
# Contents

1 Introduction ................................................. 1

2 Background .................................................. 5
   2.1 Feature size ........................................... 5
   2.2 Circuit characterization ................................. 7
      2.2.1 Delays ........................................... 7
      2.2.2 Power dissipation ................................ 9
      2.2.3 Wires ............................................. 13
   2.3 RISC pipelines ........................................... 14
      2.3.1 Computer performance .............................. 14
      2.3.2 Overview of pipelining ............................. 16
      2.3.3 Pipelining a RISC architecture .................... 17

3 Related Work .................................................. 21
   3.1 RTL power modelling .................................... 22
   3.2 Architectural level power modelling .................... 23

4 Design and Implementation ................................. 27
   4.1 Pipeline Design ......................................... 27
      4.1.1 The big picture ................................... 27
      4.1.2 Overview of design ................................ 28
      4.1.3 Pipeline characteristics ........................... 31
      4.1.4 Behavioural caches ................................ 35
   4.2 Pipeline Implementation ................................. 36
      4.2.1 Hardware description ............................... 36
      4.2.2 Synthesis ......................................... 37
      4.2.3 Place and route ................................... 39
   4.3 Preparing evaluation .................................... 40
      4.3.1 Overview of approach ............................... 41
      4.3.2 Toolflow description ............................... 42
5 Evaluation

5.1 Pipeline verification ............................................. 47
  5.1.1 Submodule verification ....................................... 48
  5.1.2 Pipeline testing ............................................. 49

5.2 Unit models ....................................................... 50
  5.2.1 Fifo buffer .................................................... 52
  5.2.2 ALU ........................................................... 57
  5.2.3 Multiplier ...................................................... 59
  5.2.4 Forwarding logic .............................................. 60
  5.2.5 Register file ................................................. 62

5.3 Pipeline stage models ............................................ 64
  5.3.1 Fetch ......................................................... 64
  5.3.2 Decode ....................................................... 66
  5.3.3 Register Read ................................................ 67
  5.3.4 Execute ....................................................... 69
  5.3.5 Write Back ................................................... 69

5.4 Model accuracy evaluation ........................................ 71

5.5 Summary .......................................................... 72

6 Summary and Conclusions ........................................... 75
List of Figures

2.1 Ideal CMOS inverter [1] ........................................... 6
2.2 Transistor feature size evolution between 1970 and 2011 [2] . . 6
2.3 Multiple-clock-cyle pipeline diagram of five instructions [3] . . 18

4.1 Block-diagram of pipeline ........................................ 30
4.2 Placed and Routed pipeline ................................. 40
4.3 Project workflow ............................................ 45

5.1 Average energy vs. FIFO buffer width. A linear trend is evident. 54
5.2 Sample file with energy values and captured attributes, ready
for input into $R$ .................................................. 55
5.3 Distribution of average-centred errors for Qsort .................. 73
5.4 Distribution of average-centred errors for CRC ................. 73
5.5 Distribution of average-centred errors for Bitcnts ............... 73
5.6 Distribution of average-centred errors for String Search ...... 74
5.7 Distribution of average-centred errors for Basicmath .......... 74
5.8 Distribution of average-centred errors for Dijkstra ............ 74
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Opcodes for implemented Instruction Set Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>Process-voltage-temperature (PVT) corners for standard cells library</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>Power numbers for FIFO buffers</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Models for FIFO buffers</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Power numbers for ALU</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>Model for ALU</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Power numbers for Multiplier</td>
<td>59</td>
</tr>
<tr>
<td>5.6</td>
<td>Model for Multiplier</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>Power numbers for Forward</td>
<td>61</td>
</tr>
<tr>
<td>5.8</td>
<td>Model for Forward</td>
<td>61</td>
</tr>
<tr>
<td>5.9</td>
<td>Power numbers for Register File</td>
<td>63</td>
</tr>
<tr>
<td>5.10</td>
<td>Model for Register File</td>
<td>63</td>
</tr>
<tr>
<td>5.11</td>
<td>Power numbers for Fetch stage</td>
<td>65</td>
</tr>
<tr>
<td>5.12</td>
<td>Model for Fetch stage</td>
<td>65</td>
</tr>
<tr>
<td>5.13</td>
<td>Power numbers for Decode stage</td>
<td>66</td>
</tr>
<tr>
<td>5.14</td>
<td>Model for Decode stage</td>
<td>67</td>
</tr>
<tr>
<td>5.15</td>
<td>Power numbers for Register Read stage</td>
<td>68</td>
</tr>
<tr>
<td>5.16</td>
<td>Model for Register Read stage</td>
<td>69</td>
</tr>
<tr>
<td>5.17</td>
<td>Power numbers for Write Back stage</td>
<td>70</td>
</tr>
<tr>
<td>5.18</td>
<td>Model for Write Back stage</td>
<td>70</td>
</tr>
<tr>
<td>5.19</td>
<td>Average normalized errors of energy estimation</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Traditionally, the mission of a computer architect was to develop faster processors, taking advantage of more and more transistors, with higher and higher switching speeds and favourable voltage scaling. When scaling voltage stopped being an option (due to leakage currents) and increasing clock frequency had to be abandoned (due to unmanageable increases in dynamic power), architects were left with only the high transistor count to help them achieve design goals.

In search of new design ideas that could bring the much needed extra performance for each generation of devices, designers had to pay closer and closer attention to implications their decisions had on power. As these decisions became ever more expensive to make, chip designers and manufacturers announced a change of tactics - instead of a single power-hungry core, they would place several smaller cores on a chip. It was the beginning of the multicore era - an era that lasts to present day. This is, however, only a stepping stone to yet another level - the manycores (chips with more than 32 cores).

The prospect is very plausible, given the fact that CMOS technology continues to scale down, and transistor counts increase dramatically with
each process generation. Yet, as previously specified, the power budget does not go up, to allow designers the freedom to use these extra transistors at full potential. In fact, there are those that claim the end of multicore scaling is near [4], and that we will soon hit a point, dubbed the utilization wall, past which we will not be able, for a fixed power budget, to fire up the entire chip at once. Worst-case predictions are that, in the near future, at most one core running at full power, with several others assisting it, will be the usual multicore utilization profile.

Once we reach that point, we will need to think of new computing paradigms. Adopting any major change will require significant knowledge about finer details of power dissipation, which we can gather only through significant investigation. This is where my project fits in - it aims to provide some much needed insight into how power dissipates in usual logic circuits - those we would find inside cores of typical multicores and manycores.

The key idea behind my project is to model power dissipation in sub-modules of a RISC pipeline. Such small cores have the highest chance of ending up in the manycores of the future, so modelling units inside them is well-justified. Furthermore, to be able to run any meaningful tests, I had to first build a test infrastructure. This came in the form of a pipeline, implementing a given Instruction Set Architecture (ISA). This pipeline, in itself, is an important product of my project, because it will save future researchers the need to take it from zero.

The same applies for models, which are the result of lengthy simulations. Any model I am able to extract might serve as starting point for cycle accurate models, meant to be integrated in a high-level cycle-accurate simulator.

So, my goal was to provide models describing energy per cycle. To derive such models I had to (i) build the pipeline; (ii) find correlations between switching activity and energy consumed in a clock cycle for each submodule; (iii) assess to what degree models found in step (ii) can predict energy
consumption of the entire pipeline. Step (i) is a bit more laborious, as it
starts with the pipeline design and, after several stages, it ends with a fully
placed and routed implementation, making use of standard cells from a 45nm
process.

Models I obtain are linear combinations of variables I collect during sim-
ulations, and are of the form:

\[ \text{energy} = coef_0 \cdot var_0 + coef_1 \cdot var_1 + \ldots, \varepsilon = \ldots \]  

(1.1)

where \( \varepsilon \) is the \textit{residual standard error} of the model.

I construct a set of such models and evaluate their performance against
power numbers taken from running a benchmark of six programs through
the pipeline. Although models I find are not accurate enough to be used in
a simulator (average error of 35\%), they prove to be sensible enough to offer
estimations correct to the first order.

Before I can go into details of my work, I must set the stage by providing
some background information about CMOS technology, RISC pipelines and
about most relevant results researchers have obtained in the field of power
modelling. Chapters 2 and 3 will cover these topics, while chapters 4 and 5
provide a detailed description of my work - methodology, tools and results.
Chapter 2

Background

This chapter gives a broad overview of the background information relevant to the scope of this dissertation. I start at the CMOS circuit level, by describing metrics for characterization - feature size, delay, and power dissipation. I then move on to the architectural level, and discuss key ideas pertaining to RISC pipelines design.

2.1 Feature size

In 1963, Frank Wanlass at Fairchild presented the first ever implementation of a logic gate using Metal-Oxide-Semiconductor Field-Effect Transistors (MOSFETs). Employing two such devices - one n-type, one p-type - the gate was dubbed Complementary MOS (CMOS). Nowadays, half of century later, this name is ubiquitous in the world of digital design. Over the years, as transistors became smaller and smaller, engineers were able to construct more complex circuits, by packing more and more CMOS devices on the same die area. In addition, the reduction in size meant lower distances between transistor source and drain, and, consequently, higher switching speeds and
lower voltage thresholds. It was this trio of factors that fuelled what we now know as the CMOS digital revolution - a period of tremendous growth, unmatched by any other area of human knowledge.

The need to report what technology generation was in use, at a given point in time, lead to the adoption of a very popular metric, still in use today - the feature size. It is the minimum distance between transistor source and drain, in a given process. The evolution of values for this metric shows the high rate of advancement in the area.

Between 1970 and 2011 [Figure 2.2], the minimum feature size shrank from 10 $\mu m$ to 28 $nm$ (yellow circles, right $y$ axis). During the same period, the number of transistors per square millimetre increased from 200 to more than 1 million (diamonds, triangles and squares, left $y$ axis).
2.2 Circuit characterization

Introducing a method to distinguish between technology generations is only the starting point of circuit characterization. To be able to understand results I obtain in simulation, one must also look into the theoretical fundamentals on which tools I describe in chapter Design and Implementation operate. Namely, I will briefly discuss key aspects related to delay estimation and power dissipation.

2.2.1 Delays

Estimating delays is of the utmost importance for every digital designer. This is because each design will have a number of paths that require attention to timing details, called critical paths. These critical paths can be modified at four main levels:

- The architectural/microarchitectural level - tradeoffs include number of pipeline stages, execution units, size of memories.
- The logic level - tradeoffs include types of functional blocks, number of stages of gates in a cycle.
- The circuit level - transistor size, styles of CMOS logic.
- The layout level - share diffusion nodes between transistors to reduce diffusion capacitance.
RC delay model

The simplest method for estimating delays starts with the approximation of a transistor as a switch in series with a resistor \( R \), and in parallel with a capacitance \( C \). It is difficult to estimate \( R \) values in modern processes, but it is clear that: (i) \( R \) is inversely proportional to the width-to-length ratio \((W/L)\) of a transistor; (ii) \( R \) is inversely proportional to the gate-source voltage drop \((V_{gs})\); (iii) \( R \) for pMOS transistors is 2-3 times higher than that for nMOS transistors. The capacitance \( C \) stems from at least two components: gate capacitance and diffusion capacitances. By employing this approximation, the delay of a CMOS inverter driving four instances of itself can be estimated using [5]:

\[
t_{pd} = 4 \cdot 3RC + 3RC = 5\tau
\]  

(2.1)

where \( \tau = 3RC \) is termed inverter time constant. This metric has become widely used, and it is known as fan-out-of-four (FO4) delay. The FO4 delay is usually quoted assuming typical process parameters and worst-case environment.

Linear Delay Model

The Linear Delay Model [6] (LDM) proposes a more general approach to propagation delay estimation for a given logical gate:

\[
d = gh + p
\]  

(2.2)

where \( p \) is the parasitic delay inherent to the gate when no load is attached, and \( f = gh \) is the stage effort, that depends on the complexity (input capacitances) and fan-out (load capacitances) of the gate.

I present this model here because CAD tools use generalized, properly

The above models convey a clear message - in order to determine delays, one needs to extract \( R \) and \( C \) values from the implementation. However, in order for them to be accurate, the tool that does this job needs to have access to the actual circuit layout.

First step is to determine transistor effective resistances. We do this at the circuit level. Since we build most transistors using minimum length, resistance is determined by transistor width, and it is a multiple of unit \( R \) (where unit \( R \) is the effective resistance of the nMOS inside the smallest inverter in the standard cells library).

Second step is to determine capacitances. We could determine gate capacitances using only transistor width values, because they scale linearly with width (so they can be expressed in terms of unit \( C \)). However, diffusion capacitances are dependent on layout (e.g. in a good layout, diffusion nodes are shared, to reduce diffusion capacitance).

For the purpose of my project, I extract accurate \( R \) and \( C \) information using a tool called *Synopsys StarRC*.

### 2.2.2 Power dissipation

A one-time secondary concern, power dissipation is now a primary design constraint, due to ever-increasing transistor counts and switching speeds, all coupled with the advent of low-power hand-held devices. As a result, designers nowadays must constantly strive to ensure intelligent tradeoffs between area, speed and power. Having tackled the first two issues, I will address the latter in this section. Power dissipation in CMOS circuits has two components:
• Static dissipation

• Dynamic dissipation

Static power

When a CMOS inverter is in either of states 0 or 1, one of the two transistors is OFF. Ideally, there is no current flow, so power dissipation is zero when the circuit is quiescent. However, in reality, various phenomena lead to small amounts of static current flowing through the OFF transistor. These phenomena are:

• Subthreshold conduction - The ideal assumption is that current flows from source to drain only when the voltage drop between gate and source \( V_{gs} \) is higher than a given threshold \( V_t \). In reality, current does not cut off instantly below threshold, but rather drops off exponentially [7]. This is also called leakage. By decreasing threshold voltage \( V_t \) or by increasing temperature, subthreshold current increases exponentially. So, for the very-low threshold transistors used nowadays, subthreshold current has to be taken into account as a source of power dissipation.

• Tunnelling - The ideal assumption is that the gate is a perfect insulator. This is, to some extent, accurate. However, as the layer of \( SiO_2 \) gets thinner with each process generation, the probability that electrons will tunnel across the insulator increases. This phenomenon has become important starting with the 130 nm feature size, where the gate oxide was about 20 Å wide [6].

• Junction leakage - The ideal assumption is that reversed-biased diodes, created at p-n junctions inside or between transistors, do not conduct current. However, a small amount of current does flow. This source of
static power dissipation is less significant than the first two.

Due to the high number of transistors, these seemingly unimportant sources of power dissipation add up and have the potential to significantly reduce battery life in idle state for mobile devices.

**Dynamic power**

The primary component of dynamic power dissipation is charging and discharging the load capacitance. If this process is repeated at an average frequency $f$, with a total charge of $Q = CV_{DD}$ being transferred each cycle, we obtain the well-known formula:

$$P_{\text{dynamic}} = CV_{DD}^2 f$$  \hspace{1cm} (2.3)

This formula has been discussed extensively in undergraduate digital design courses, so I will not dwell on it. It is interesting to note that we can amend it with an activity factor $\alpha$ and obtain

$$P_{\text{dynamic}} = \alpha CV_{DD}^2 f$$ \hspace{1cm} (2.4)

The activity factor has a value of 1 only for the clock signal, that rises and falls every cycle. This, in turn, leads to the situation in which the clock network is accountable for a significant portion of total dynamic power dissipation. See chapter *Evaluation* for relevant results. For regular logic (i.e. other than the clock network), empirical results hold that static CMOS has an activity factor closer to 0.1.

An additional source of dynamic power dissipation is the short-circuit current that results from both transistors being ON for a brief period of time, during switching. This current increases as edge rates become slower, because both transistors are ON for more time. Also, it becomes less significant if
Several techniques have been proposed to offer reduction in power dissipation:

- Clock gating - reduce dynamic power by turning the clock network OFF in some rarely used portions of the chip; area of high research interest, at some point.

- Small device capacitance - reduce device capacitance by choosing small transistors on non-critical paths. Increasing stage effort $f$ [formula 2.2], only slightly increases delays, but significantly reduces transistor size. Careful floorplanning and placement of communicating units near each other is also important for reducing device capacitance.

- Dynamic adjustment of supply voltage and frequency - reduce dynamic power.

- Dedicated circuits - reduce dynamic power by replacing general-purpose circuits with dedicated ones (where possible).

- Selective application of multiple threshold voltages - maintain performance on critical paths with low-$V_{th}$ transistors and reduce leakage on other paths with high-$V_{th}$ transistors.

- Reverse/forward body bias [8, 9] - reduce leakage by forcefully lowering or increasing $V_{t}$.

The above enumeration does not aim to be exhaustive.

Finally, because power is so important, there are several metrics researchers and engineers use to communicate results:

- Power - questionable metric, it does not explicitly account for switching
frequency.

- Power-delay product (energy) - better metric, as long as supply voltage is not dynamically adjusted. I consider this metric to be adequate for reporting my results, as I keep supply voltage constant.

- Energy-delay product - a slightly more complicated metric; it has the advantage that, once normalized for process, only varies by about a factor of two across a wide range of general-purpose microprocessor architectures [10].

### 2.2.3 Wires

Before concluding the section about circuit characterization, I address one last issue that has become relevant in recent years, from all three standpoints - area, delay and power. It concerns the interconnect - the collection of wires that link transistors together on a chip. Without going into a detailed description of the complex physical phenomena that accompany metals conductors on a chip, I will provide a qualitative description and point out the main problem we are facing today.

For years, transistors were relatively slow compared to wires, that were wide enough and thick enough not to create any problems. With each process generation, transistors, on one hand, became smaller and faster. Wires, on the other hand, became narrower and highly resistive, pushing their $RC$ constant higher than that of gates. Moreover, with increasing numbers of layers of metal, and wires packed ever-closer to their neighbours, significant capacitance also developed, adding, in turn, to wire delay. Finally, interconnect inductance, that used to be negligible, is now becoming a significant factor for systems with fast edge rates and closely packed buses [6].
The point to make here is, I believe, that chip design nowadays involves as much wire engineering as it does transistor engineering. This is why my investigation attempts to account for wires, and find out to what extent they influence the performance of my design, as described in chapter Design and Implementation.

2.3 RISC pipelines

Previous section aimed to provide interested readers with basic tools needed to understand the finer details of work I present in this dissertation. Particularly, being aware of differences between reality and ideal assumptions can prove invaluable to someone striving to make progress in this field. Moreover, supplementing notions of computer architecture with those relating to the target technology has the potential of bringing even closer together the two worlds of high-level architecture design and low-level silicon implementation, allowing a designer to easily move between levels presented in section 2.2.1.

This section moves from the lower levels of previous section to the microarchitectural and logical levels, to show what a designer’s main tools are, and to present one of the most frequently used approaches in logical design - pipelining.

2.3.1 Computer performance

Broadly speaking, the performance of a computer is affected by three key factors: instruction count, clock cycle duration and clock cycles per instruction (CPI), as described by the classic CPU performance equation:

\[
\text{Time} = \frac{\text{seconds/program}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Cycle}} \quad (2.5)
\]
An architect controls the first of these measures through Instruction Set Architecture (ISA) specifications. There are two major approaches to designing ISAs:

1. Complex Instruction Set Computers (CISC) - instructions describe complex operations, aim to reduce the size of a program, and come in a wide variety of lengths, complicating the hardware structure meant to interpret them.

2. Reduced Instruction Set Computers (RISC) - instructions describe only basic operations and have fixed length, leading to a less complicated hardware structure for interpretation. The downside is that programs tend to become longer than those for CISC.

The second category of ISAs are dominant today, because cores built for them have clear advantages in size and power, and are well-suited for multiplication on a chip. Thus, they are the ideal choice in the context of nowadays architectural paradigms - multicores and manycores.

Returning to equation 2.5 - it is clear where the lower levels of design come into play. If the ISA and the compiler are those primarily responsible for instruction count, tradeoffs at logic, circuit and layout levels influence the other two measures - cycle time and CPI. For instance, it would be possible to build a hardware structure that executes an instruction in one clock cycle. This would lead to an optimal CPI of 1. However, in doing so, we would, most likely, have to settle for a very long clock cycle. Alternatively, we could execute an instruction in multiple shorter clock cycles. A straightforward use of any of the two approaches offers no obvious benefits, as equation 2.5 shows. This is where decisions at the logical level come into play, as next subsection describes.
2.3.2 Overview of pipelining

Pipelining is a technique almost universally used today. The basic idea is to execute instructions in multiple shorter cycles, while allowing multiple instructions to overlap in execution. What we gain, primarily, is not a reduction in instruction execution time, but in overall program execution time.

Being able to pipeline an instruction is dependent on the possibility of breaking it into multiple equally-long steps of execution, and providing hardware structures to operate on it, at each step (called pipeline stage). More importantly, constructing a fully pipelined processor requires that all instructions in the ISA can be pipelined, imposing restrictions on how varied they can be. More on this in the next subsection.

Theoretically, pipelining offers the possibility of tremendous speed-ups - equal to the number of pipeline stages, if instruction count is much higher than the number of stages. This theoretical result has determined designers to build increasingly long pipelines, in the hope that performance will scale up linearly. However, this has been found to not be the case. Rather, there are points of optimum, relating to various factors. For instance, Zyuban et al. [11] report a power-performance optimal pipeline design point at 18 FO4 delays. From a purely performance related standpoint, Kunkel et al. [12] find that deeper pipelines allow for optimized execution of vector code, whereas shallower pipelines favour scalar code. The bottom line would be that pipelines cannot be indefinitely long, because, at some point, overhead between partitioning logic between stages outweighs the gains.

A benefit of pipelining is that it is fundamentally invisible to the programmer. This concept of transparency to the programmer has lead to very complicated designs, heavily involving the compiler. Such designs (VLIW computers, superscalar computers) leveraged successive gains in transistor numbers to increase functionality, but, in the process, have become almost unmanageable from a power dissipation perspective. They are, however, be-
2.3.3 Pipelining a RISC architecture

As previously stated, constructing an architecture that is suitable for pipelining is not trivial. In this regard, RISC ISAs have proven to be a very good solution, contributing to their popularity, and ultimately, their wide-spread use today.

MIPS pipeline

A classical RISC pipeline example is the one inside MIPS CPUs. The MIPS ISA has equal length instructions (32 or 64 bits), with only a few formats (field bit significance varies with instruction code). Any instruction can read up to two registers and write at most one. Register addresses are placed at fixed positions in all instructions, facilitating register read operations. Also, memory operations are performed solely through load/store instructions, on operands aligned in memory. Because of these design choices, an instruction can be executed in five stages [3]:

1. Fetch instruction from memory.
2. Read registers and decode instruction.
3. Execute an operation (including address calculation).
4. Access an operand in memory.
5. Write the result into a register.

yond the scope of this dissertation, and I will not dwell on them any longer.
A traditional *multiple-clock-cycle pipeline diagram* of five instructions is depicted in Figure 2.3.

Figure 2.3: Multiple-clock-cycle pipeline diagram of five instructions [3]

Pipelining has, of course, its limitations. There are situations when the next instruction cannot execute in the immediate clock cycle. These situations are called *hazards*, and there are three different types:

1. Structural hazards - The hardware cannot support the combination of instructions to be executed in a given clock cycle. Most often, this is an issue related to number of memories in the system. It can be easily avoided in RISC designs by employing separate instruction and data caches.

2. Data hazards - The pipeline must be stalled because one step needs to complete before the next can proceed.

3. Control hazards - Arises from the need to make a decision based on the result of one instruction, while others (possibly incorrect ones) are executing.
Data hazards

Data dependencies between successive instructions are very frequent in computer programs. As a result, stalls on the pipeline would add up to a significant amount of time, severely decreasing overall execution speed. This is why pipelines provide hardware support to deal with data hazards, in the form of forwarding logic. The basic idea behind it is that results can be delivered to a stage that requires them as soon as they are ready in another stage (e.g. Execute stage results can be fed back into the ALU, should it need them on the next clock cycle, and not wait for them to be written to the register file first). Forwarding logic has been found to have acceptable overhead, considering gains it makes possible [3].

The important thing to note is that there are cases when forwarding cannot completely eliminate stalls. These situations are handled by inserting nops on the pipeline - they do not change registers or memory, but merely create empty cycles, in wait of the required results.

Control hazards

While against data hazards forwarding proves to be very effective, there is no similarly effective solution against control hazards. There are several schemes for handling control hazards:

- **Assume Branch Taken** - always assume that a branch will be taken. Stall the pipeline until branch target is known. It is the least effective approach.

- **Assume Branch Not Taken** - always assume that a branch will not be taken. Continue fetching until branch target is known. If target is different than already fetched instructions, clear the pipeline and start fetching new instructions.
- Dynamic Branch Prediction - speculate whether the branch will be taken, based on previous executions of the given branch instruction. Simplest form is using 1-bit branch prediction buffer. More advanced approaches use 2-bit prediction buffers and branch target buffers. Even more elaborate predictors are correlating predictors and tournament predictors [3].

- Delayed Branching - with the help of the compiler, several instructions that need to be executed are placed after the branch instruction, and they are fetched while the branch target is being computed. This effectively hides branch delays. Usual implementations have 1-cycle branch delays.

Although there are many other techniques to deal with issues such as hazards and pipeline length (modern processors often have more than the described five pipeline stages), aspects I presented in this section are the bare minimum anyone interested in pipelines needs to be familiar with. Terminology introduced here will prove useful in chapter Design and Implementation, where I present the pipeline design I implemented and ran tests on (which is a five-stage RISC pipeline).
Chapter 3

Related Work

Long before power became the primary design constraint that is today, researchers have been looking into ways of creating mathematical models to describe it. These models, researchers envisioned, would be used to help designers better estimate power requirements of units they were using, thus improving time-to-market and reliability of devices. There have been several approaches, and quite a substantial number of tools have been designed/enhanced in the process. This chapter aims to provide a brief overview of some of these achievements, and how they relate to my work.

Work in the area of power modelling can be divided taking into account the design level at which it is targeted. Architectural level power modelling is meant to speed up design space exploration for architects. At this level, estimates are usually less accurate and are based on analytical models. At RTL level, a design’s logical functionality can be verified and more details (actual resources used to implement functional units) become available for power modelling. Gate level and circuit level phases of design add even more information for use in potential power models. Finally, layout level power modelling is the most detailed one, and, very often, information it provides is used, in conjunction with that from gate level, to refine models at higher
Among these levels of model detail, RTL saw the most interest from researchers. Additionally, there has been a great deal of interest to enhance models obtained here with data from gate-level simulations. Finally, researchers have paid significant attention to integrating such models into high-level simulators.

\section{RTL power modelling}

The simplest power models were based on \textit{Register Transfer Level} (RTL) simulations, and approximated power dissipation of functional units using a single value - the average power dissipation \cite{13, 14}. Based on the fact that switching activity is the primary source of power dissipation in CMOS logic (it was then, and it is now, for low-leakage cells), authors obtained the average value by stimulating the unit with uniform random inputs. To compute the power of a larger system composed of such functional units, they were adding average power values. As I will show in chapter \textit{Evaluation}, this method does not provide accurate results.

A more complex RTL approach \cite{15} took input statistics into account, and tried to fit two coefficients to data - one for uniform random bits, the other for special bits (control bits, for instance). This model degrades to the previous, if there are no special bits.

An important observation came from Mehta et al. \cite{16}, who stated that closely related input \textit{transition vectors} have similar power dissipation. While this is not the case for every piece of logic, it is definitely worth taking into account. In my work, I make use of this observation by incorporating \textit{Hamming distances} between successive inputs into some of my models.
Benini et al. [17] were some of the first to apply *linear regression* to derive power models from RTL simulations. Their solution proposed correlating switching activity on inputs and outputs of a unit to its power dissipation. They took it one step further, and introduced *regression trees* - a method of automatically producing multiple linear models, based on values of control inputs. Their solution is, however, only applicable to hard IP cores, for which there is sufficient information at RTL simulation time. Furthermore, they only model purely combinational logic. Finally, their approach, while more elaborate than previous ones, still suffers from inaccuracies due to its results being based only on RTL simulations (rather than more detailed gate-level simulations).

### 3.2 Architectural level power modelling

Over the last decade, significant work in this field has been done by David Brooks, at Harvard.

Brooks et al. [18] proposed a tool called *Wattch* to help architects make choices early-on in the design process. *Wattch* power models are based on equation (2.4) describing dynamic power dissipation. They use a set of rules, most of them described in previous chapter, to determine capacitances and activity factors to include in each unit model. Models provided span a wide range of structures usually found in a processor, and offer parameterization possibilities. As authors state, *Wattch* is intended not as a replacement to more detailed modelling methods, but merely as a fast method of exploring the design space. The idea of building such a tool has real merit, so I believe it should also be pursued with other types of power models (other than analytical ones). Namely, models I propose in this dissertation (based on results of gate-level simulations) could end up in such a high-level simulator, offering designers cycle-accurate power numbers, without the need to resort to lengthy lower-level simulations.
Another proposed tool is *PowerTimer* [19]. Power models used by this tool are extracted using several methodologies.

(i) For very early stage modelling, it uses latch-based energy models (for non-array chip portions). More specifically, the design team estimates latch counts based on prior experience, adjusted for current process. This type of model does an acceptable job because, in current processors, clocked latches account for a significant portion of power dissipation. Care must be taken, though, as it has been reported that this assumption introduces errors, in the context of present-day clock-gated designs [20].

(ii) The second method of extracting power models is an instance of an approach presented in previous section — use detailed circuit-level simulation results of individual units in a system. These simulation results are used to construct linear models of the form

\[
\text{Power} = C \times SF + \text{Hold}
\]

where \(C\) is a unit-specific constant, \(SF\) is the input *switching factor*, and \(Hold\) is a measure for power when no switching occurs. A number of such unit models are being used to construct models for larger units, through a process of *summing* — sum \(\text{Hold}\) values, average \(C\) values. Detailed data required for these models comes from units built in previous technology generations, so the authors also address the issue of *adequate scaling*.

(iii) For units that are unavailable in older processes, detailed simulations are performed to obtain energy readings. The authors then use these readings to build analytical equations capable of modelling combinations of such fundamental blocks as latches, multiplexers or interconnect.

(iv) The final path to get energy models is similar to the one used by *Wattch* — analytical equations developed for regular structures, with the use of technology-specific parameters, provided directly by vendors.
The second of four approaches shows the importance of inputs and their switching activity on power dissipation. It can be combined with the third approach - run detailed simulations, and use their results to create more complicated linear models (i.e. models that take into account more than just the switching factor).

One can ascertain, from what I presented so far, that running meaningful simulations and adequately sampling their results is very important for power models at the architectural level. This is why researchers have invested quite a significant amount of effort in developing and using various strategies for efficiently sampling the design space to derive power models. Their tool of choice was, as in earlier examples, linear regression [21, 22]. Aspects investigated by David Brooks and other researchers included pipeline dimension, cache size or number of stages in reservation stations of superscalar computers, all from a power vs. performance perspective. Most of these results were used to enhance the previously described tool - Power-Timer.

In recent years, researchers have realized how important it is to be able to adequately select what metrics might be of interest for power model construction. Jacobson et al. [23] at IBM propose systematic methods of evaluating potential usefulness of metrics to be employed in a model. The starting point for Jacobson’s work is a result presented by Powell [24] - “linear regression models based only on 9 usage events (attributes) can predict core power to within 8% average error”. However, Powell provided no algorithm or guidelines for selecting best attributes for model construction. Key results presented by Jacobson et al. include:

- an algorithm that allows selecting a subset of attributes for inclusion in a model. This algorithm iteratively goes through the list of attributes, starting with the one most highly correlated to power. For each attribute, it removes the corresponding correlated part from the power value, before proceeding to the next one.
• a validation of the claim that although most metrics investigated have high correlation to power (they investigated 2300), as few as 8 are sufficient to provide a good estimate (below 1.5% error) of power consumption;

• a method to scale models for design space exploration;

• a method for linear regression model validation. To validate models, the authors use a machine learning approach. They break the data (about 15000 observations of power and corresponding attributes) in two separate sets: 10% constitutes the training set, 90% the test set. They build linear models on 100 training sets and check to see how well they predict 100 test sets. They report the relative error, in percents (residual error divided by measured power).

Key ideas presented in this chapter - primarily, correlating switching activity to power numbers, and model validation techniques - form the foundation for my work, as I describe in next two chapters.
Chapter 4

Design and Implementation

This chapter motivates the choice of building a pipeline, by setting things in context, gives a detailed description of my pipeline design and presents tools I used for implementation and testing.

4.1 Pipeline Design

As stated in chapter *Background*, RISC ISAs are particularly well-suited for pipeline implementations. However, once an ISA has been chosen, implementing it requires a significant amount of work, which has to be well-justified. This section aims to provide that justification and to present choices I made in order to reach my final design.

4.1.1 The big picture

Nowadays, for reasons explained in chapter *Background*, industry has moved from uniprocessors (1 core per chip) to multicores (2-16 cores per
chip), in its transition towards manycores (32 or more cores per chip). With such high core counts, it stands to reason that we cannot employ large, superscalar cores (at least not in symmetric manycores). The better choice consists of small, shallow-pipelined RISC cores.

This is also the case with the Loki manycore, which is currently being designed in the Computer Architecture Group. The idea behind this manycore is much broader than simply cramming together RISC cores on a chip. While many details are beyond the scope of this dissertation, what I can say is that Loki has provided me with a tangible context of RISC usage in current-day designs. Moreover, I benefitted from an already designed ISA. Of course, the extended ISA itself was beyond the scope of my project, so I had to trim it down to a more manageable version, that would still allow me to run any application (with, at most, some restrictions on input-output mechanisms). This constrained ISA (table 4.1) has constituted the starting point of my design.

4.1.2 Overview of design

A diagram of the pipeline I designed can be consulted in figure 4.1. As depicted, there are five stages - Fetch, Decode, Register Read, Execute and Write Back.

Fetch stage interacts with a behavioural instruction cache to get instructions. Loki is a 32-bit RISC ISA, so all instructions are equal in length (32 bits), similar to MIPS (as described in chapter 2). In fact, Loki instructions have many characteristics of the MIPS ISA, with at least one significant difference - load instructions do not put memory contents directly into destination registers, but issue calls to memory to deliver data into special FIFOs. The immediate effect is that there is no need for a Memory Access stage, like MIPS has. Of course, there are a number of other implications, at both the hardware and software level.
Table 4.1: Opcodes for implemented Instruction Set Architecture

<table>
<thead>
<tr>
<th>6..2</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>ALU</td>
<td>ALU.p</td>
<td>nori</td>
<td>nori.p</td>
</tr>
<tr>
<td>00001</td>
<td>psel</td>
<td>andi</td>
<td>andi.p</td>
<td></td>
</tr>
<tr>
<td>00010</td>
<td>mulhw</td>
<td>ori</td>
<td>ori.p</td>
<td></td>
</tr>
<tr>
<td>00011</td>
<td>mullw</td>
<td>xori</td>
<td>xori.p</td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>mulhwu</td>
<td>seteqi</td>
<td>seteqi.p</td>
<td></td>
</tr>
<tr>
<td>00101</td>
<td></td>
<td></td>
<td>setnei</td>
<td>setnei.p</td>
</tr>
<tr>
<td>00110</td>
<td></td>
<td></td>
<td>settlti</td>
<td>settlti.p</td>
</tr>
<tr>
<td>00111</td>
<td></td>
<td></td>
<td>settltui</td>
<td>settltui.p</td>
</tr>
<tr>
<td>01000</td>
<td></td>
<td></td>
<td>setgtei</td>
<td>setgtei.p</td>
</tr>
<tr>
<td>01001</td>
<td></td>
<td></td>
<td>setgteui</td>
<td>setgteui.p</td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
<td>slli</td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
<td>srli</td>
<td>srli.p</td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
<td>sraui</td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
<td>addui</td>
<td>addui.p</td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td></td>
<td>ldw</td>
<td></td>
</tr>
<tr>
<td>10001</td>
<td>psel.fetch</td>
<td>ldhwu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10010</td>
<td>syscall</td>
<td>ldbu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10011</td>
<td></td>
<td></td>
<td>lli</td>
<td></td>
</tr>
<tr>
<td>10100</td>
<td></td>
<td></td>
<td>stw</td>
<td></td>
</tr>
<tr>
<td>10101</td>
<td>stlw</td>
<td>fetchr</td>
<td>fetch</td>
<td></td>
</tr>
<tr>
<td>10110</td>
<td></td>
<td></td>
<td>stb</td>
<td></td>
</tr>
<tr>
<td>10111</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000</td>
<td>lui</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>11101</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>11110</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>11111</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
<td>reserved</td>
</tr>
</tbody>
</table>
Figure 4.1: Block-diagram of pipeline
Decode stage uses a 7-bit opcode field to interpret the instruction, deciding (i) which registers will be read (if any); (ii) which register will be written (if any); (iii) how the immediate field should be extended; (iv) what function should the ALU/Multiplier execute.

Register Read contains logic to handle (i) reading registers; (ii) data sent from data cache (load FIFOs and control logic); (iii) the Program Counter (PC). This stage does not contain the actual register file, or the PC Manager, inside it. These are separate bits of logic, interfaced to the RegRead stage.

Execute contains an ALU, capable of executing 15 functions, and a multiplier. The ALU is purely combinational logic, whereas the multiplier is sequential (two-stage multiplier). Although the multiplier could be used to pipeline mul operations, I currently employ it inside a state machine that takes its result every two clock cycles (should there be a multiply request), effectively cancelling pipelining possibilities. Had I not made this design choice, I would have had to break Execute in two distinct pipeline stages, thus making forwarding more difficult.

Write Back contains logic coordinating writes to both data cache and register file. It also signals load operations to the data cache, so it can deliver requested data to the load FIFOs.

4.1.3 Pipeline characteristics

This subsection builds upon the previous one, providing further detail on individual modules in the design. It also describes how the pipeline handles forwarding and stall.
Distributed stall control

In my pipeline, two-element fallthrough FIFOs (*FIFO Buffers*) replace stage registers present in the MIPS design. These buffers store results from one stage and feed it into the next one. They vary in width across the pipeline, according to datapath width between stages.

The buffer between *Fetch* and *Decode* is 32-bit in width, because only instructions travel between those stages.

The one between *Decode* and *Register Read* has to accommodate the newly created control signals (20-bit wide), an *immediate* value (32-bit wide), and three register addresses (like MIPS, Loki uses mostly three-register instructions).

Between *Register Read* and *Execute*, two values read from the register file increase buffer width (64 additional bits).

Finally, buffer width between *Execute* and *Write Back* decreases, as only values needed to write to the register file and data cache are retained (along with control signals).

I selected them to be two-element FIFOs, instead of only one, because I considered that they might offer better decoupling between stages (particularly useful on stalls).

Using control signals available from the buffers, each pipeline stage can stall locally, so there was no need to implement global stall logic. There are several possible stall sources:

- *Register Read* can stall while waiting for the data cache to deliver data to its load data.
• *Register Read* can also stall on one particular type of jump, if registers it requires have yet to pass the Execute stage (so they do not yet have up-to-date values).

• *Fetch* can stall if the instruction cache does not have the required instruction (and must wait for it to be loaded from main memory).

• *Fetch* can also stall artificially on branches (more details below).

• *Execute* always stalls one cycle on multiply operations, to allow the two-cycle multiplier to finish computing.

**Forwarding logic**

As is the case with RISC programs, Loki programs induce frequent data hazards in the pipeline. As described in chapter *Background*, forwarding logic is the solution to this problem. However, FIFO buffers between stages create complications when it comes to forwarding. This is why I implemented three levels of forwarding in my pipeline:

1. Forwarding inside *Execute* - inputs to the ALU and the multiplier can be fed from any of the two elements of the EX FIFO buffer.

2. Forwarding from *Execute* to *Register Read* - I placed a piece of forwarding logic between the register file and the Register Read stage, effectively allowing Register Read (should need be) to read values from the EX FIFO buffer.

3. Forwarding from *Write Back* to *Register Read* - this level of forwarding is implemented as bypasses at the register file level.
Register file

The register file contains 32 visible registers and 32 that can be addressed indirectly (through special instructions). Because most instructions need to read two registers (named rs and rt), the register file has two read ports. Furthermore, if Register Read attempts to read a register that is currently being written, the value to be written bypasses the register file (it is read in place of the value currently in register).

Branching

Loki handles branches in a particular manner, stemming from its treatment of programs as collections of instruction packages. A jump instruction is called fetch, because it signals that a new instruction package should be fetched into the instruction cache, and that execution should continue at the beginning of the new package, once the current one is completed. While my pipeline does not offer full support for this mechanism, it achieves a satisfactory degree by implementing delayed branching. For increased flexibility (considering that a fetch instruction can appear at any point in a package), I chose branching delay with a variable window - it starts when a fetch is detected and ends when an eop-marked (end-of-package) instruction is fetched.

This approach relies on the compiler to place sufficient instructions between a fetch and an eop, in order to hide target address computation latency.
4.1.4 Behavioural caches

Previous subsections described parts of the design that get implemented all the way down to the layout level. However, for simulation purposes, the pipeline needs to interact with caches. I provide two separate such memories, which are only meant to emulate real cache interfaces - they are not synthesizable modules, but merely simulate the outer world, as seen by my pipeline.

Data cache

The behavioural data cache holds data values programs store or load, very much like an ordinary cache. For running specific programs, it needs to hold some data the program expects to find, at specific addresses. Since the module is not synthesizable, I read this data from files.

A particular aspect of the data cache is that it does not write values straight into the register file. Instead, when it receives a load request, it send the requested content to a FIFO, where it will be consumed by a subsequent instruction.

One final note to make about the data cache is that, unlike the instruction cache, it is byte-addressable. This happens despite the fact that the pipeline operates with words, and the reason is increased flexibility. For instance, a program could replace a single byte inside a memory word, then load the entire word and use it for future computations. Note, however, that memory is assumed to be word-aligned (even if it is byte addressed).
### Instruction cache

Since instructions are all 32-bit wide, the instruction cache is word-addressed (as well as word-aligned).

This behavioural instruction cache receives an address from the Program Counter Manager module inside the pipeline and a read request from Fetch. On the next clock cycle, it offers an instruction (which can be the same as before, if the PC has not changed, like in stalls).

As I will describe shortly, I run two types of simulation on my pipeline - at the register level (purely behavioural) and at the gate level (with delays extracted from the layout). The first type of simulation poses no problems to interactions between pipeline and caches. However, the second one needs some adjustments inside caches. Namely, any input to the pipeline changing on a clock edge needs to be delayed, in order to avoid hold violations (new signals arriving before the clock edge).

### 4.2 Pipeline Implementation

Having described above my major design choices, I will now move on to describing the implementation process. I applied this procedure to the pipeline as a whole, but also to individual modules inside the pipeline, for separate testing.

#### 4.2.1 Hardware description

Hardware implementation starts with a description of the hardware, using a Hardware Description Language (HDL). There are several approaches one
could take to describe their design, from a very high level (system level), to a very low level (lists of nets). High-level approaches are becoming more popular, but they have a clear disadvantage - they place levels of abstraction between the designer and the underlying technology, removing engineers ever farther from the actual circuit [6]. CAD tools are becoming increasingly relied on, and our demands of them increase continuously. The upside is that this trend has increased productivity. Furthermore, it is reversible, in the sense that hybrid approaches can be taken - tools do allow engineers to get into what level of detail they feel to be necessary.

It is also the case with the Synopsys tools, which I used in conjunction with a Register Transfer Level (RTL) description of my hardware. I found an RTL description of the hardware to be adequate to the scope of my project, because it allows me to have a significant level of control over exactly what resources should be placed in a module, without sacrificing too much of the productivity associated with higher levels of description. The language of choice for this description was System Verilog. It not only allowed me to describe the hardware in a fairly elegant manner, but also to set up simulation environments, for debugging and experimenting.

Results of this lengthy stage in my project include descriptions for each module inside the pipeline, connections between them, and testbenches I used for debugging. They add up to thousands of lines of code, so they are too big to include in this dissertation. However, I have made everything publicly available for consultation in the project archive.

4.2.2 Synthesis

This subsection and the next describe tools and parameters used during implementation. Constraints I will describe for synthesis also apply to place and route.
To translate the hardware description into logic (synthesis), I used the Synopsys Design Compiler (DC). It required several iterations before I could settle on what I consider to be an adequate frequency to drive my pipeline. I chose this to be 307.5 MHz for reasons concerning timing and power. First, with some adjustments to the hardware description, the pipeline could be driven at higher frequencies. However, increasing frequency is of little relevance to the scope of my project - adequate metrics can be used to account for frequency (see chapter Background). A second, more important aspect is that, for a given maximum frequency attainable, logic on the critical paths will come under a lot of strain, leading to overestimations in power numbers. This is why I relaxed the imposed frequency to a tolerable level.

However, achieving proposed frequency is not trivial. To get good results, one must make DC work harder. For my design, I employed 25% clock speed-up in DC. Setting a 25% higher frequency target makes sure that the synthesized design meets timing with a lot to spare. This is because next stage introduces real delays, due to cell placement and interconnects, and it is highly demanding in terms of time.

Aside from the clock frequency, I also use a constraint that helps avoid setup violations - when evaluating time, I force DC to account for a delay between clock edge arrival and other inputs being changed. I set this delay to 300ps, which adequately accounts for clock-to-register delay and some interconnect delays.

Since my project also involved passing individual modules through DC, I had to add a supplementary constraint for modules without a clock signal (purely combinational logic). Namely, I had to constrain the time interval from inputs to outputs. This constraint varied from one module to the other, depending on their size and their position in the pipeline.
Table 4.2: Process-voltage-temperature (PVT) corners for standard cells library

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Process case</th>
<th>Temperature (°C)</th>
<th>Supply voltage (V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC</td>
<td>worst</td>
<td>-40</td>
<td>0.99</td>
</tr>
<tr>
<td>TC</td>
<td>typical</td>
<td>25</td>
<td>1.10</td>
</tr>
<tr>
<td>BC</td>
<td>best</td>
<td>0</td>
<td>1.21</td>
</tr>
</tbody>
</table>

4.2.3 Place and route

Final step in getting the implementation consists of choosing adequate cells from a standard cells library, placing them according to time, area and power constraints, and routing the entire design to achieve desired functionality. The tool I used in this phase was Synopsys Integrated Circuit Compiler (ICC), with constraints similar to those for DC (except clock speed-up, which I set to 11%).

The ICC flow I employed is fairly complex, touching on all major practices in industry, for ensuring proper design margin:

- It uses cells from a low-power, low-leakage 45nm process. Lower leakage is achieved by setting a high threshold voltage ($V_t$). Typical supply voltage is $V_{DD} = 1.1V$.

- It performs timing analysis, to ensure timing is met, for worst case (WC) process-voltage-temperature (PVT) conditions (see table 4.2). Notice the low temperature for WC. It is due to a phenomenon termed temperature inversion.

- Checks for hold violations using best case (BC) conditions.

- Adds margins, to take on-chip variation into account ($-10\%$ to $8\%$).

- Assumes skew and jitter on the clock (up to $3.5\%$).
Figure 4.2 presents the final result of this phase - a wired array of standard cells, built in a 45nm process. I constrained the width of the array to 125µm, and obtained an array height of approximately 250µm. Pipeline inputs are the top, outputs are at the bottom.

![Figure 4.2: Placed and Routed pipeline](image)

### 4.3 Preparing evaluation

The implementation in figure 4.2 constitutes the starting point of the evaluation process. In order to avoid over-encumbering chapter *Evaluation* with description of my testing methodology, I present it in this section. I give a detailed description here, and will refer to it whenever needed, in next
chapter. This, I consider, allows me to present results more clearly.

4.3.1 Overview of approach

This subsection aims to clarify how building a pipeline fits in with developing power models for individual units.

Power modelling can (and needs to) be done at finer-grained levels than that of a pipeline. As next chapter will show, I take special interest in modules inside the pipeline, and not the pipeline itself. However, modelling units that are not part of a system means there is virtually no way of reporting how those units would fit in a larger system. Basically, I would lack a solid test environment.

Also, since I am using a low-leakage standard cells library, most of the power dissipation comes from switching activity. This, in turn, means that I must stimulate the unit with some inputs, each cycle. One approach would be to use random inputs (possibly constrained, in some sense). I do this, and I obtain what I call baseline readings. However, to test models, one would have to use inputs generated by a real program. This is where the pipeline comes in. Ensuring complex synchronization mechanisms between pieces of logic inside itself, the pipeline is capable of executing instruction traces from real programs.

Moreover, taking into account how big an impact the place and route process has on a design, modelling power for a unit that is not part of a complex system would not be sound methodology. Using the routed pipeline, models can be used to point out major sources of error - differences between unit layout when it is on its own versus when it is inside the pipeline, or additional delays and power consumption due to interconnects.

To sum up, the general approach is to record switching events inside units,
each cycle, and to correlate those to per-cycle energy consumption. Recorded events vary from one unit to the other, as shown in next chapter. To extract models, I use multiple linear regression, in $R$.

### 4.3.2 Toolflow description

Achieving what previous subsection describes is not as straightforward as might seem. Rather, it entails usage of several tools and multiple iterations before obtaining a satisfactory result for each unit. This subsection describes the steps involved in building a model.

**Generate implementation layout**

Under this name, I group the previously presented steps - hardware description, synthesis, place and route. These steps are part of the methodology because it often happened, while I was building a model, to go back, tweak the hardware description, and redo the model. This helped me get an insight into how tools were operating and what measures were most adequate to include in my models.

**Build Standard Delays File**

As presented in chapter *Background*, $R$ and $C$ values can be extracted from the layout, for the purpose of determining delays and estimate power. To accurately extract these values, I used a tool called Synopsys *StarRC*. While I will not go into details of its operation, I must say that StarRC is a highly appreciated tool in industry because of its speed of operation, the multitude and the degree of accuracy of parasitics it can determine for large designs - it can determine *subfemtoFarad* parasitic capacitances in processes.
that go as low as 28nm.

StarRC produces results that can be back-annotated onto the gate-level netlist and used by another tool, called Synopsys PrimeTime (PT). This tool is, in fact, a software suite. For the purpose of this stage, I used its component that computes delays based on the back-annotated netlist and library data. Computed delays are then placed in a Standard Delay Format (SDF) file. Such a file can be produced at any point in ASIC design flow. However, the one I obtain here is particularly important because it contains accurate post-layout data.

Run gate-level simulation

The SDF file is used to back-annotate the design and run a detailed gate-level simulation. For all simulations I ran (RTL or gate-level) I used a tool called Synopsys VCS, which offers System Verilog support, simulation engines and debugging environment (among other features). Occasionally, I would use a waveform viewer (GtkWave), for lightweight debugging.

At this step, I collect all switching activity in my module, because, as I said above, it is the one primarily responsible for power dissipation. In order to capture meaningful activity into the vcd file I generate at this stage, I need to construct a testbench that adequately stimulates the module under test. As next chapter presents, I used random inputs to create my baseline models.

It is also at this stage that I need to think about what the adequate measures to include in my model could be. This is a fairly non-standard process, involving me going back at the module hardware description, to help make a decision about events I might capture and place in a separate file. Such events could be, for a FIFO, the existence of a read or write operation in a given cycle, or whether successive inputs varied greatly among each other
(usually quantified through Hamming distance).

**Power estimation**

Switching activity collected previously gets used at this stage by a different component of PT, which generates cycle-accurate power dissipation values. It also offers an insightful summary of average power dissipation per cycle, including a break-down of values among various sources of power dissipation (clock network, registers etc). The important thing to note here is that power values I obtain are highly dependent on what actions the device under test performs during gate-level simulation.

These power numbers are collected under TC conditions (see table 4.2). WC conditions would produce lower number, whereas BC conditions would produce higher numbers.

**Final step**

Finally, with both power numbers and events of interest at my disposal, I run a series of Perl scripts to generate a file containing energy values, on one column, and values for each event of interest, on other columns. Each line in this file represents values at a given moment in time.

This file format is carefully chosen to be readable by $\mathbf{R}$, the tool I use next. In $\mathbf{R}$, I look for correlation between captured events and energy numbers, and do multiple linear regression to get a model of energy versus each of the inputs. As one can imagine, for large units, events I capture can become quite numerous, resulting in large amounts of data. However, as next chapter will show, models I present are fairly simple. More on that at the right time.

Occasionally, this step would only serve to construct preliminary models,
before going back and deciding on a final testbench form.

To conclude this chapter, I provide a diagram of my workflow in figure 4.3. The toolflow box is an abstraction of all steps described above, save the final step.

Figure 4.3: Project workflow
Chapter 5

Evaluation

Current chapter provides details on steps I took to derive energy models, presents these models and provides a summary of validation results.

5.1 Pipeline verification

The entire project relies on an infrastructure I built almost from zero - using provided specification, a set of guidelines and a number of System Verilog code examples. This justifies the significant effort I had to make before I could reach a phase where modelling energy consumption would yield usable results.

As described in previous chapter, the pipeline design contains a number of submodules that need to be implemented. I took a bottom-up approach for doing this:

1. Construct major building blocks of stages - FIFO buffer, ALU, multiplier, forwarding logic, PC manager, load FIFO etc.
2. Construct pipeline stages.

3. Construct logic to synchronize stages, update stage logic to handle local stalls.

4. Construct pipeline unit, and interface with behavioural caches.

5.1.1 Submodule verification

To ensure implementation correctness and to determine adequate usage for each individual unit inside a larger structure, I had to do both behaviour verifications and timing analyses.

Behaviour

I accompanied each module description with an RTL testbench, providing test stimuli to the module and checking against expected outputs. This endeavour varied in complexity from simple FIFOs, where outputs are checked against inputs in a straightforward manner, to the more complicated Decode or Forwarding logic units. Testbenches can be consulted in the online archive.

Timing

In order for me to get a clear picture of what impact a module would have once added to the pipeline, I did timing analyses, using Synopsys DC. This helped me make various implementation decisions, two of the most important being described below:

- The multiplier was the slowest unit. Built in a straightforward man-
ner (letting System Verilog choose the implementation), the multiplier could not meet timing for clock periods shorter than 4\(\text{ns}\). This, in turn, meant that \textit{Execute}, the stage hosting the multiplier, could not run at frequencies higher than 250MHz (because of some additional delays in the stage, this number would have gone down to about 200MHz, in the end). To overcome the problem, I used a library two-stage multiplier, which I integrated in a \textit{finite state machine} (FSM) running inside \textit{Execute}.

- The second slowest unit was the ALU. My implementation of this unit made it impossible to meet timing for clock periods shorter than 2\(\text{ns}\). Considering that the ALU is on the critical path inside \textit{Execute} (so additional penalties would be incurred), and looking to avoid placing too much strain on the unit (which would have lead to artificially high power numbers), I made the decision to drive my pipeline at a frequency of 350MHz. Because of some additional delays in the stage, this number went down to 307.5MHz, as presented in previous chapter.

5.1.2 Pipeline testing

A complex construct, like a pipeline, is notoriously difficult to verify, even when submodule functionality is fully known. Testament to this is the fact that design verification has become a distinct branch of science and engineering. Among the numerous approaches proposed, I chose a widely used method - create test cases to cover the testing space in a statistically significant manner, thus eliminating most dysfunctions. I do this by running instruction traces taken from real programs with relatively different pipeline usage profiles. The benchmark I used for simulation is composed of six programs. They are, as follows, in ascending order of number of cycles needed for their execution:

1. \textit{Qsort} - quick sorting of a vector of integers in memory.
2. **CRC** - cycle redundancy check computation for a 150 character long string. The same algorithm is used for frame check sequence in ADCCP (32-bit variant).

3. **Bitcnts** - count bits for a sequence of random numbers, omitting leading zeroes.

4. **String search** - search if a list of strings can be found among the entries in another list of strings. It uses the Boyer-Moore string search algorithm.

5. **Basicmath** - Performs three distinct sets of tasks: (i) solve cubic equations (2 with specified coefficient; 10,000 with random coefficients); (ii) compute integer square roots (1,000 computations); (iii) convert angle values (360 conversions from degrees to radians, 360 conversions from radians to degrees).

6. **Dijkstra** - find the 10 shortest paths in a 100 nodes graph, using Dijkstra’s algorithm.

Using these programs, I checked pipeline functionality by (i) following signals with a waveform viewer (in early stages of debugging), (ii) dumping memory and register contents to files, and (iii) displaying program results (**printf** statements). I considered pipeline verification to be satisfactorily completed when programs ran completely and produced desired results.

### 5.2 Unit models

Previous section has described an essential step in my project - build test infrastructure. Once I proved that the pipeline functions correctly, I could consider that step completed and stop making changes to the hardware de-
scription of modules inside the pipeline. This allowed me to continue to the next step - the analysis of energy consumption of individual units.

Section 4.3 provided a generic description of my workflow, at the pipeline level. However, a similar approach can be taken with each individual unit inside the pipeline. Taking this into account in current chapter, I only provide description of details particular to model construction (for each unit, in relevant subsections). These details include decisions I made about:

- what (if any) particular patterns should stimuli have to create a sensible, yet generic enough model. For most models, data inputs I use are random (with uniform distribution), but control inputs have some constraints, which I derive from usage profiles of that particular unit inside the pipeline.

- what attributes should I capture for use in model construction. As this section will show, I usually capture more attributes than are in the final version of my model. This is because, in spite of most of them having high correlation to power dissipation ($p < 0.01$), a subset is usually enough to get an acceptable compromise between model simplicity and model accuracy [24, 23].

To evaluate model quality, I use the following metric:

$$\text{err} = \frac{\varepsilon}{E} \cdot 100[\%]$$

(5.1)

where $\varepsilon$ is the model residual standard error, as reported by $R$, and $E$ is the average energy consumption of the unit under test, computed using power numbers provided by Synopsys PrimeTime. For each model, I attempted to minimize this value, while making sure the model does not get too complex.
5.2.1 Fifo buffer

First piece of logic I model is the FIFO buffer - a fallthrough two-element FIFO that I use to store results from one stage of the pipeline and to feed them to the next stage (see figure 4.1). It is built around a three-state FSM - FIFO empty, FIFO full, FIFO has elements. One such FIFO buffer gets used for each stage, each with its own width, as described in previous chapter (see section 4.1.3).

Unit stimuli:

- random data inputs;
- constrained random control inputs - do not write to full buffer, do not read empty buffer.

Captured attributes:

- Hamming distance - normalized Hamming distance between successive inputs; describes switching activity in a data-independent manner;
- number of ones in written inputs - correlates to power dissipation, although not as strongly as I might have anticipated;
- write operation;
- read operation;
- FIFO status - 3-bit variable offering one-hot encoding for FIFO state.

Fetch FIFO buffer power statistics obtained by running a testbench with random inputs are displayed below. I call these baseline statistics.

Table 5.1 shows several interesting pieces of information.
Table 5.1: Power numbers for FIFO buffers

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch FB</td>
<td>clock network</td>
<td>48.92</td>
<td>$E_{pk} = 1.16pJ/op$</td>
<td>$E = 0.6pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>12.20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>38.88</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode FB</td>
<td>clock network</td>
<td>50.21</td>
<td>$E_{pk} = 1.7pJ/op$</td>
<td>$E = 0.88pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>15.51</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>34.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegRead FB</td>
<td>clock network</td>
<td>53.9</td>
<td>$E_{pk} = 3.1pJ/op$</td>
<td>$E = 1.50pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>22.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>23.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute FB</td>
<td>clock network</td>
<td>51.82</td>
<td>$E_{pk} = 2.23pJ/op$</td>
<td>$E = 1.07pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>18.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>29.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First, power dissipation due to clock network accounts for about half of the total power dissipation of the unit. This comes as a proof to some of the statements I made in chapter *Background*, about why researchers have been interested in clock gating techniques.

Second, as module width scales up, average energy increases almost linearly, as figure 5.1 shows. A close analysis of power dissipation percentages shows that this linear scaling is actually the result of compound scaling: (i) nearly linear scale *up* of register power (was expected, considering that buffer scale up means adding extra storage capacity); (ii) nearly linear scale *up* of clock network power (makes sense, because additional registers need additional clock paths); (iii) nearly linear scale *down* of combinational logic (it does not need to increase in size with buffer scale up, so its ratio of the total goes down).

One final note about the baseline statistics: dynamic power dissipation
accounts for about 99% of the total power dissipation, which leaves about 1% for leakage, glitches, X-transition power and other static phenomena; this comes as a proof to my claim that the standard cells library I am using is built in a low-leakage process.

Aside from generating the baseline statistics, Synopsys PrimeTime provides, based on module switching activity, cycle accurate power numbers. As presented in previous chapter, I input these numbers into a set of *Perl* scripts to generate a file with data formatted in a manner readable by *R*. Figure 5.2 presents the first lines of such a file, for FIFO buffers. Each variable is arranged on one column, and each line represents data collected at a given point in time. The difference between two successive values on the time columns is 3.25 - the length of one clock period.

Figure 5.1: Average energy vs. FIFO buffer width. A linear trend is evident.
Using this file as a data source, I then do multiple linear regression analysis in R to get a model that relates the energy consumption to some/all of the attributes captured. With some effort involved in choosing the right combination of inputs (as described above), I get the models presented in table 5.2.

Notations used in models signify: write - write took place, read - read took place, status - buffer status (integer), hdi - normalized hamming distance between successive inputs (float), \( rs2 \) - read took place while buffer in state 2, \( rs3 \) - read took place while buffer in state 3, \( ws1 \) - write took place while buffer in state 1, \( ws2 \) - write took place while buffer in state 2. For each unit, the last row indicates normalized residual errors, as reported by R. Finally, all variables used in models exhibit a significant correlation to energy (\( p < 0.01 \)).

A first look at the table would seem to indicate that models for FIFO buffers in RegRead or Execute are much weaker than the other two (because of the high residual errors). Let us present some computations to help decide model strength. At first glance, one might compute

\[
\frac{\varepsilon_{RR}}{\varepsilon_F} = \frac{0.609}{0.169} = 3.603
\]

which would give an almost four times increase in error. However, taking the
Table 5.2: Models for FIFO buffers

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model1</th>
<th>Model2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch FB</td>
<td>$E = 0.182<em>1 + 0.179</em>write + 0.145<em>read + 0.051</em>status + 0.028*hdi$</td>
<td>$E = 0.297<em>1 + 0.096</em>rs2 + 0.320<em>rs3 + 0.164</em>ws1 + 0.192*ws2$</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 0.178$</td>
<td>$\varepsilon = 0.169$</td>
</tr>
<tr>
<td>Decode FB</td>
<td>$E = 0.434<em>1 + 0.242</em>write + 0.348<em>read + 0.059</em>status + 0.201*hdi$</td>
<td>$E = 0.539<em>1 + 0.259</em>rs2 + 0.647<em>rs3 + 0.321</em>ws2$</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 0.368$</td>
<td>$\varepsilon = 0.359$</td>
</tr>
<tr>
<td>RegRead FB</td>
<td>$E = 0.613<em>1 + 0.662</em>write + 0.613<em>read + 0.151</em>status + 0.463*hdi$</td>
<td>$E = 0.886<em>1 + 0.447</em>rs2 + 1.333<em>rs3 + 0.931</em>ws1 + 0.771*ws2$</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 0.640$</td>
<td>$\varepsilon = 0.609$</td>
</tr>
<tr>
<td>Execute FB</td>
<td>$E = 0.498<em>1 + 0.375</em>write + 0.419<em>read + 0.106</em>status + 0.736*hdi$</td>
<td>$E = 0.696<em>1 + 0.415</em>rs2 + 0.857<em>rs3 + 0.578</em>ws1 + 0.384*ws2$</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 0.466$</td>
<td>$\varepsilon = 0.422$</td>
</tr>
</tbody>
</table>

average energy into consideration, we can compute more meaningful values:

$$\frac{\varepsilon_{RR}}{E_{RR}} = \frac{0.609}{1.5} = 0.406, \frac{\varepsilon_F}{E_F} = \frac{0.169}{0.6} = 0.281 \quad (5.3)$$

Taking into account these values, the relative increase in error is, in fact:

$$\frac{0.406}{0.281} = 1.44 \quad (5.4)$$

or about 44%. So, the error does increase, but it is not dramatic.
The increase in error happens as the width of the buffer increases, which would seem to suggest that the models lack a way of accounting efficiently for power dissipated in storage elements. An approach would be to adapt a hybrid model, by including a width variable and refitting data. However, such a model would have required extra investigation. Furthermore, adopting this type of model for every unit would have dramatically increased project duration. This happens because of the manner in which I extract models - each time a parameter changes in the hardware description, I have to start at the beginning of the workflow described in previous chapter. For complex units (it is not the case for FIFO buffers), some of the steps involved (especially Place and route and gate-level simulation) take very long to complete.

Most of the aspects discussed in this subsection apply to the other units, as well. Next subsections present only results and some key observations.

5.2.2 ALU

The ALU is a piece of purely combinational logic, capable of executing 15 distinct functions, on two 32-bit operands.

Unit stimuli:

- send random data inputs (operands);
- send random function input (with a bias towards add operations, to better simulate the ALU usage profile inside a pipeline).

Captured attributes:

- Hamming distances between successive operands;
- Hamming distances between successive outputs;
• operation executed;

• if the same operation was executed twice in a row;

• deactivate bit value - control bit used to signal ALU results are not required (it proves not to be correlated to energy consumption).

ALU baseline statistics are displayed below.

Table 5.3: Power numbers for ALU

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>clock network</td>
<td>0.0</td>
<td>$E_{pk} = 2.18pJ/op$</td>
<td>$\bar{E} = 0.85pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>99.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since this is purely combinational logic, power dissipation is distributed accordingly.

Derived model for this unit is presented below.

Table 5.4: Model for ALU

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>$E=0.217<em>1+0.059</em>hd_a+$ $0.072<em>hd_b+$ $0.010</em>hd_{out}^-$ $0.006*func$ $\varepsilon = 0.21$</td>
</tr>
</tbody>
</table>

Notations used in table 5.4 signify: $hd_a$ - Hamming distance between successive operand 1, $hd_b$ - Hamming distance between successive operand 2, $hd_{out}$ - Hamming distance between two successive outputs, $func$ - operation executed.
5.2.3 Multiplier

From a black-box perspective, the multiplier serves a function similar to the ALU. This is why stimuli and captured attributes are virtually identical. Power numbers are displayed in Table 5.5.

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock network</td>
<td>19.9</td>
<td></td>
<td>$E_{pk} = 9.13pJ/op$</td>
<td>$\overline{E} = 6.04pJ/op$</td>
</tr>
<tr>
<td>register</td>
<td>5.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>combinational</td>
<td>75.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is immediately evident that the multiplier is significantly more power-hungry than previous units. Numbers presented above make the multiplier the single most power-demanding unit in the pipeline (aside from the register file), which means that its modelling should be done more precisely.

To build the multiplier model, I used a testbench that stimulates the multiplier with different inputs each cycle. An alternative testbench I used stimulated the multiplier with the same input for two consecutive cycles. Understandably, power numbers for the two cases are vastly different ($\approx 50\%$). Multiplier usage profile inside the pipeline is somewhere between the two - there are numerous cycles when the inputs change, but there are also a lot of pipeline stalls, which I cannot accurately model in my testbench, without knowing the stalling profile of the pipeline (I have not studied this). As a result, I have to content with the model I present below, but make a note that all energy estimates based on it are likely to be higher than actual values.

One final comment: if computed in a manner similar to equations (5.3, 5.4), the residual error for this unit would yield a better result than that of the ALU. The only problem is, as mentioned before, its large energy consumption, which means that even though the error is relatively smaller, it is, in
Table 5.6: Model for Multiplier

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>$E = 3.44 \times 1 + 0.220 \times h_{da} + 0.224 \times h_{db} + 0.024 \times h_{d_{out}} - \varepsilon = 0.63$</td>
</tr>
</tbody>
</table>

5.2.4 Forwarding logic

To tackle data hazards, two such pieces of logic function inside the pipeline - one inside the *Execute* stage, the other between the *Execute* stage, register file and the *RegRead* stage. This is a relatively small unit, as proven by power numbers presented below.

Unit stimuli:

- send random data inputs (register contents);
- send random control inputs, but make sure to create forwarding situations (which are too unlikely to happen with a purely random set of control inputs).

Captured attributes:

- Hamming distances between successive register contents from primary inputs;
- Hamming distances between successive register contents from secondary inputs;
• status of the FIFO buffer providing the secondary inputs (those that might get forwarded);

• control values indicating register read and write operations (needed by *forward* to make a decision);

• values indicating that the source address of primary inputs is the same as the destination address of secondary inputs (also needed for forward decision).

Table 5.7: Power numbers for Forward

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>clock network</td>
<td>0.0</td>
<td>$E_{pk} = 0.89pJ/op$</td>
<td>$E = 0.24pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>99.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Model for Forward

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>$E=0.207*1+$</td>
</tr>
<tr>
<td></td>
<td>0.192*rs0+</td>
</tr>
<tr>
<td></td>
<td>0.074*rs1+</td>
</tr>
<tr>
<td></td>
<td>0.123*rt0+</td>
</tr>
<tr>
<td></td>
<td>0.192*rt1</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 0.06$</td>
</tr>
</tbody>
</table>

Notations used in table 5.8 signify: $rs0$ - forwarded $rs$ from first element of *Execute* FB; $rs1$ - forwarded $rs$ from second element of *Execute* FB; $rt0$ - forwarded $rt$ from first element of *Execute* FB; $rt1$ - forwarded $rt$ from second element of *Execute* FB.

Although this unit is fairly small in comparison to the others, it is important because: (i) there are two instances of it in the pipeline; (ii) both of those instances are on critical paths.
5.2.5 Register file

The register file has a memory-like regular structural and would, therefore, be more adequate to modelling using parameterizable models. Nevertheless, I applied the same technique as for the other units. The register file is composed of 64 registers, each 32 bit long.

Unit stimuli:

- send random control inputs on the two read ports;
- send random control inputs on the write port (with minor changes, to create bypass situations).

Captured attributes:

- was write operation;
- was read operation;
- did bypass on port 1;
- did bypass on port 2;
- Hamming distance between successive outputs;
- Hamming distance between successive inputs;
- number of ones in the outputs.

Baseline power numbers are presented below.

These results show, as it was expected, that the register file is an extremely large source of power dissipation. An apparently surprising find is
Table 5.9: Power numbers for Register File

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock network</td>
<td>7.43</td>
<td>43.43</td>
<td>$E_{pk} = 19.83pJ/op$</td>
<td>$E = 7.87pJ/op$</td>
</tr>
<tr>
<td>register</td>
<td>29.09</td>
<td>63.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>combinational</td>
<td>63.49</td>
<td>63.49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequential</td>
<td>0.0</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

that combinational logic consumes more power than registers inside the register file. These is perfectly explainable, though - read logic is implemented in a combinational fashion, to provide outputs as soon as possible for the Register Read stage.

Table 5.10: Model for Register File

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$E=7.16*1+$</td>
</tr>
<tr>
<td></td>
<td>$0.445*write+$</td>
</tr>
<tr>
<td></td>
<td>$0.327*by1+$</td>
</tr>
<tr>
<td>Register File</td>
<td>$0.220*by2+$</td>
</tr>
<tr>
<td></td>
<td>$0.015*oc1+$</td>
</tr>
<tr>
<td></td>
<td>$0.027*oc2$</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 1.93$</td>
</tr>
</tbody>
</table>

Notations in table 5.10 signify: write - there was a write operation; by1 - bypass on read port 1; by2 - bypass on read port 2; oc1 - number of ones in output of read port 1; oc2 - number of ones in output 2.

It is immediately apparent that for this model I had to sacrifice some of the simplicity of previous models. Moreover, the end result is not satisfactory - with such a high residual error, estimates based on this model will, most likely, be higher than actual values. This model, I believe, can be further refined using a hybrid approach or by employing new, more complex methods of capturing meaningful switching activity.
5.3 Pipeline stage models

Models I present in this section are the result of knowledge I gathered during previous steps. They represent the final stage of power modelling in my project, making it possible for me to start making use of the whole infrastructure, for evaluation purposes. Modelling them ensures that I do not miss significant pieces of logic from my pipeline (e.g. each stage contains stall logic, which I did not model previously). Also, by comparing average energy values of FIFO buffers with those of stages that contain them, I draw an interesting conclusion: it would seem that average energy tends to overestimate how much energy that unit is going to consume when placed inside a larger unit.

5.3.1 Fetch

Unit stimuli:

- send constrained random instructions (instructions with a uniform distribution across an opcode set);
- use random control signals from the instruction cache and Decode, in a manner similar with in-pipeline behaviour (do not ask for instruction when Fetch is not ready, respond adequately to stalls etc.).

Captured attributes:

- stage asked for input;
- stage provided its output to next stage;
- stage stalled;
Baseline statistics for Fetch are presented below.

Table 5.11: Power numbers for Fetch stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>clock network</td>
<td>54.71</td>
<td>54.71</td>
<td>E_pk = 1.12pJ/op</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>13.66</td>
<td>13.66</td>
<td>E = 0.64pJ/op</td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>20.70</td>
<td>20.70</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>10.93</td>
<td>10.93</td>
<td></td>
</tr>
</tbody>
</table>

As expected from a fairly complex piece of logic built around an FSM, Fetch exhibits some power dissipation due to sequential logic. The clock network is, again, the main consumer. Overall, this stage behaves similarly to its buffer, because the buffer represents a significant part of it. That is why the model I derive is, to some extent, shaped after that of a FIFO buffer.

Table 5.12: Model for Fetch stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>E=0.969<em>1-0.063</em>output-0.327<em>input-0.220</em>stalled-0.027*dropped</td>
</tr>
<tr>
<td></td>
<td>( \varepsilon = 0.184 )</td>
</tr>
</tbody>
</table>

Notations used in table 5.12 signify: output - Decode asked for new output from Fetch; input - Fetch asked for new input from the instruction cache; stalled - Fetch was not asked to provide new output and it did not ask new input; dropped - Fetch asked for new input from the instruction cache, but it did not provide it to Decode.
5.3.2 Decode

Unit stimuli:

- send constrained random instructions (instructions with a uniform distribution across an opcode set);
- use random control signals from Fetch and RegRead, in a manner similar with in-pipeline behaviour (do not ask for instruction when Decode is not ready, respond adequately to stalls etc.).

Captured attributes:

- stage asked for input;
- stage provided its output to next stage;
- stage stalled;
- stage has nothing in its FIFO buffer.

Baseline statistics for Decode are presented below.

Table 5.13: Power numbers for Decode stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock network</td>
<td>46.34</td>
<td>$E_{pk} = 2.20pJ/op$</td>
<td>$E = 1.01pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>10.79</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>42.87</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The clock network is, again, the main consumer, but there is no longer a sequential component to power dissipation (this stage is purely combinational, aside from its FIFO buffer). Overall, this stage also retains some similarity to a buffer, because such a buffer represents a significant part of it.
That is why this model is also, to some extent, shaped after that of a FIFO buffer.

Table 5.14: Model for Decode stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode</td>
<td>E=1.318<em>1-0.001</em>output-0.106<em>input-0.658</em>stalled-0.198*flushed</td>
</tr>
</tbody>
</table>

Notations used in table 5.14 signify: output - RegRead asked for new output from Decode; input - Decode asked for new input from the Fetch; stalled - Decode was not asked to provide new output and it did not ask new input; flushed - this stage’s FIFO buffer is empty.

5.3.3 Register Read

The logic I am modelling here does not include the register file, which I modelled separately. It is only the logic coordinating read/write operations from/to the register file; it also contains the load FIFO, and some logic to coordinate the PC Manager (counter that keeps a track of instruction addresses). Unit stimuli:

- send random values for fields of bits that do not affect RegRead function (e.g. the immediate field);

- use random control signals from Execute and Decode, in a manner similar with in-pipeline behaviour;

- make sure that the values for control sequence (bits generated by Decode) are random, but not impossible to find during pipeline operation.
Captured attributes:

- stage asked for input;
- stage provided its output to next stage;
- stage stalled;
- FIFO buffer of stage is empty;
- has processed a fetch (jump) instruction;
- attempted to take data provided by the data cache (through its load FIFO).

Baseline statistics for RegRead are presented below.

Table 5.15: Power numbers for Register Read stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock network</td>
<td>47.80</td>
<td>$E_{pk} = 1.12 \text{pJ/op}$</td>
<td>$E = 0.64 \text{pJ/op}$</td>
</tr>
<tr>
<td>Register Read</td>
<td>register</td>
<td>27.44</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>24.76</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The clock network is, again, the main consumer, but there is no longer a sequential component to power dissipation (despite the fact that this stage is not purely combinational). Overall, the stage retains only reduced similarity to its buffer, because the buffer is not the dominant logic (it is not insignificant, though). As a result, some new variables appear in this model.

Notations used in table 5.16 signify: output - Execute asked for new output from RegRead; input - RegRead asked for new input from the Decode; stalled - RegRead was not asked to provide new output and it did not ask new input; fetch - fetch instruction currently in stage; load - stage attempts to take value from load FIFO.
Table 5.16: Model for Register Read stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>E = 2.689<em>1^-0.250</em>output - 0.300<em>input - 1.350</em>stalled - 0.047<em>fetch - 0.073</em>load</td>
<td></td>
</tr>
<tr>
<td>ε = 0.64</td>
<td></td>
</tr>
</tbody>
</table>

5.3.4 Execute

This stage is highly heterogeneous, which made modelling it very difficult. Instead, I chose to approximate it with a combination of four units:

1. ALU;
2. Multiplier;
3. Forwarding logic;
4. Adequately sized FIFO buffer.

Models for all of these have been described previously.

5.3.5 Write Back

The last stage, Write Back (WB) is purely combinational, and constitutes a thin layer of logic between the FIFO buffer in Execute and (i) the register file, (ii) data cache.

Unit stimuli:
• send random values for contents to be written to the data cache or registers, and for destination addresses;

• use random values for control sequence, making sure that there are enough writes to the register file and enough memory operations.

Captured attributes:

• total Hamming distance between successive inputs - since this stage is purely combinational, Hamming distances offer a good measure of power.

Baseline statistics for Write Back are presented below.

Table 5.17: Power numbers for Write Back stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Power consumed by ...</th>
<th>%</th>
<th>Peak energy</th>
<th>Avg. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Back</td>
<td>clock network</td>
<td>0.0</td>
<td>$E_{pk} = 0.022pJ/op$</td>
<td>$E = 0.008pJ/op$</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>combinational</td>
<td>99.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sequential</td>
<td>0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is obvious from these results that WB logic is relatively insignificant for total pipeline power dissipation. This is true especially if compared with large consumers, like the register file or the multiplier.

Table 5.18: Model for Write Back stage

<table>
<thead>
<tr>
<th>Module name</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Back</td>
<td>$E = 0.008*1-$</td>
</tr>
<tr>
<td></td>
<td>$0.000009*hd-$</td>
</tr>
<tr>
<td></td>
<td>$\varepsilon = 0.005$</td>
</tr>
</tbody>
</table>

Parameter $hd$ used in the model signifies total Hamming distance between successive inputs (a sum of Hamming distances between successive values to be written to the register file, between successive cache addresses and between successive register addresses).
5.4 Model accuracy evaluation

For evaluation I chose 10 of the models previously described:

1. Fetch
2. Decode
3. Register Read
4. ALU + Multiplier + Forwarding logic + Execute FIFO buffer
5. Forwarding logic (second instance)
6. Write Back
7. Register file

To evaluate these models, I

1. ran an RTL simulation with instruction traces of the six programs presented in section 5.1.2;
2. captured all necessary attributes (as required by the collective of models);
3. used models and captured attributes to estimate pipeline energy consumption;
4. ran a gate-level simulation;
5. used Synopsys PrimeTime to get power numbers;
6. converted these power numbers to energy.
The metric I use for evaluation is *average normalized error*. For two sets of values - real \((r_n)_{n \geq 1}\), and predicted \((p_n)_{n \geq 1}\) -, the average normalized error is computed using:

\[
\bar{\varepsilon} = \frac{1}{N} \sum_{i=1}^{N} \frac{|r_i - p_i|}{r_i}
\]  

(5.5)

I also plot the distribution of average-centred errors, for each benchmark trace. Table 5.19 and figures 5.3-5.8 present the results.

Table 5.19: Average normalized errors of energy estimation

<table>
<thead>
<tr>
<th>Trace</th>
<th>(\bar{\varepsilon}[^%])</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort</td>
<td>37.03</td>
</tr>
<tr>
<td>CRC</td>
<td>29.67</td>
</tr>
<tr>
<td>bitcnts</td>
<td>42.01</td>
</tr>
<tr>
<td>search</td>
<td>29.02</td>
</tr>
<tr>
<td>basicmath</td>
<td>36.77</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>36.63</td>
</tr>
</tbody>
</table>

5.5 Summary

This chapter has presented a methodology for modelling power. The average energy prediction error of models obtained with this method across the benchmark was

\[
\bar{\varepsilon} = 35.18[^\%]
\]

By employing this unique number as my criterion for method efficiency evaluation, I conclude that:

- the method is sensible - its estimations are correct to an order of magnitude;

- more work is required to (i) refine models of individual units, (ii) identify causes of errors. See next chapter for discussion.
Figure 5.3: Distribution of average-centred errors for Qsort

Figure 5.4: Distribution of average-centred errors for CRC

Figure 5.5: Distribution of average-centred errors for Bitcnts
Figure 5.6: Distribution of average-centred errors for String Search

Figure 5.7: Distribution of average-centred errors for Basicmath

Figure 5.8: Distribution of average-centred errors for Dijkstra
Chapter 6

Summary and Conclusions

In this chapter I offer a general summary of undertaken work and point out results, successes, failures and possible future work.

This dissertation represents the summary of work that took me several months to complete and that involved:

- design of a RISC pipeline to implement the *Loki* ISA;
- description (bottom-up) of pipeline using System Verilog;
- verification of individual units inside the pipeline;
- verification of pipeline on benchmark instruction traces;
- model derivation for individual units;
- model derivation for pipeline stages;
- evaluation of energy models prediction efficiency.
My research group will benefit from results I obtained because:

- I created a test infrastructure that is capable of running instruction traces from real programs; it can be extended to implement the extended *Loki* ISA, and it *must* be upgraded to full functionality in regard to system calls (heavily used in simulations).

- baseline models I propose could be a starting point or a reference for someone working in a related project;

- by applying my methodology of creating models for units using constrained random stimuli, and by evaluating its performance, I proved that this is worth pursuing further (in spite of it not providing the best results right now).

As previous chapter concluded, models are sensible, in the sense that their estimations are in line with other reports on similar techniques. Of course, errors such as the one I get (35%) were expected from the start, and the intention was to identify where their source lay. I will conclude this chapter with a brief discussion of investigations that might lead to better models in the future.

1. Model refining. This would require applying the same mechanism I used, only with an even higher degree of attention to details, and employing some new tools. For instance, new attributes could be tracked and checked for correlation with power dissipation. A new way of constructing testbenches could also yield some results - for every unit that I found to have considerable percentage of power dissipated in registers, the testbench could include *spacer cycles* (quite useful for separating reading and writing energy, to find correlation more easily).

2. One further step could be taken over my baseline models, with a *machine learning* approach. For the particular case of my benchmark, one
could use an instruction trace taken from one of the six programs to *train* a model, then validate it on the remaining instruction traces.

3. At one point, model refinement will be deemed to no longer be capable of achieving significant gains in energy estimation efficiency. I appreciate that a sizeable error will still exist, even with refined models. This is when completely different sources of error must be investigated. (i) One of them is pieces of logic I have not modelled (for instance, the PC manager, which is a typical counter with synchronous reset and load value, and is attached to a small FIFO buffer). (ii) Wiring is another. In my unit implementations, I attempted to account for wiring when setting timing constraints. However, wires require proper modelling, considering how much of a problem they have become for new generation CMOS processes (see chapter *Background*). (iii) Considering how important a source of power dissipation the clock tree is, a model describing it should also be investigated.

4. Finally, a designer always has the option of going to the fundamentals. This entails dealing with lower levels - as low as layout - to determine how relative placement of cells affects energy dissipation or to what extent good layout rules are respected by the tool that does the automatic placement.

All in all, it seems there is still a lot of work to do in this area. Unfortunately, we are running out of time - very soon, manycores will hit the *utilization wall* (see intro). Without reliable tools (such as high-level cycle-accurate simulators) to guide designers, it will become difficult to maintain even the present rate of evolution, which is not even closely as high as it used to be during the *golden age* of silicon scaling.

Hopefully, the high level of interest the area of low power computing enjoys from industry will help speed up the process of finding better solutions to exploit ever shrinking CMOS technologies. One can only hope that we will
succeed at this for the next decade, before a new technology becomes mature, and we can adopt the graphene transistor.
Bibliography


106, April 1998.


