Technical Report

Learning-Based Architecture-Level Power Modeling of CPUs

Ajay Krishna Ananda Kumar and Andreas Gerstlauer

UT-CERC-20-01

June 04, 2020

Computer Engineering Research Center
Department of Electrical & Computer Engineering
The University of Texas at Austin

2501 Speedway, Stop C8800
Austin, Texas 78712

http://www.cerc.utexas.edu

The University of Texas at Austin
Electrical and Computer Engineering
Cockrell School of Engineering

Learning-Based Architecture-level Power Modeling of CPUs

Ajay Krishna Ananda Kumar and Andreas Gerstlauer
Department of Electrical & Computer Engineering,
The University of Texas at Austin.

With the end of Dennard scaling, energy efficiency has become an important metric driving future processor architectures, particularly in the fields of mobile and embedded devices. To support rapid, power-aware design space exploration, it is important to accurately quantify the power consumption of the different micro-architectural components of processors early in the design flow and at a high level of abstraction. Existing CPU power models rely on either generic analytical power models or simple regression-based techniques that suffer from large inaccuracies. Recently proposed machine learning techniques for accurate power modeling still require slow RTL simulations or have only been demonstrated for fixed-function accelerators at higher levels.

In this report, we present a novel approach that uses machine-learning to model cycle-accurate power consumption of programmable processors and their internal structures at a high micro-architectural level. Using only high-level information that can be obtained from micro-architecture simulations, we extract representative features and develop low-complexity learning formulations for different micro-architecture components that require a small number of gate-level simulations for training. We further present a hierarchical power

model composition methodology to build power models for complete CPUs. Our composed models provide cycle-accurate and hierarchical power estimates down to sub-block granularity.

We demonstrate the generality of our approach by modeling a simple in-order core as well as a complex superscalar out-of-order core. Results show that our hierarchically composed models for two RISC-V processor cores, RI5CY core from the PULP platform and Berkeley Out-of-Order Machine (BOOM), predicts cycle-by-cycle power consumption to within 2.2% and within 2.9% of a gate-level power estimation on average, respectively. In addition, our power model for the BOOM core, trained on micro-benchmarks, has an error rate of less than 3.6% when predicting cycle-by-cycle power consumption of a real-world application.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the breakdown of Dennard scaling, power consumption, especially that of processors, is a first-order concern in all modern chips. Accurately quantifying the power consumption through power analysis in early design stages is crucial for power-aware hardware and processor design.

Accurate power estimation relies on gate-level analysis, which comes at the cost of long simulation times and availability only in very late phases of the design flow. At the register-transfer level (RTL), industry tools such as PowerArtist and PowerPro can provide accurate aggregate power estimates sufficient to highlight coarse-grain RTL power saving opportunities. Regression-based approaches [1], [2], [3] support building power models at a finer granularity, but at the expense of decreased accuracy. More recently, advanced machine learning (ML) approaches using deep neural networks (DNNs) have demonstrated the capability for highly accurate RTL power estimation [4]. However, deep learning requires a large amount of training data to be obtained from gate-level reference simulations. Furthermore, the need for slow RTL simulations limits the usefulness and extent of design space exploration that is possible with any RTL power estimation.

Early design space exploration of CPUs is most commonly performed at an abstract micro-architecture level. Traditionally, spreadsheet-based or generic analytical power models [5] are used to provide power estimates at this level. However, such models have been shown to be highly inaccurate [6]. Regression methods have also been applied instead to model power at higher instruction and micro-architecture levels [7], [8], but they often similarly suffer from larger inaccuracies due to the challenge of modeling the non-linear power characteristics of the underlying circuits accurately at such high levels of abstraction. Advances in machine learning have made it possible to accurately capture such complex relationships. At the same time, training and inference costs should not negate the speed benefits of working at a higher abstraction level. This rules out expensive deep learning approaches. Instead, dedicated learning formulations that can achieve high accuracy with low complexity need to be developed. Such approaches have recently been provided for fixed-function accelerators [9], but they have not yet been applied to model programmable processors above RTL.

The goal of this report is to explore using simple machine learning algorithms and functionality dependent feature engineering techniques to develop low complexity power models of CPUs that can provide highly accurate power estimates while working at a high-level of abstraction.

1.1 Report Overview

In this report, we present a methodology to develop machine learning-based micro-architecture level power models for CPUs that provides accurate cycle-by-cycle power estimates. Using high-level activity information available from micro-architecture simulations, we extract features and develop learning formulations that can capture correlations with minimal training overhead and complexity. Our models are trained on gate-level simulations of small instruction sequences. Trained models can then provide highly accurate cycle-by-cycle power estimates in a hierarchical fashion at the complete core level and down to different CPU sub-blocks.

1.2 Contributions

The specific contributions of this report are as follows:

- We present a methodology for feature selection and engineering to model common in-order and out-of-order CPU structures at micro-architectural level.
- We explore advanced non-linear regressors for power modeling of different micro-architectural blocks in CPUs with low training overhead and high accuracy.
- We propose a hierarchical model composition approach that supports power modeling down to sub-block granularity while accounting for glue logic in super-block power modeling.

- We demonstrate our power modeling approach on RI5CY, an in-order RISC-V core and BOOM, a superscalar Out-of-Order (OoO) RISC-V core. We identify key representative features for modeling of common CPU blocks with high predictability and low complexity. Our hierarchically composed power models for RI5CY and BOOM cores have an average error rate of 2.2% and 2.9% respectively.

1.3 Report Outline

The remainder of this report is organized as follows: Chapter 2 first highlights related work and Chapter 3 then gives an overview of our power modeling flow. Chapter 4 provides details of our CPU power modeling approach. Case studies and results are presented in Chapters 5 and 6. Finally, Chapter 7 concludes the report and proposes directions of future research.

Chapter 2

Related Work

In this chapter, we briefly review prior works in the field of power modeling/analysis. Figure 2.1 categorizes key existing power modeling works at different level of abstractions and shows their accuracy range with coloring, where a darker color corresponds to an increase in accuracy. In the following sections, we will further review and contrast the works in each category.

2.1 Analytical and Library-Based Power Models

Instead of slow SPICE simulations for power analysis, common practice to get highly accurate power estimates for a design is to use power-characterized standard cell libraries with commercial tools such as Synopsys PrimeTime PX [10] by feeding in gate-level netlist and simulation results. Even though they are very accurate and can provide fine-grain power estimates, the requirement for a netlist necessitates that such analysis occurs late in the design process, where slow gate-level simulations hamper micro-architectural exploration. Tools such as PowerArtist [11] and PowerPro [12] have been developed to speed-up the process of providing power feedback and can provide a course grain power estimate at the register transfer level (RTL).

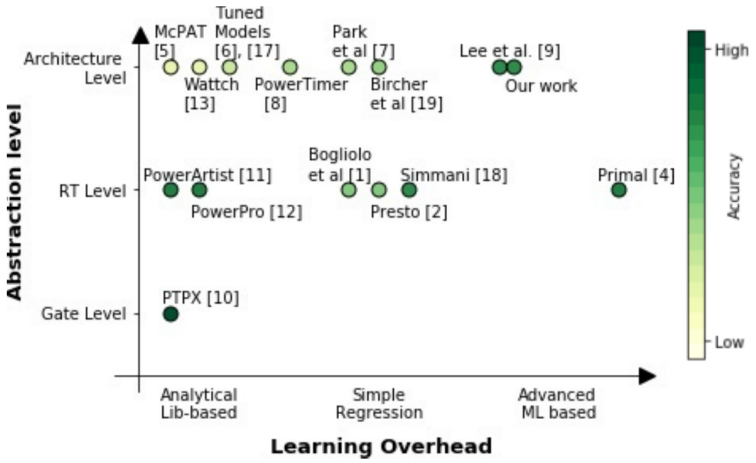


Figure 2.1: Power modeling/analysis approaches.

However, early design space exploration of CPUs typically happens at an abstract micro-architecture level. Analytical power models [5], [13], [14] coupled with micro-architectural software simulators such as gem5 [15] are generally used during this exploration phase. Tools such as McPAT [5] analytically model the power at the physical technology level, using physical parameters of the devices and wires, and then map the sub-blocks to commonly used circuit structures and underlying physical technology models. Such approaches are generic, do not map well to one specific processor implementation. As such, they suffer from large inaccuracies if not carefully tuned to the evaluated processor [16]. It is possible to calibrate analytical models against low-level measurements [6], [17], but the parameter fitting will limit interpretability at the sub-block level.

2.2 Statistical & Regression-Based Models

Regression-based RTL models [1], [2], [3], [18] propose various approaches for selecting critical signals and registers strongly correlated with power and build simple regression models trained from gate-level power analysis results. These approaches typically deploy simple linear models, which do not capture the non-linear relationship in complex designs [4] and hence are limited in accuracy.

Regression-based approaches at the architecture level rely on simulating an implementation, sampling and fitting generic regression equations for modeling CPU power at the pipeline or instruction level [7]. Methodologies based on correlating performance counters to power by simple curve fitting is another common power modeling approach at the architecture level [19], [20]. Other works [8] combine analytical approaches with regression equations formulated using pre-characterized power data from existing designs. However, all of these simple models still suffer from inaccuracies in modeling data-dependent, cycle-by-cycle power of a processor at fine sub-block granularity.

2.3 Advanced Machine-Learning Based Power Models

Several advanced ML-based approaches for power modeling have recently been explored to capture the non-linear power-feature correlation. PRIMAL [4] uses a convolutional neural network (CNN) for modeling RTL power trained from gate-level simulations using the combined activity of all registers in a design. Such a CNN-based model is very accurate but requires a large

amount of training samples and training time compared to simple regression models. The proposed model also relies on details available only at RTL or lower levels of abstraction during late design stages, and thus is not ideal for early design space exploration.

Recent work [9] has shown the possibility of building ML-based power models at a higher, C++/ SystemC level of abstraction. The work proposes several feature selection and model decomposition techniques to enable highly accurate prediction using low-complexity non-linear regressors. However, it has only been demonstrated for fixed-function accelerator IPs. Our proposed approach is motivated by [9] and aimed at modeling programmable CPUs by adopting similar supervised learning based regression methods at the CPU sub-block level, and then hierarchically composing such models.

Chapter 3

Power Modeling Methodology

This chapter describes the methodology used for developing and validating our architecture-level power models for CPUs.

3.1 Power Modeling Flow

Fig 3.1 shows an overview of our power modeling flow. The primary inputs are the gate level netlist and a cycle-accurate model of a processor. In this work, we generate a cycle-accurate C++ model from the RTL description of the processor using the Verilator tool [21]. However, our approach only requires high-level activity information, and the Verilator model can be easily replaced with a high-level cycle-accurate, micro-architecture simulation model. During the training phase, simulations are run at both gate and cycle-accurate levels using the same micro-benchmarks. Cycle-by-cycle per block reference power traces are generated using industry-standard gate-level simulations, and activity traces are extracted from the cycle-accurate model simulation. In the power model synthesis step, we extract features for the different functional blocks and apply feature selection and decomposition techniques by analyzing the functionality and attributes of the blocks. Using extracted features

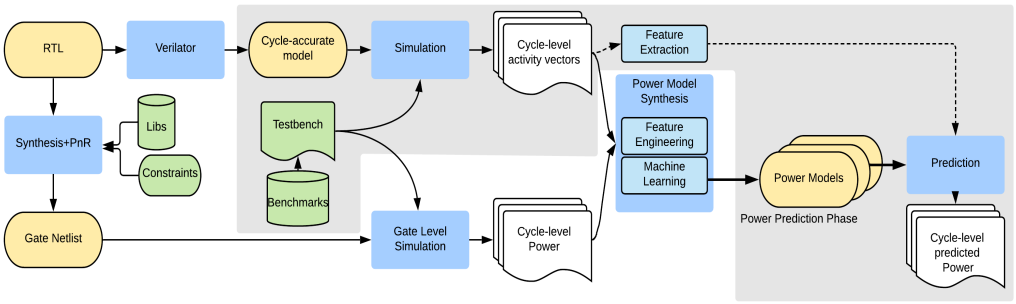


Figure 3.1: Power modeling flow.

and reference power values, a ML regressor is trained to learn the correlation between the decomposed features per block and the power consumed by that block across cycles. These learned models are then stored to be used during the prediction phase.

During the prediction phase, the full workload to analyze is simulated in the cycle-accurate model. Feature extraction and decomposition is applied to the activity information extracted from the simulation and previously trained models are used to predict cycle-by-cycle power per block hierarchically up to the full core level. Hierarchically decomposed power models down to the sub-block level thereby enable micro-architecture level exploration as pre-trained blocks can be arranged in different compositions and only blocks that are modified need to be re-trained or analytically scaled.

3.2 Model Validation

In order to evaluate the accuracy of our power models, we compare the predicted power against cycle-by-cycle power traces obtained using a commercial gate-level power estimation tool, which we take to be the ground truth for evaluation. We use cycle-by-cycle mean absolute error (MAE) of values predicted by each model compared to gate-level power estimation, normalized to mean reference power of the block as our evaluation metric, given by the below equation:

$$MAE\ [\%] = \frac{\frac{1}{n}\sum_{i=1}^{n}|P_{pred}(i) - P_{ref}(i)|}{\frac{1}{n}\sum_{i=1}^{n}P_{ref}(i)} \times 100 \tag{3.1}$$

,where $P_{pred}(i)$ is the predicted power in the i^{th} cycle, and $P_{ref}(i)$ is the reference power estimated by PrimeTime PX in the same cycle.

To evaluate the accuracy of our model in predicting the average power of the CPUs, we compute a difference of the measured and predicted power consumption over the whole simulation. The difference is normalized against measured average power of the block using the following equation to compute average error (AE):

$$AE\ [\%] = \left|\frac{\frac{1}{n}\sum_{i=1}^{n}P_{pred}(i) - P_{ref}(i)}{\frac{1}{n}\sum_{i=1}^{n}P_{ref}(i)}\right| \times 100 \tag{3.2}$$

The limitations of our model are evaluated using a maximum error (ME) metric, which captures the error in the cycles where the predicted power deviates maximally from the measured power.

$$ME\ [\%] = \frac{\max_i |P_{pred}(i) - P_{ref}(i)|}{\frac{1}{n}\sum_{i=1}^{n}P_{ref}(i)} \times 100 \tag{3.3}$$

Chapter 4

Power Modeling Approach

The effectiveness of supervised learning approaches depends on engineering features that are highly correlated with the values to be predicted as well as on selection of appropriate learning models that can capture underlying correlations with low overhead. Power consumption of a circuit specifically is sensitive to certain key contributor signals [1], [2]. ML-based hierarchical power modeling of CPUs thus involves the following steps: (i) identification of key contributing activity information, (ii) mapping of key contributing signals to features and feature engineering, (iii) model selection for each block, and (iv) super-block power model composition. This chapter presents our approach for feature and model selection of common attributes and structures found in the CPU blocks as well the handling of super-blocks for power modeling.

We limit feature selection to activity information that can be extracted from cycle-accurate, micro-architectural performance models, such as MARSS [22], SimpleScalar [23] or gem5 [15] augmented to trace the data activity. We evaluate the following linear as well as non-linear ML regressors to model the power consumption of different blocks in the CPU: (i) least-squares linear regression (LR), (ii) linear regression with l2-norm regularization (LR-R), (iii)

linear regression with L1 prior regularization (LR-L), (iv) a linear model with l2 regularization where the priors over the hyperparameters are chosen to be gamma distributions (LR-B), (v) a decision tree based regressor (DT), (vi) a gradient boosting model of equivalent complexity with a regression tree fitted on the negative gradient of the loss function in each stage (GB), and (vii) a random-forest with number of estimators fitted to match the decision-tree on its complexity (RF). We compare our ML-based models against a model predicting average power across the training set (Avg).

4.1 Block-Level Modeling

Components of a CPU have certain micro-architecturally visible characteristics and features that determine the total power consumption as well as the variance in power from the previous cycle. The performance of a power model depends on how accurately it captures these underlying characteristics or attributes. This section categorizes the basic set of attributes that different types of CPU blocks possess and provides generic guidelines for modeling those attributes with high accuracy.

4.1.1 Data Processing Blocks

The most common types of CPU blocks have circuit structures that perform similar operations every cycle on multi-bit data signals. Intuitively, the power consumption strongly depends on the activity of the data they process. Hamming distance (HD) has been widely used as a feature to concisely

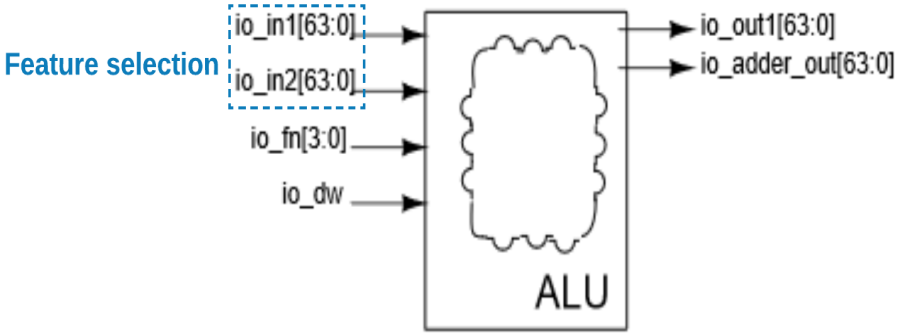


Figure 4.1: ALU sub-block with signals selected for prediction.

capture such data activity. At the same time, hamming distance of the entire multi-bit data has weak correlation to power. This is because of the difference in the circuit components that each toggling bit can effectively activate. For most of the commonly used datapath components, bits far off spatially (LSBs vs. MSBs) differ significantly, while those closer together (e.g. bits 0 and 1) show similar power behavior as a function of toggling activity. Based on this observation, the multi-bit data can be decomposed into smaller contiguous bit-groups and the hamming distance of these bit-groups can be separately captured to obtain features with strong correlation to power consumed. The optimal granularity of this decomposition needs further exploration.

Figure 4.1 shows a basic block diagram for the ALU sub-block in the RISC-V core used in our experiments (see Chapter 6 for details). We select the input operands as the only signals used for prediction and traced in the cycle-accurate simulations. Figure 4.2 summarizes the mean absolute prediction error (MAE) of using the hamming distance of varying bit-widths of operands

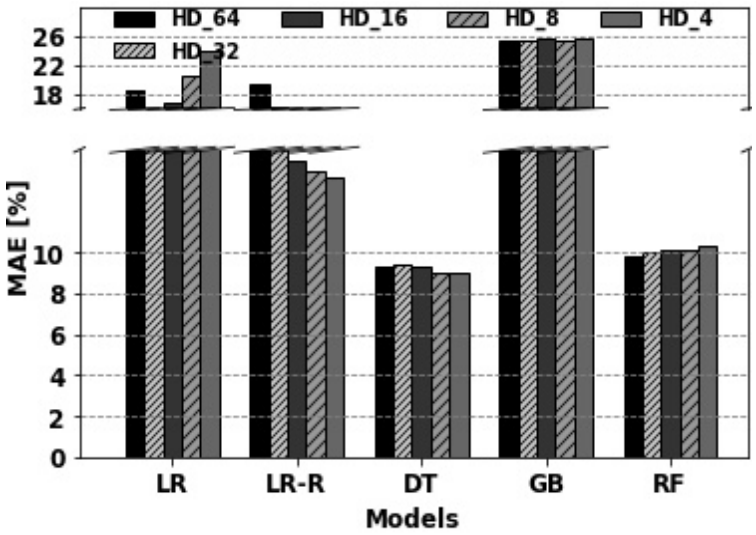


Figure 4.2: ALU feature and model selection.

(HD_64, HD_32, HD_16, HD_8, HD_4) for data inputs across different learning models at ALU block level. As results show, a decision tree based model performs better in accuracy than other models and byte-wise hamming computation provides power models with good accuracy (improves accuracy by an additional 0.29% in this case). We also observe that DT starts overfitting for nibble-wise decomposition and is slightly worse than byte-wise decomposition. Based on this analysis, we propose to model the data processing blocks with byte-wise decomposition of input signals, which provides good accuracy while still retaining the simplicity of using hamming distance of byte groups in place of single bit switching traces as features and avoiding overfitting.

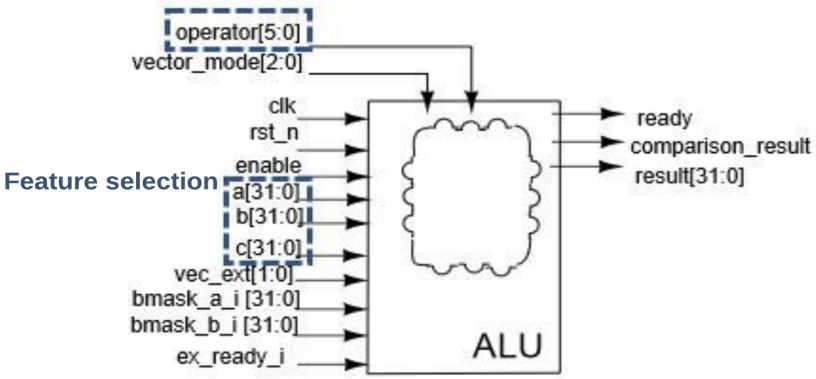


Figure 4.3: ALU block feature selection.

4.1.2 Control Modes

Multiple CPU blocks possess a key characteristic of having multiple modes with significantly different operations depending on a control input or a state available in the performance model. Each mode typically activates different portions of the circuit and hence can consume a very different amount of energy. It is important to capture which specific portion of the circuit is active or idle in each cycle to develop accurate power models. Also, mode switches can cause large power variances at the cycle-by-cycle level. For example, a shift from normal read to update mode of a branch predictor would cause significant power deviation.

We propose to model the absolute power consumption by using the current value of the control input and the power variance by using the hamming distance of the control input as features for our underlying ML models. Figure 4.3 shows a basic block diagram for the ALU block in the RISC-V core (see

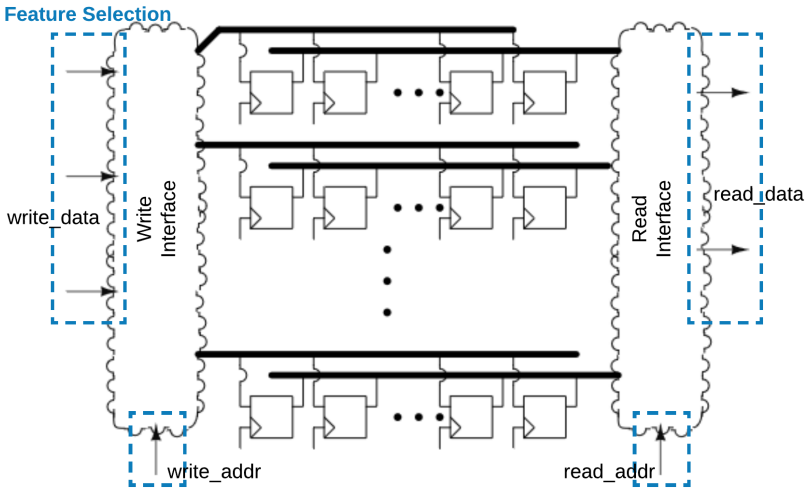


Figure 4.4: Buffer modeling.

Chapter 5 for details) and the ALU operands as well as the ALU operating mode input being used as features for power modeling.

4.1.3 Buffering

Another common attribute of CPU blocks is having the capability to store data, meta-data and control information such as micro-ops in buffers with possibly multiple readers and writers. The majority of the sequential component's clock power has very low variance at cycle-level granularity and can be very easily modeled as a constant bias term in regression models. The variance in power consumption in these blocks is dominated by the switching of muxes and routing logic driven by the data that is being written/read in the current cycle. Based on this rationale, the read and write data and addresses are selected as signals to model the buffering property as shown in Figure 4.4.

Notably, such a buffering attribute is common in many superscalar out-of-order CPU blocks, ranging from a branch target buffer to re-order buffer. Even though the underlying sequential buffering logic is similar (varying only in bit width and partition), these blocks have widely different read and write characteristics - from having indexed read/write to fully associative read/write, from partial data being processed on read to full data being used. Also, the availability of corresponding signal features in high-level simulations varies among blocks. For instance, addresses to a physical register file, that buffers the operand data, can differ between high-level model and implementation due to equivalent but different renaming implementations. Although an ideal approach is to use all the inputs (data and addresses) as features, we limit ourselves to features that are only available in a typical high level simulation.

4.1.4 Pipelining

CPU blocks can either be pipelined or un-pipelined, where the number of pipeline stages is typically already decided at the micro-architecture level for a majority of the blocks. Intuitively, the power consumption of such pipelined structures does not only depend on the current input to the blocks but also on the partially processed input stored in internal pipeline registers. However, a high-level simulation model will typically not model the entire pipeline of a sub-block accurately, and thus might not contain enough information for developing a stage-by-stage power model. To model these structures with available information, we can instead store and use the *history* of the last N

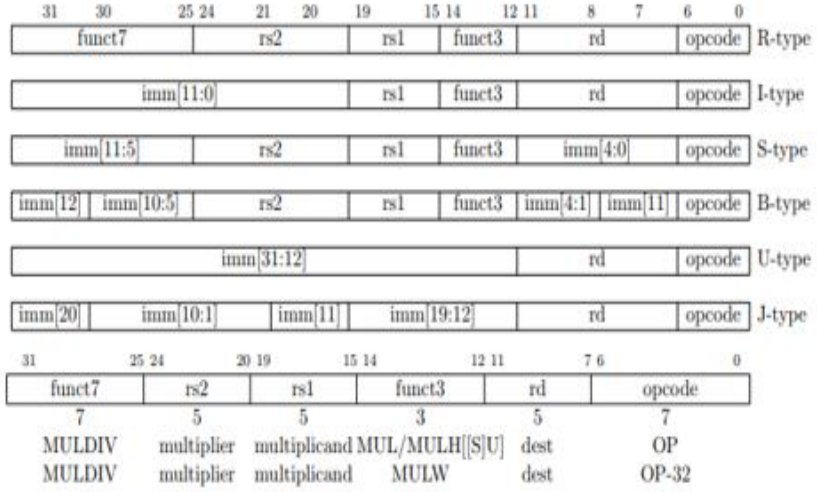


Figure 4.5: Instruction word bitfields.

inputs as features [9]. The activity of internal pipeline registers and hence the power of internal pipeline stages will be correlated to the previous primary inputs of a block depending on the depth of the pipeline. By giving the input history as features, the pipeline behavior as well as the impact on power consumption can be learned together.

4.1.5 Mixed-Attribute Blocks

Most of the common CPU blocks possess one or more of the above mentioned attributes in an independent fashion, and power models can be developed by orthogonally modeling the associated features. However, a few CPU blocks possess mixed characteristics for the same input or output signals and hence need special handling. For example, for an instruction decoder, the instruction word affects both the mode as well is the data that the de-

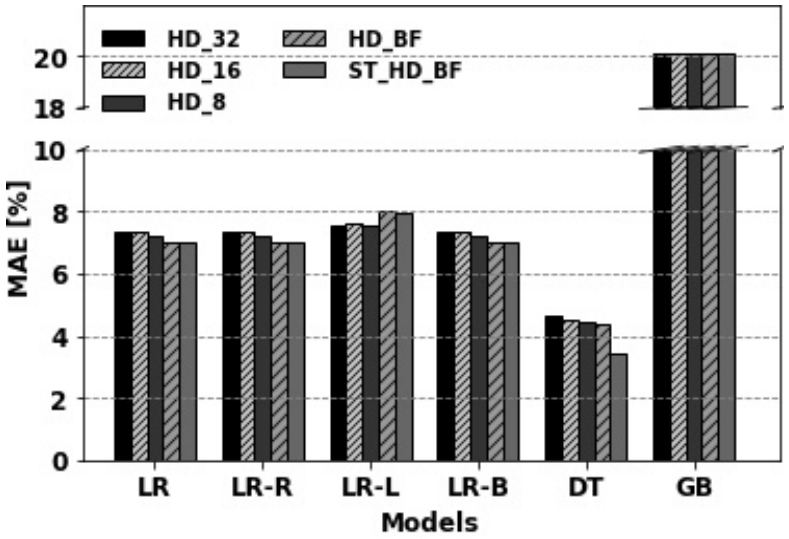


Figure 4.6: Instruction decode stage feature and model selection.

code stage processes in each cycle. Figure 4.5 shows the different bitfields in an instruction of the RISC-V IM instruction set. Rather than a generic byte-wise decomposition, the instruction word is sliced based on the sub-field boundaries in the instruction format to allow the model to learn the relation of each sub-fields with power. Figure 4.6 shows the error trend across different learning models with different feature decomposition techniques: HD_32 (hamming distance of the entire instruction word), HD_16 (hamming distance of half-words), HD_8 (hamming distance per byte),HD_BF (hamming distance per bitfield), ST+HD_BF (current value and hamming distance per bitfield). Again, a decision tree model provides the best accuracy, where feeding both the current value and hamming distance per bit field into the model provides between 0.9% and 1.2% better accuracy than other decompositions.

4.2 Gating Models

Gating is an RTL/implementation-level technique to prevent unnecessary toggles reducing the total energy consumed. Though efficient as a power saving technique, it significantly affects the ability of high-level models to capture the underlying circuit characteristic. In this section, we propose approaches to capture these circuit level optimizations at a high level with reasonable accuracy as a special case of our control attribute modeling.

4.2.1 Data Gating

Data gating is a common technique used to prevent unnecessary circuits from toggling when the output is not needed. Typically, instead of mux-ing the outputs, the enable signals of the input latches are disabled such that whole sub-blocks will not toggle. Common examples include splitting the input paths of different execution modules in a multi-function ALU and gating specific paths based on the incoming micro-op as shown in Fig 4.7. The power consumption of such gated circuit is thus dependent on the toggling of the gated or control qualified signals rather than only the primary inputs. We use the history of data applied to a data gated path and calculate hamming distance per path based on the previous actively applied data input rather than on the value in the previous cycle. The feature decomposition is still based on the attributes that we associate the block with.

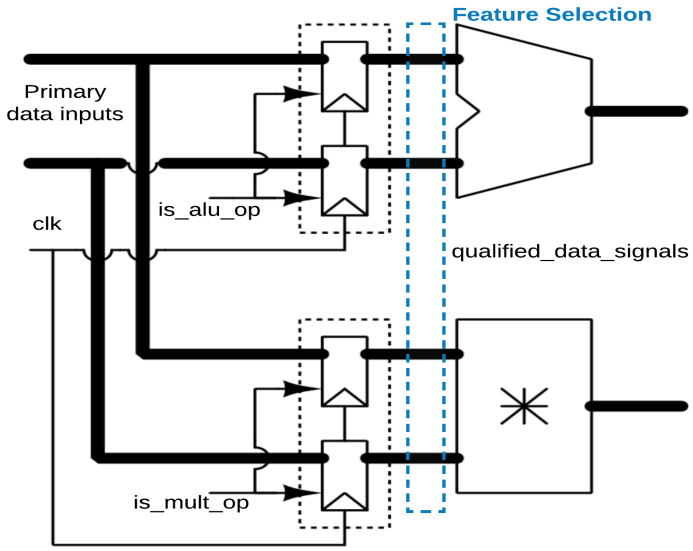


Figure 4.7: Data gating modeling.

4.2.2 Clock Gating

Although data gating can be implemented by gating the clock that feeds the input data latches, clock gating [24] is a general low-level circuit design technique that can be automatically applied by synthesis tools to save clock tree power. Different clock gaters with complicated conditions, as shown in Fig 4.8 increase the variance in power consumption and thus make it much more difficult to model power at a high level. The degree of power variance due to the clock being switched between gated and ungated state depends on its load. We propose the following approach to model combinational clock gating, commonly inserted on logic synthesis, on heavily populated buffer blocks (blocks that possess only buffering attribute). These buffer blocks typically have certain micro-architecturally visible conditions that determine usage of

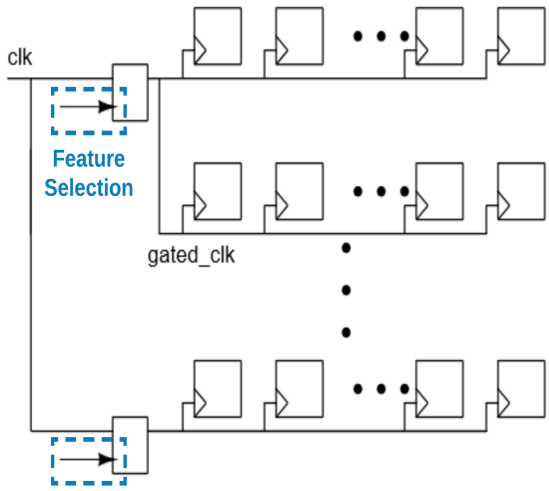


Figure 4.8: Clock gating modeling.

some segment or the whole of the buffer. For example, a branch snapshot buffer might only be needed when there is an incoming branch mapped to the specific branch tag. Such conditions can be formulated as features for modeling these clock gated buffer units. We recognize the complexity in modeling a sequential clock gating circuit and we leave it for future work.

4.3 Hierarchical Model Composition

To handle super-blocks in hierarchical power modeling, there are two possible approaches: (i) synthesize a separate power model for the super-block, or (ii) compose a power model for the super-block from the component power models. Though the first approach can generate accurate power models with the right set of features, the second approach has the advantage of

reduced power model synthesis time and better architectural exploration support. However, the composition approach suffers from inaccuracies due to the additional glue logic that is present in the super-block not being modeled.

Such glue logic can be a significant contributor to total super-block power. From our gate-level analysis, glue logic at the core level can consume about 5% of the average total power in a very simple CPU and can contribute up to 10% on a cycle-by-cycle basis. Our approach to solving this problem is to treat the glue-logic as a virtual block and synthesize a power model for it. This is achieved by subtracting the sum of component powers from the total power during training to obtain the reference power for the glue logic block. During prediction, the glue logic block then forms a part of the composed super-block power model. Super-block power modeling and model composition with and without glue logic will be evaluated in Chapters 5 and 6.

Chapter 5

Scalar In-Order Core Results

In this chapter, we first evaluate our proposed approach by modeling the power consumption of a simple in-order RISC-V based processor [25]. Specifically, we model the open-source RI5CY core that is part of the PULP platform [26] developed at ETH Zurich and the University of Bologna. It is a 4-stage, in-order 32-bit RISC-V processor core. It fully implements the RV32IMFC ISA and many other PULP-specific extensions such as post incrementing load-store, MAC operations, and hardware loops. For this work, the floating-point module was not instantiated.

5.1 Experimental Setup

Figure 5.1 shows the major blocks in the core. The open-source RI5CY core RTL is synthesized with the Nangate 45nm PDK [28] using Synopsys Design compiler (L-2016.03-SP5)[29]. Seven benchmarks from the pulpino test suites listed in Table 5.1 are chosen for training and evaluation of the models. The benchmarks are compiled using the riscv-gnu-toolchain and object code is used for the simulations. Synopsys VCS [30] is used for the zero-delay gate-level simulation of the RI5CY core (@25MHz) with the chosen benchmarks

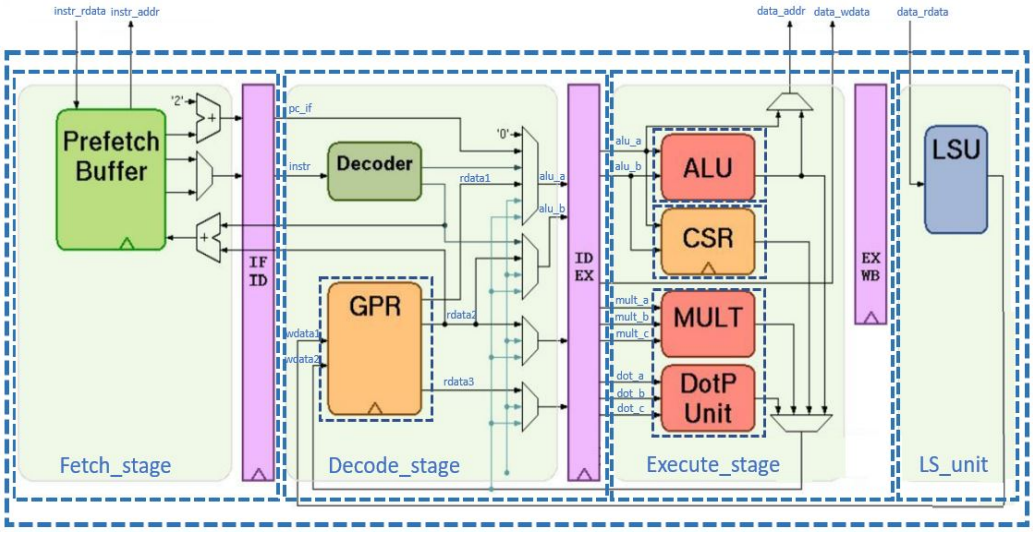


Figure 5.1: RI5CY core and its sub-block decomposition [27].

using provided inputs. Activity vectors at sub-block and per-cycle level are then extracted from the simulation dumps, and Primetime PX (PTPX) [10] is run in time-based power mode to generate cycle-by-cycle reference power numbers. Table 5.2 shows the gate counts and power statistics of the top level blocks when analyzed at the gate level using PTPX, while running all the seven benchmarks back to back.

The Scikit-learn Python package [31] is used for model synthesis and prediction.

5.2 Power Model

The hierarchical decomposition for power modeling purposes is highlighted with dotted boxes in Figure 5.1. For generality, the physical memory

Table 5.1: Benchmarks.

Test	**Description**	**Cycles**
aes_cbc	Small code version of AES implementation	77332
conv2d	2D convolution	17713
fdctfst	From ffmpeg libavcodec/jfdctfst.c	4863
fft	Fast fourier transform	112370
fir	10 Coefficient FIR filter	48757
keccak	Sha3 baseline implementation	607795
matmul	Matrix multiplication	660901

Table 5.2: Power statistics of RI5CY core blocks.

Block	**Cells (cb/sq)**	**Avg.**	**Std. dev.**	**Max**	**Min**
Fetch	3853 / 316	0.29mW	0.05mW	0.42mW	0.11mW
Decode	14561 / 1631	0.49mW	0.12mW	1.01mW	0.35mW
Execute	11248 / 123	0.21mW	0.05mW	0.58mW	0.17mW
LS_unit	1250 / 42	0.03mW	0.02mW	0.13mW	0.02mW
CSR	5481 / 846	0.25mW	0.01mW	0.31mW	0.18mW
PMP	12643 / 1	0.16mW	0.06mW	0.54mW	0.12mW
Core	49904 / 3050	1.48mW	0.27mW	2.63mW	0.99mW

protection (PMP) unit and control and status register (CSR) block with all the performance counters instantiated are also included for power modeling purposes. However, the CSR is not included as part of Execute_stage power model, but as a separate block. We have also evaluated our approach for hierarchical power modeling by further de-composing the execute stage into ALU and MULT sub-blocks.

The list of the major blocks and their mapping to the basic power modeling concepts introduced earlier is shown in the Table 5.3. The Fetch stage of the RI5CY core is comprised of a prefetch buffer that interfaces with the

Table 5.3: Power modeling features of different RI5CY blocks.

Block	**Data**	**Control**	**Data Gating**	**Clock Gating**	**Buffer**	**Pipeline**
Fetch_stage	✓				✓	
Decode_stage	✓	✓			✓	
Execute_stage	✓	✓				
LS_unit	✓					
CSR					✓	
Pmp_unit	✓					
Glue	✓					

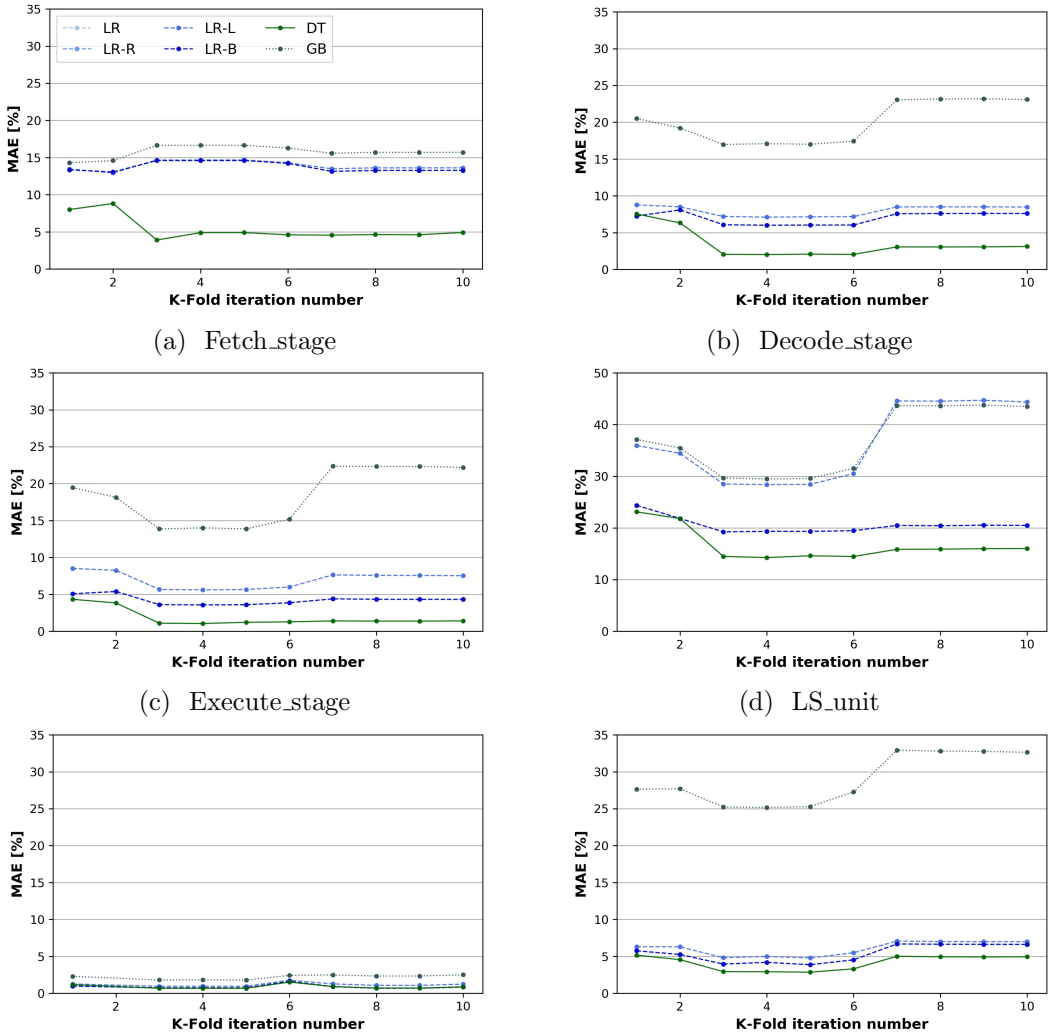
instruction cache and holds logic that treats instruction word and address as data. The Decode stage contains the instruction decoder (mixed attribute block as explained in Section 4.1.5), register file (buffering attribute) and register read logic, which handles the bypass data (data processing). Datapath elements - the ALU and Multiplier - are the main components of the execute stage. We employ byte-wise decomposition of the input data to model these components. The multiplier of the RI5CY core maintains an internal state for sub-word selection during the 4-cycle MULH (multiplication with upper word result) operation. This sub-word selection signal is control input for subsequent cycles of a multi-cycle operation and as such included as feature. The LSU unit handles the data interface between the execute stage and the cache, and as such can be modeled as a data processing block with both address and operand value as data. The CSR majorly contains the control and status registers and performance counters and is handled as a buffer block. The PMP unit protects the memory by performing range checks on the incoming data and instruction addresses, hence mapped to a data processing block.

5.3 Cross-Validation Results

10-fold cross-validation is used for the evaluation of the feature correlation and accuracy of the models on the cumulative data samples constructed from the 7 benchmarks. This resampling procedure nullifies the model bias on the data split and thereby allow us to evaluate the pure feature correlations to power.

Figure 5.2 summarizes the 10-fold cross-validations results for the 6 core-level sub-blocks. We can observe that the decision tree (DT) based model performs well compared to the linear models in all cases. As has been demonstrated in earlier work [9], decision tree-based data representations efficiently capture the inherent non-linear but typically discrete power behaviour of design blocks.

Gradient boosting based models are of similar complexity, but perform poorly in most of the cases; this shows the necessity of more depth to capture the non-linearity. As listed in Table 5.2, the CSR block has the least variance in power in the modeled core, which explains the high accuracy of the power models. On the other hand, low power but high variance blocks such as the LS_unit show poor accuracy in modeling with the evaluated models. Accurately modeling the power consumption of the blocks in this category needs more study and considered for future work. For this work, due to its small contribution to the total power of the core, a model with 16.6% error rate for the LS_unit is sufficient for gaining high accuracy for the core composed power model.



(a) Fetch_stage

(b) Decode_stage

(c) Execute_stage

(d) LS_unit

(e) CSR

(f) Pmp_unit

Figure 5.2: 10-fold cross-validation results.

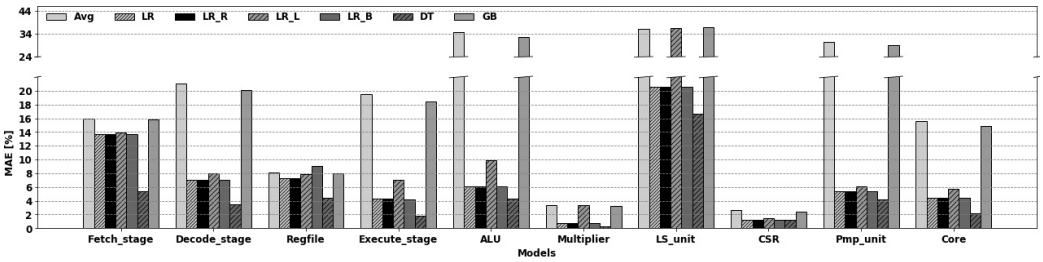


Figure 5.3: Average accuracy of block models.

Table 5.4: Top decision tree features for different blocks.

Block	**Features (Importances)**
Fetch_stage	HD(instr_addr) (0.32), instr_rdata (0.27), instr_addr (0.22), HD(instr_rdata) (0.19)
Decode_stage	HD(instr[24:20]) (0.70), HD(alu_a) (0.14), HD(instr[31:25]) (0.05), instr[11:7] (0.02), instr[24:20] (0.01), HD(instr[11:7]) (0.01), HD(alu_b) (0.01), instr[19:15] (0.01), instr[6:0] (0.01), instr[31:25] (0.01)
Execute_stage	HD(alu_a[7:0]) (0.51), HD(alu_a[23:16]) (0.19), HD(alu_operator) (0.10), HD(mult_a[7:0]) (0.07), HD(alu_b[23:16]) (0.03), HD(alu_b[7:0]) (0.02), HD(alu_a[15:8]) (0.02), HD(alu_b[31:24]) (0.02), HD(alu_a[31:24]) (0.02), HD(mult_b[7:0]) (0.01)
LS_unit	HD(data_rdata[15:8]) (0.52), HD(b[7:0]) (0.20), HD(b[31:24]) (0.07), HD(data_rdata[7:0]) (0.04), HD(a[7:0]) (0.02), HD(a[15:8]) (0.02), HD(b[15:8]) (0.02), HD(a[23:16]) (0.01), HD(a[31:24]) (0.01), HD(data_wdata[15:8]) (0.01)
CSR	HD(csr_wdata) (0.95), HD(pc_if) (0.03), HD(branch_i) (0.01)
Pmp_unit	HD(data_addr) (0.95), HD(instr_addr) (0.04)
Core	HD(pc_if) (0.69), HD(data_addr) (0.16), HD(instr_rdata[31:25]) (0.04), HD(csr_wdata) (0.02), instr_rdata (0.01), HD(alu_operator) (0.01), HD(alu_a) (0.01), instr_addr (0.01), HD(data_rdata[7:0]) (0.004), HD(instr_addr) (0.004)

5.4 Model Accuracy and Learning Rate

Figure 5.3 summarizes the accuracy of evaluated machine learning algorithms in modeling different blocks in the RI5CY. Decision tree based models are superior than the other linear variants in modeling the power consumption and better than the boosting based model of equivalent complexity.

Table 5.4 lists the major features selected by the decision tree arranged

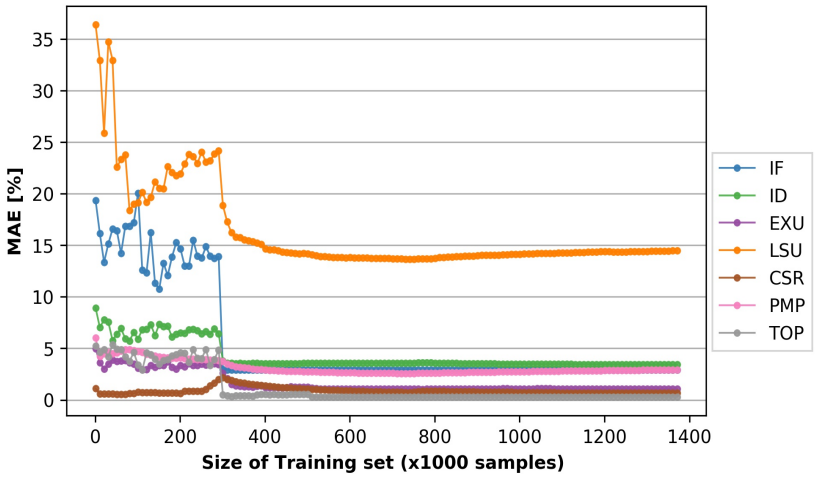


Figure 5.4: Learning curve for different blocks.

in ascending order of importance (normalized to 1), where HD(x) denotes that the hamming distance of x. This ranking can convey additional information about the power behavior to drive power optimizations.

Figure 5.4 shows the training curve and learning rate of decision tree based power models of different blocks for the best fold. The main learning overhead is the time required for reference gate-level simulations. As Figure 5.4 shows, in all cases, the model is able to learn power behavior with less than 300K cycle-level samples and hence instructions needing to be simulated, compared to the 2.2M samples required to train the CNNs in [4].

Finally, Table 5.5 summarizes the performance of the decision tree (DT) based power model for different blocks, including hierarchical composition of the core using either a standalone model or as the sum of sub-block models with and without a dedicated glue logic model. Modeling the core power with

Table 5.5: Predicted power statistics of decision tree (DT) based power model.

Block	**Avg_Pwr**	**Max_Pwr**	**Min_Pwr**	**MAE**	**ME**	**AE**
Fetch_stage	0.29mW	0.42mW	0.11mW	5.38%	66.0%	0.26%
Decode_stage	0.49mW	1.01mW	0.35mW	3.43%	58.0%	0.14%
Regfile	0.26mW	0.40mW	0.22mW	4.44%	64.0%	0.08%
Execute_stage	0.21mW	0.58mW	0.17mW	1.84%	90.9%	0.02%
ALU	0.11mW	0.35mW	0.07mW	4.28%	147%	0.10%
Multiplier	0.09mW	0.49mW	0.09mW	0.30%	235%	0.09%
LS_unit	0.03mW	0.13mW	0.02mW	16.65%	210%	0.34%
CSR	0.25mW	0.31mW	0.18mW	1.17%	16.9%	0.01%
Pmp_unit	0.16mW	0.54mW	0.12mW	4.15%	114%	0.01%
Core (standalone)	1.48mW	2.63mW	0.99mW	1.07%	40.7%	0.04%
Core (composed)	1.48mW	2.63mW	0.99mW	3.37%	34.7%	2.83%
Core (w/ glue logic)	1.48mW	2.63mW	0.99mW	2.15%	34.8%	0.006%

single DT model has a mean absolute error of 1.07%. By contrast, building a power model for core by composing block-level power models has a much higher MAE of 3.37%. This can be compensated and made more accurate by modeling the glue logic as a virtual block (with 2.15% MAE compared to 15.53% error rate on predicting the average power of the core every cycle).

Chapter 6

Superscalar Out-of-Order Core Results

In this chapter, we further demonstrate our power modeling approach by generating a power model for a complex superscalar out-of-order (OoO) core. We integrate models of different general and OoO-specific micro-architectural components. We target the Berkeley Out-of-Order Machine (BOOM), an open source RISC-V implementation of an out-of-order processor [32] in its Medium configuration. BOOM implements the open-source RISC-V ISA (RV64G) and its generator is written in Chisel hardware construction language. BOOM implements a seven-stage pipeline consisting of the following stages: Fetch, Decode/Rename, Rename/ Dispatch, Issue/RegisterRead, Execute, Memory and Writeback (Commit occurs asynchronously). More information about the BOOM micro-architecture can be found in [33]. Configuration parameters of the core used in our work are tabulated in Table 6.1.

6.1 Setup

Figure 6.1 shows the major blocks in the core. RTL for simulation and synthesis is generated from the Chisel generator using the Chipyard framework. The generated RTL is synthesized with the Nangate 45nm PDK [28]

Table 6.1: BOOM core features.

Frontend Parameters	
Fetch Width	2 instructions
Branch Target Buffer	64 Sets x 2 Ways
Bi-Modal Table	1024 entries across 2 banks
Branch predictor	Gshare - 4096 entries
Core Parameters	
Issue width	4 uops (2 int, 1 mem, 1 fp)
Issue window size	Int - 16, Mem - 16, FP - 16
Register file	Int - 80 phsyical registers - 6r3w FP - 64 physical registers - 3r2w
Execution unit	iALU + iMul + iFPU iALU + iDiv FMA + fDiv AGU
Load/Store Unit	16 loads/ 16 stores
Max Branch count	8
ROB	64 entries

using Synopsys Design compiler (M-2016.12-SP4) [29], where clock gating is automatically inferred during synthesis. The buffers in the core are synthesized as flip-flop arrays. The clock tree is synthesized and the netlist is placed and routed using Cadence Innovus (16.13-s045) [34]. For training and validating our power models, we use the micro-benchmarks from the riscv-tests suite [35]. In addition, we validate our trained model on a segment of the CoreMark benchmark suite, representing real-world applications. The list of benchmarks and their brief description is given in Table 6.2. In our experiments, we focus on running baremetal simulations and treat the address translation related logic, such as the TLB, as overhead power as there are no actual translations involved. Synopsys VCS (M-2017.03-SP2) [30] is used for running zero-delay gate-level simulation of the placed-routed netlist (@333.3MHz) for the chosen

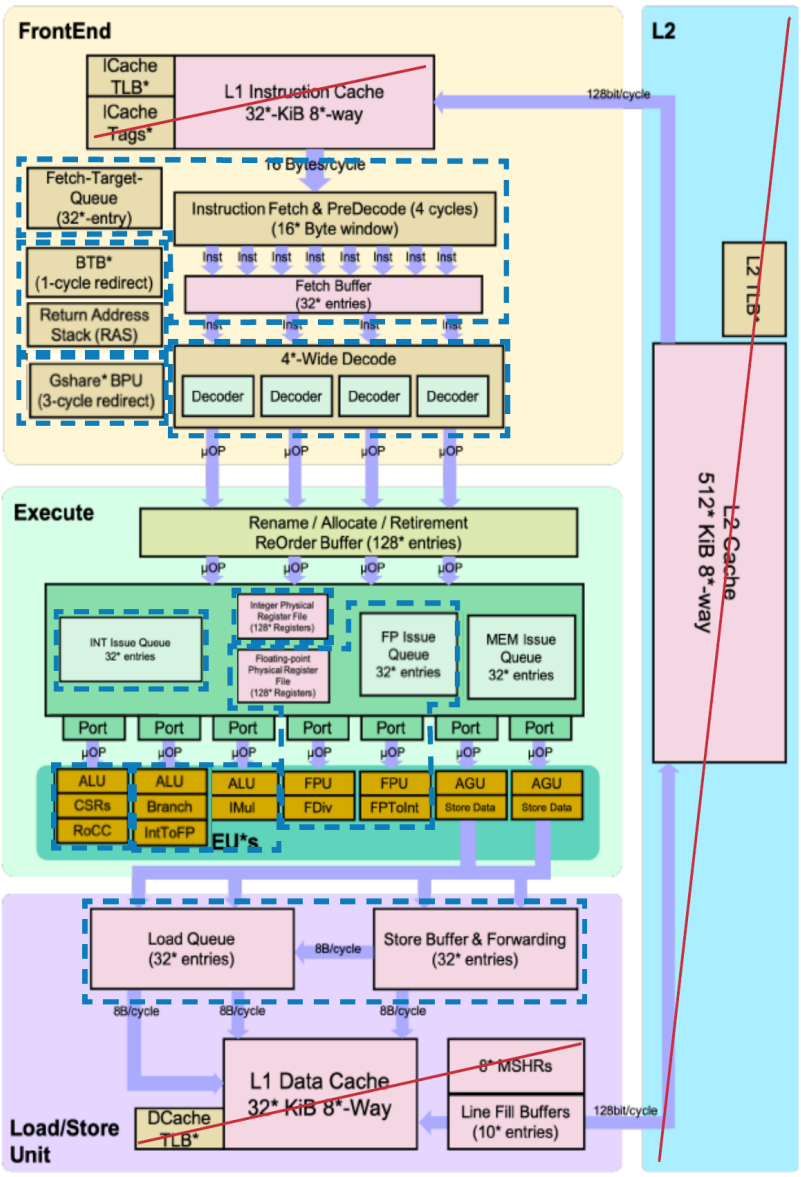


Figure 6.1: BOOM block diagram [33].

Table 6.2: Benchmarks.

Group	Test	Description	Cycles
riscv-tests	dhrystone	A synthetic embedded integer benchmark	242630
	mm	Performs a floating-point matrix multiply	289521
	multiply	A software implementation of multiply	70134
	median	Performs a 1D three element median	36735
	vvadd	Sums two arrays and writes into a third array	21794
	towers	Solves the Towers of Hanoi puzzle recursively	22094
	spmv	Double-precision sparse matrix-vector multiplication	151015
	qsort	Sorts an array of integers using quick sort	450577
CoreMark		list processing, matrix manipulation, state machine, and CRC	475888 (segment)

benchmarks. The golden reference trace of cycle-by-cycle power used for training and validation, is obtained from PrimeTime PX [10] running in time-based power mode. Table 6.3 shows the gate counts and power statistics of different micro-architectural blocks running the CoreMark benchmark. To the best of our knowledge, no highly correlated cycle-accurate simulator for BOOM currently exists. We instead use Verilator [21] models to extract signals needed to construct features for the power models. We avoid using signals that will not be available in a high-level simulation. Verilator models can be directly replaced with a cycle-accurate simulator, once one exists. The Scikit-learn [31] python package is used for model synthesis and prediction.

Table 6.3: Power statistics of BOOM core blocks.

Block	**Cells (cb/sq)**	**Avg.**	**Std. dev.**	**Max**	**Min**
Fetch controller	21567/4342	11.89 mW	3.90 mW	21.59 mW	2.13 mW
Branch Target Buffer	39757/18330	12.22 mW	2.36 mW	26.13 mW	6.01 mW
Branch Predictor	20855/20493	22.67 mW	9.15 mW	56.77 mW	17.23 mW
Decode unit - 0	695/0	0.51mW	0.416mW	1.625mW	0.03mW
Decode unit - 1	685/0	0.52mW	0.38mW	1.59mW	0.03mW
Rename Stage - Maptable	7819/1992	3.11 mW	2.13 mW	9.34 mW	0.76 mW
Rename Stage - Freelist	3836/713	2.25 mW	1.21 mW	4.37 mW	0.24 mW
FP Rename Stage - Maptable	7696/1768	2.59 mW	2.01 mW	8.57 mW	0.77 mW
FP Rename Stage - Freelist	2999/569	0.64 mW	0.71 mW	2.70 mW	0.19 mW
Issue unit	18527/2130	6.73 mW	3.96 mW	22.96 mW	1.51 mW
Mem issue unit	12857/1570	3.61 mW	2.11 mW	13.34 mW	1.22 mW
Iregister file	57290/5177	11.15 mW	7.53 mW	45.84 mW	4.19 mW
Iregister read	3981/971	3.96 mW	0.82 mW	7.77 mW	2.61 mW
CSR	5080/1043	1.28 mW	0.96 mW	7.33 mW	0.63 mW
ALU	41190/1616	7.23 mW	2.94 mW	35.05 mW	3.55 mW
CSR Exe Unit	8125/349	1.28 mW	0.96 mW	7.33 mW	0.63 mW
FP Pipeline	89460/10148	9.23 mW	0.75 mW	37.77 mW	8.17 mW
LSU	26188/5449	7.62 mW	2.79 mW	20.37 mW	4.45 mW
ROB	9094/4329	5.08 mW	1.71 mW	10.68 mW	1.54 mW
GLUE	2119/227	19.09 mW	4.28 mW	47.91 mW	8.66 mW
Core	378440/81216	132.97 mW	27.48 mW	236.33 mW	71.99 mW

6.2 Power Model

The block clusters for power modeling purposes are highlighted with dotted boxes in Figure 6.1. In this section, we analyze and map different blocks in the BOOM core to the power modeling approaches discussed in Chapter 4. Table 6.4 summarizes the different blocks and the approaches we follow to model those blocks.

Table 6.4: Power modeling features of different BOOM blocks.

Block	**Data**	**Control**	**Data Gating**	**Clock Gating**	**Buffer**	**Pipeline**
Fetch controller (FC)	✓	✓			✓	✓
Branch Target Buffer (BTB)					✓	
Branch Predictor (BPD)					✓	
Decode unit - 0 (DEC0)	✓	✓				
Decode unit - 1 (DEC1)	✓	✓				
Rename Stage - Maptable (RNM)				✓	✓	
Rename Stage - Freelist (RNF)				✓	✓	
FP Rename Stage - Maptable (F_RNM)				✓	✓	
FP Rename - Freelist (F_RNF)				✓	✓	
Issue unit (ISS)					✓	
Mem issue unit (M_ISS)					✓	
Iregister file (IRF)					✓	
Iregister read (IRR)	✓					
CSR					✓	
ALU	✓		✓			✓
CSR Exe Unit (CSRX)	✓		✓			✓
FP Pipeline (FP)	✓		✓		✓	✓
LSU	✓	✓			✓	
ROB		✓			✓	
GLUE	✓					

6.2.1 Frontend, Decode and Rename Blocks

The Fetch controller (FC) is comprised of the fetch buffer (FB) that holds instructions for decode, the fetch target queue (FTQ) that tracks the branch prediction information associated with the in-flight Micro-Ops, RVC expanders that can expand compressed instructions, branch decoders computing target and branch type, and a branch checker that compares the predicted target to the computed target. It is comprised of blocks working across 2 out of 3 main fetch pipeline stages, and maps to our pipelined approach.

The Branch target buffer (BTB) module is comprised of data arrays allocated for the bimodal predictors (BIM) and data and tag arrays for predicting the target address. We map all the individual structures to the buffer

category and use corresponding modes, addresses as features.

The Gshare Backup predictor (BPD) is comprised of a counter table with 4096 entries that maps to the buffer structure. Pessimistically, we assume that the hash function can vary between the architecture-level model and actual implementation and only use the control determining read/write operation and the actual data value as features.

Decode units - 0 and 1 map to the mixed attribute blocks category and are modeled by decomposing the instruction word as explained in Section 4.1.5.

Integer and floating point (FP) rename stage majorly comprises of maptable, responsible for mapping the input logical registers to the corresponding physical tags and freelist, responsible to allocate a physical tag for the destination operand of the incoming uop from the list of free physical tags. The BOOM core implements a rename snapshot methodology to facilitate single-cycle rollback on mispredictions. There exist 8 sets of snapshot registers in the maptable and freelist, one for each outstanding branch that is supported in the core. This architecture can be automatically and efficiently transformed to clock gated buffers by the synthesis tool, which we confirmed from our analysis of the synthesized netlist. Mapping to our clock-gating and buffer approaches, we choose the conditions corresponding to the enable of the clock gates as well as the logical destination (available from the instruction) as features.

6.2.2 Issue and Execute Blocks

Issue units hold the dispatched micro-ops until they are executed. Once an operand is ready, the corresponding slot request bit is set. Among the slots raising requests, an issue width number of slots are selected on age-based priority. For example, the integer issue unit has 16 issue slots, 2 dispatch and issue interface and 5 wakeup interfaces that trigger comparison with all of the occupied issue slots. The comparisons in the issue units are based on the physical tags, which can differ between high-level model and an implementation, thereby cannot be used as features. Also, BOOM implements the issue unit as collapsing queue, which makes it difficult to model unlike other normally indexed buffers. We have approximately modeled the unit based on the number of slots that change occupancy status at this cycle along with the incoming micro-op code. The collapsing behavior of the queues is an interesting attribute for power modeling, which we leave for future studies.

The unified integer physical register file (IRF) maps to the buffer category. We limit our features to only data signals since the address is implementation dependent. The integer register read (IRR) block corresponds to the bypass muxes before the execution unit. It collects operands from the register file and the output of the execution units and can direct correct operands to the inputs of the execution units based on the micro-op that is being issued in the current cycle. We map this block to the data dominant circuit approach and chose the byte-wise hamming of all the input data interfaces as features.

Finally, the integer ALU execution unit (ALU) is comprised of the

arithmetic and logical blocks, the integer to floating point unit and a pipelined multiplier unit. Inputs to these blocks are data-gated based on the executed micro-op. Based on this analysis, the ALU combines data-dominant, data-gated and pipelined features. The CSR execution unit (CSRX) is comprised of an integer arithmetic and logical block and an integer divider, which we deal with a similar fashion as ALU. The floating point pipeline (FP) includes the floating point register file and the floating point execution unit capable of fused multiply-add and floating point to integer operations. Hence, it uses data processing, data-gated, pipelined and buffered circuit features. For the execute blocks, we further fine-tune our approach to model only the dominant stage in the pipeline by using the corresponding input from the stored last N inputs history as modeling all the stages resulted only in a small improvement in the accuracy and comes with a higher complexity and training cost.

6.2.3 LSU and ROB

The Load store unit (LSU) is comprised mainly of the store address, data and load address queues. The entire functionality can be referred to in [33]. For power modeling, we focus on the key characteristics of enqueueing and dequeueing the buffers (mapped to buffer circuits approach), triggering of store address queue comparisons (dependent stores) when there is an incoming load, triggering of comparisons of executed young loads when there is an incoming store (for detecting memory ordering failure), and store queue hit. In addition to modeling the control modes capturing the characteristics listed above, we

model the data routing and selection logic involved as well.

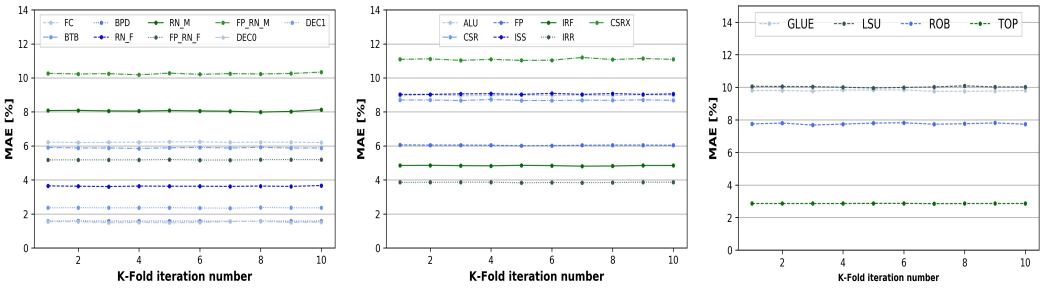
The Reorder buffer (ROB) is implemented as a circular buffer enqueued on tail on dispatch and dequeued from the head of the buffer as the oldest instruction completes. In addition, valid writebacks, the load store unit clearing busy bits, branch misprediction, etc. can cause different operations to be carried out in the associated cycle. We map ROB to the control dominant and buffer features and chose the different control signals as well the micro-op that is being written or read from. Similar to the approach for other blocks, we did not choose features to model the interface dealing with the stale physical destination tag.

6.2.4 Glue Power Model and Composition

We model the glue portion of the core, dominated by mostly the routing structures, as a dataprocessing block with the instruction and data from the dcache interface as features. We compose a core power model by integrating all the sub-block level power models including the glue model.

6.3 Cross-Validation Results

10-fold cross-validation is used for the evaluation of the feature correlation and accuracy of the models on the cumulative data samples constructed from the 8 benchmarks in the riscv-tests suite. As in our experiments with the RI5CY core, a decision tree based power model performs consistently better than linear models as well gradient boosting and random forest based models



(a) Frontend and Rename (b) Issue and execute (c) LSU, ROB and Core

Figure 6.2: 10-fold cross-validation results.

of equivalent complexity. This implies the capability of a deeper decision tree to capture the non-linear power characteristics of different micro-architectural blocks in CPUs. Figure 6.2 summarizes the 10-fold cross-validation results of decision tree based models for the different sub-blocks. A decision tree based power model for the fetch controller has an MAE of 6.22% on average. For the branch target buffer, we observe that the BIM writes causes significant power variance. The decision tree based model ranks the mode to be the most important feature for power modeling and can predict cycle-by-cycle power with an MAE of 5.88%. The write mode and value correlates more tightly to the power variance in case of the backup predictor and a decision tree based model could learn the correlation and has an effective average MAE of 1.58%.

Power models for the decode units 0 and 1 have an average MAE of 1.52% and 2.361%, respectively. Generated decision tree based power models can capture the power variance due to clock gating and have MAEs of 3.634%, 8.061%, 5.184%, 10.25% for the integer freelist, maptable and floating point freelist and maptable, respectively.

Due to the limitation in modeling the collapsing queue behaviour and not using the physical tags as features, integer issue unit and mem issue unit power models have a higher MAE of 9.05% and 19%, respectively. By contrast, by only modeling the data interface signals, our power model for the register file can reach an accuracy of 95.16% (MAE - 4.84%). The model for the integer register read has an MAE of 3.85% on average. The power models for the execution units - ALU, CSRX, FP - have an average MAE of 8.97%, 11.091%, 6.041%, respectively.

Our generated power model can predict cycle-by-cycle power of the LSU with an average MAE of 10.025%. Our analysis of a decision tree based power model for LSU without data and address features shows a degradation of accuracy by 11.5% (MAE - 21.625%), highlighting the significance of capturing data dependent power characteristics in cycle-accurate power models. Note that the address translation related logic is not modeled specifically as the virtual memory is disabled in our bare-metal simulations. Our power model for ROB has an average MAE of 7.762%.

Finally, a hierarchical power model for the BOOM core, built by integrating decision tree based power models for the micro-architectural blocks and the glue logic, has an average MAE of 2.86%.

6.4 Model Accuracy

We validate our power models by training them on the riscv-test power traces and testing the trained model on a segment of the CoreMark benchmark

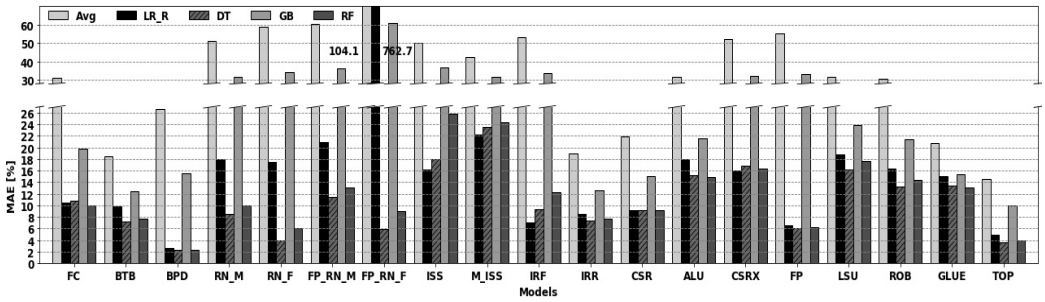


Figure 6.3: Accuracy of block models.

test. Figure 6.3 shows the MAE of the major blocks for different ML models we evaluated.

In general, we observe that a deeper decision tree can learn the non-linear power characteristics better than a linear model or a ensemble model of similar complexity. It is noticeable from the Figure 6.3 that our decision tree based models for different blocks can effectively learn the correlation between the high level activity information and power consumption and have much higher accuracy than just predicting the average power of the block every cycle.

Table 6.5 summarizes the performance of decision tree based models for different micro-architectural blocks. Power models for the ALU and CSRX blocks have degraded accuracy when evaluated on an unseen workload, which we attribute to the training set's incomplete coverage in the data input space. Preliminary results obtained by selectively moving around 50 cycles from the CoreMark test set to the training set show an accuracy improvement of upto 2% for these blocks. Generation of training sets for all the CPU blocks with

Table 6.5: Predicted power statistics of decision tree (DT) based power model.

Block	**Avg_Pwr**	**Max_Pwr**	**Min_Pwr**	**MAE**	**ME**	**AE**
Fetch controller	11.36mW	19.81mW	2.45mW	10.86%	74.44%	4.48%
Branch Target Buffer	11.96mW	22.66mW	6mW	7.18%	95.45%	2.17%
Branch Predictor	22.4mW	54.34mW	17.4mW	2.38%	21.38%	2.38%
Decode unit - 0	0.51mW	1.50mW	0.03mW	8.27%	155.90%	0.21%
Decode unit - 1	0.52mW	1.41mW	0.03mW	9.15%	182.98%	0.58%
Rename Stage - Maptable	3.13mW	9.47mW	0.79mW	8.51%	80.22%	0.61%
Rename Stage - Freelist	2.25mW	4.18mW	0.24mW	3.88%	46.08%	0.19%
FP Rename Stage - Maptable	2.55mW	8.63mW	0.82mW	11.44%	66.35%	1.48%
FP Rename Stage - Freelist	0.63mW	2.81mW	0.22mW	5.97%	260.65%	1.25%
Issue unit	6.19mW	22.02mW	2.03mW	17.92%	215.26%	6.11%
Mem issue unit	3.26mW	15.22mW	1.92mW	22.84%	229.04%	9.46%
Iregister file	10.9mW	42.17mW	4.29mW	9.34%	137.63%	2.2%
Iregister read	3.91mW	7.56mW	2.66mW	7.42%	81.27%	1.19%
CSR	0.91mW	3.32mW	0.66mW	9.1%	139.59%	3.25%
ALU	6.96mW	29.66mW	3.91mW	15.15%	362.82%	3.78%
CSR Exe Unit	1.30mW	8.13mW	0.68mW	16.85%	310.42%	1.07%
FP Pipeline	9.27mW	20.11mW	8.74mW	6.14%	263.18%	0.47%
LSU	7.72mW	21.64mW	5.7mW	16.27%	132.35%	1.32%
ROB	5.08mW	9.26mW	1.76mW	13.24%	117.78%	0.09%
Core (composed)	117.05mW	199.78mW	68.12mW	14.14%	41.79%	12.14%
Core (w/ glue logic)	136.01mW	221.56mW	77.62mW	3.59%	26.88%	0.23%

good coverage of the signal activity space is left for future work.

Our hierarchically composed core level power model can predict the power of the evaluated segment of CoreMark workload per cycle to within 3.59% of a gate-level power estimate (compared to 14.62% MAE of predicting average core power), all by just using the features that are available in a high-level simulation.

Chapter 7

Summary and Future Work

This chapter briefly reviews and summarizes the report. Then, we discuss future research opportunities.

7.1 Summary

In this report, we presented a hierarchical power modeling approach that supports development of simple yet accurate power models for CPUs and their internal components at micro-architecture levels of abstractions. We presented a methodology for feature selection and engineering that enables using low complexity learning formulations to accurately model common micro-architectural sub-blocks in CPUs. Our core power model, synthesized by integrating these sub-block level models, provides cycle-accurate power estimates at sub-block granularity with low training overhead using features that are extracted from micro-architecture simulations. Results show that decision tree based hierarchically composed models, built using our approach, can predict cycle-by-cycle power consumption with less than 2.2% and 2.9% error rate for RI5CY core and BOOM core, respectively. Resultant BOOM core power models can also predict cycle-by-cycle power consumption of an unseen workload (CoreMark) to within 3.6% of a gate-level power estimate.

7.2 Future Work

In our work so far, we have been able to build accurate power models for CPUs at cycle-level of granularity. In the following, we outline the possible future research directions to extend and augment our work:

- The performance of supervised learning algorithms heavily depends on the training set quality. Limitations of insufficient training showed up as large maximum error values in our experiments. An ideal approach in assessing the training set quality is to analyze its coverage at the circuit level. This potentially leads to challenges in high-level power modeling where all circuit level details are still unavailable. One possible direction of future work is to come up with a proper training assessment methodology and automatic generation of training sets that have good coverage.

- Our primary focus in this work was to generate accurate cycle-level power traces for the processors and hence, providing a cycle and bit-accurate high level model for the CPUs. However, micro-architecture simulators often approximate the implementation. The resulting performance models are no longer cycle-accurate, but correlate only over larger number of cycles or over a benchmark segment. To couple with these approximate high-level performance models, our approach needs to be modified to predict average power rather than at the cycle-level. Based on preliminary results with our setup, we found that splitting the total power of

the blocks into its circuit-level components - clock tree power, sequential power, combinational power, etc. - and following an ensemble learning approach that trains different models for each component is promising as a potential average power modeling methodology. For example, for a source clock gated block, the clock tree power is linear over the number of cycles the clock gate is enabled, which can be easily learned with a linear ML model.

- Our general power modeling based on the attributes of the blocks can be easily extended to other processors including massively parallel processors such as GPUs. Extending it over other domains of processor families can be a potential future research direction.

Appendix

Appendix 1

BOOM Power Modeling Features List

This appendix contains the list of features used for generating power models for different blocks in the BOOM core described in Chapter 6. The notation used in the tables are as follows: X denotes the value of the signal X in the current cycle, DEL(X) denotes value of the signal X in the previous cycle and HD(X) denotes hamming distance between values in the current and previous cycle of signal X . The importance of each feature in a decision tree based power model are presented in brackets next to the feature.

Table 1.1: Fetch controller features

Features (importances)	**Description**
HD(BranchDecode.io_inst[31:24]) (0.007)	Input instruction to the branch decoder from RVC expander module.
HD(BranchDecode.io_inst[23:16]) (0.001)	
HD(BranchDecode.io_inst[15:8]) (0.002)	
HD(BranchDecode.io_inst[7:0]) (0.002)	
HD(BranchDecode_1.io_inst[31:24]) (0.005)	
HD(BranchDecode_1.io_inst[23:16]) (0.002)	
HD(BranchDecode_1.io_inst[15:8]) (0.001)	
HD(BranchDecode_1.io_inst[7:0]) (0.027)	
HD(BranchDecode_2.io_inst[31:24]) (0.006)	
HD(BranchDecode_2.io_inst[23:16]) (0.001)	
HD(BranchDecode_2.io_inst[15:8]) (0.002)	
HD(BranchDecode_2.io_inst[7:0]) (0.001)	
HD(BranchDecode_3.io_inst[31:24]) (0.001)	
HD(BranchDecode_3.io_inst[23:16]) (0.001)	

Features (importances)	**Description**
HD(BranchDecode_3.io_inst[15:8]) (0.002)	
HD(BranchDecode_3.io_inst[7:0]) (0.003)	
RVCExpander.io_rvc (0.004)	Compressed instruction control signal
RVCExpander_1.io_rvc (0.003)	
RVCExpander_2.io_rvc (0.002)	
RVCExpander_3.io_rvc (0.003)	
HD(bchecker.io_aligned_pc[39:32]) (0)	Pipelined version of raw PC at fetch controller level
HD(bchecker.io_aligned_pc[31:24]) (0)	
HD(bchecker.io_aligned_pc[23:16]) (0)	
HD(bchecker.io_aligned_pc[15:8]) (0.001)	
HD(bchecker.io_aligned_pc[7:0]) (0.007)	
HD(bchecker.io_br_targs_0[39:32]) (0)	Computed branch targets in the branch decode module
HD(bchecker.io_br_targs_0[31:24]) (0.001)	
HD(bchecker.io_br_targs_0[23:16]) (0.006)	
HD(bchecker.io_br_targs_0[15:8]) (0.001)	
HD(bchecker.io_br_targs_0[7:0]) (0.766)	
HD(bchecker.io_br_targs_1[39:32]) (0)	
HD(bchecker.io_br_targs_1[31:24]) (0.001)	
HD(bchecker.io_br_targs_1[23:16]) (0.001)	
HD(bchecker.io_br_targs_1[15:8]) (0.001)	
HD(bchecker.io_br_targs_1[7:0]) (0.001)	
HD(bchecker.io_br_targs_2[39:32]) (0)	
HD(bchecker.io_br_targs_2[31:24]) (0.001)	
HD(bchecker.io_br_targs_2[23:16]) (0.004)	
HD(bchecker.io_br_targs_2[15:8]) (0.001)	
HD(bchecker.io_br_targs_2[7:0]) (0.001)	
HD(bchecker.io_br_targs_3[39:32]) (0)	
HD(bchecker.io_br_targs_3[31:24]) (0.001)	
HD(bchecker.io_br_targs_3[23:16]) (0.001)	
HD(bchecker.io_br_targs_3[15:8]) (0.001)	
HD(bchecker.io_br_targs_3[7:0]) (0.001)	
HD(bchecker.io_btb_resp_bits_target[38:32]) (0)	Branch target predicted by the branch target buffer
HD(bchecker.io_btb_resp_bits_target[31:24]) (0.001)	
HD(bchecker.io_btb_resp_bits_target[23:16]) (0)	
HD(bchecker.io_btb_resp_bits_target[15:8]) (0.001)	
HD(bchecker.io_btb_resp_bits_target[7:0]) (0.019)	

Features (importances)	**Description**
HD(bchecker.io_fetch_pc[39:32]) (0)	Fetch PC at current cycle
HD(bchecker.io_fetch_pc[31:24]) (0)	
HD(bchecker.io_fetch_pc[23:16]) (0)	
HD(bchecker.io_fetch_pc[15:8]) (0.001)	
HD(bchecker.io_fetch_pc[7:0]) (0.011)	
HD(bchecker.io_jal_targs_0[39:32]) (0)	Computed jump targets in the branch decode module
HD(bchecker.io_jal_targs_0[31:24]) (0.001)	
HD(bchecker.io_jal_targs_0[23:16]) (0.001)	
HD(bchecker.io_jal_targs_0[15:8]) (0.001)	
HD(bchecker.io_jal_targs_0[7:0]) (0.001)	
HD(bchecker.io_jal_targs_1[39:32]) (0)	
HD(bchecker.io_jal_targs_1[31:24]) (0.001)	
HD(bchecker.io_jal_targs_1[23:16]) (0.001)	
HD(bchecker.io_jal_targs_1[15:8]) (0.001)	
HD(bchecker.io_jal_targs_1[7:0]) (0.001)	
HD(bchecker.io_jal_targs_2[39:32]) (0.001)	
HD(bchecker.io_jal_targs_2[31:24]) (0.001)	
HD(bchecker.io_jal_targs_2[23:16]) (0.001)	
HD(bchecker.io_jal_targs_2[15:8]) (0.001)	
HD(bchecker.io_jal_targs_2[7:0]) (0.001)	
HD(bchecker.io_jal_targs_3[39:32]) (0)	
HD(bchecker.io_jal_targs_3[31:24]) (0.001)	
HD(bchecker.io_jal_targs_3[23:16]) (0.001)	
HD(bchecker.io_jal_targs_3[15:8]) (0.001)	
HD(bchecker.io_jal_targs_3[7:0]) (0.001)	
HD(bchecker.io_req_bits_addr[39:32]) (0)	F4 stage - request address
HD(bchecker.io_req_bits_addr[31:24]) (0)	
HD(bchecker.io_req_bits_addr[23:16]) (0)	
HD(bchecker.io_req_bits_addr[15:8]) (0.001)	
HD(bchecker.io_req_bits_addr[7:0]) (0.001)	
HD(fb.io_enq_bits_exp_insts_0[31:24]) (0.001)	Instructions enqueued into fetch buffer every cycle
HD(fb.io_enq_bits_exp_insts_0[23:16]) (0.082)	
HD(fb.io_enq_bits_exp_insts_0[15:8]) (0.026)	
HD(fb.io_enq_bits_exp_insts_0[7:0]) (0.001)	
HD(fb.io_enq_bits_exp_insts_1[31:24]) (0.001)	
HD(fb.io_enq_bits_exp_insts_1[23:16]) (0.001)	

Features (importances)	Description
HD(fb.io_enq_bits_exp_insts_1[15:8]) (0.001)	
HD(fb.io_enq_bits_exp_insts_1[7:0]) (0.001)	
HD(fb.io_enq_bits_exp_insts_2[31:24]) (0.001)	
HD(fb.io_enq_bits_exp_insts_2[23:16]) (0.001)	
HD(fb.io_enq_bits_exp_insts_2[15:8]) (0.001)	
HD(fb.io_enq_bits_exp_insts_2[7:0]) (0.001)	
HD(fb.io_enq_bits_exp_insts_3[31:24]) (0.001)	
HD(fb.io_enq_bits_exp_insts_3[23:16]) (0.001)	
HD(fb.io_enq_bits_exp_insts_3[15:8]) (0.002)	
HD(fb.io_enq_bits_exp_insts_3[7:0]) (0.002)	
HD(ftq.io_enq_bits_bpd_info[26:24]) (0)	Branch predictor information - control flow instruction index, counter entry, PC and history used for branch prediction enqueued into Fetch Target Queue
HD(ftq.io_enq_bits_bpd_info[23:16]) (0)	
HD(ftq.io_enq_bits_bpd_info[15:8]) (0.001)	
HD(ftq.io_enq_bits_bpd_info[7:0]) (0.002)	
HD(ftq.io_enq_bits_fetch_pc[39:32]) (0)	
HD(ftq.io_enq_bits_fetch_pc[31:24]) (0)	
HD(ftq.io_enq_bits_fetch_pc[23:16]) (0)	
HD(ftq.io_enq_bits_fetch_pc[15:8]) (0.001)	
HD(ftq.io_enq_bits_fetch_pc[7:0]) (0.001)	
HD(ftq.io_enq_bits_history[22:16]) (0.001)	
HD(ftq.io_enq_bits_history[15:8]) (0.001)	
HD(ftq.io_enq_bits_history[7:0]) (0.001)	

Table 1.2: Branch Target Buffer features

Features (importances)	Description
HD(bim.bim_data_array_.RW0_addr[8:0]) (0.13)	Bimodal predictor array - address, mode and number of bits to be written in current cycle
bim.bim_data_array_.RW0_wmode (0.01)	
bim.bim_data_array_.RW0_wmask_0 (0.01)	
bim.bim_data_array_.RW0_wmask_1 (0.01)	
bim.bim_data_array_.RW0_wmask_2 (0.01)	
bim.bim_data_array_.RW0_wmask_3 (0.01)	
bim.bim_data_array_.RW0_wmask_4 (0.01)	
bim.bim_data_array_.RW0_wmask_5 (0.01)	
bim.bim_data_array_.RW0_wmask_6 (0.01)	
bim.bim_data_array_.RW0_wmask_7 (0.01)	

Features (importances)	Description
bim.bim_data_array_.RW0_en (0.02)	
HD(bim.bim_data_array_1.RW0_addr[8:0]) (0.21)	
bim.bim_data_array_1.RW0_wmode (0.01)	
bim.bim_data_array_1.RW0_wmask_0 (0.01)	
bim.bim_data_array_1.RW0_wmask_1 (0.01)	
bim.bim_data_array_1.RW0_wmask_2 (0.01)	
bim.bim_data_array_1.RW0_wmask_3 (0.01)	
bim.bim_data_array_1.RW0_wmask_4 (0.01)	
bim.bim_data_array_1.RW0_wmask_5 (0.01)	
bim.bim_data_array_1.RW0_wmask_6 (0.01)	
bim.bim_data_array_1.RW0_wmask_7 (0.01)	
bim.bim_data_array_1.RW0_en (0.01)	
HD(btb.btb_data_array.RW0_addr[5:0]) (0.57)	Branch target buffer - address, data and modes
HD(btb.btb_data_array.RW0_wdata_target[37:32]) (0)	
HD(btb.btb_data_array.RW0_wdata_target[31:24]) (0.01)	
HD(btb.btb_data_array.RW0_wdata_target[23:16]) (0)	
HD(btb.btb_data_array.RW0_wdata_target[15:8]) (0.01)	
HD(btb.btb_data_array.RW0_wdata_target[7:0]) (0.01)	
btb.btb_data_array.RW0_wmode (0.01)	
HD(btb.btb_data_array_1.RW0_wdata_target[37:32]) (0)	
HD(btb.btb_data_array_1.RW0_wdata_target[31:24]) (0)	
HD(btb.btb_data_array_1.RW0_wdata_target[23:16]) (0)	
HD(btb.btb_data_array_1.RW0_wdata_target[15:8]) (0.01)	
HD(btb.btb_data_array_1.RW0_wdata_target[7:0]) (0.02)	
btb.btb_data_array_1.RW0_wmode (0.01)	
btb.btb_tag_array.RW0_wmode (0.01)	
HD(btb.btb_tag_array.RW0_wdata[19:0]) (0.01)	
btb.btb_tag_array_1.RW0_wmode (0.01)	
HD(btb.btb_tag_array_1.RW0_wdata[19:0]) (0.01)	

Table 1.3: Branch Predictor features

Features (importances)	Description
counter_table.R0_en (0.01)	Counter table read interface
HD(counter_table.d_R0_data_cfi_idx[1:0]) (0.01)	
HD(counter_table.d_R0_data_counter[1:0]) (0.01)	

Features (importances)	Description
counter_table.W0_en (0.01)	Counter table write interface
HD(counter_table.d_W0_data_cfi_idx[1:0]) (0.13)	
HD(counter_table.d_W0_data_counter[1:0]) (0.87)	

Table 1.4: Decode unit - 0 features

Features (importances)	Description
io_enq_uop_inst[31:25] (0.01)	Instruction word in bit-field decomposed fashion
HD(io_enq_uop_inst[31:25]) (0.18)	
io_enq_uop_inst[24:20] (0.01)	
HD(io_enq_uop_inst[24:20]) (0.02)	
io_enq_uop_inst[19:15] (0.01)	
HD(io_enq_uop_inst[19:15]) (0.01)	
io_enq_uop_inst[14:12] (0.01)	
HD(io_enq_uop_inst[14:12]) (0.01)	
io_enq_uop_inst[11:7] (0.01)	
HD(io_enq_uop_inst[11:7]) (0.01)	
io_enq_uop_inst[6:0] (0.01)	
HD(io_enq_uop_inst[6:0]) (0.8)	

Table 1.5: Decode unit - 1 features

Features (importances)	Description
io_enq_uop_inst[31:25] (0.01)	Instruction word in bit-field decomposed fashion
HD(io_enq_uop_inst[31:25]) (0.16)	
io_enq_uop_inst[24:20] (0.01)	
HD(io_enq_uop_inst[24:20]) (0.02)	
io_enq_uop_inst[19:15] (0.01)	
HD(io_enq_uop_inst[19:15]) (0.01)	
io_enq_uop_inst[14:12] (0.01)	
HD(io_enq_uop_inst[14:12]) (0.01)	
io_enq_uop_inst[11:7] (0.01)	
HD(io_enq_uop_inst[11:7]) (0.02)	
io_enq_uop_inst[6:0] (0.01)	
HD(io_enq_uop_inst[6:0]) (0.8)	

Table 1.6: Rename Stage - Maptable features

Features (importances)	**Description**
io_remap_reqs_0_ldst[5:0] (0.01)	Logical destination and delayed version for clock gating modeling at cycle accurate level
DEL(io_remap_reqs_0_ldst[5:0]) (0.01)	
io_remap_reqs_1_ldst[5:0] (0.01)	
DEL(io_remap_reqs_1_ldst[5:0]) (0.01)	
io_remap_reqs_0_valid (0.02)	Input request valid control signals
DEL(io_remap_reqs_0_valid) (0.08)	
io_remap_reqs_1_valid (0.01)	
DEL(io_remap_reqs_1_valid) (0.01)	
io_ren_br_tags_0_valid (0.4)	Branch valid control signals
DEL(io_ren_br_tags_0_valid) (0.41)	
io_ren_br_tags_1_valid (0.01)	
DEL(io_ren_br_tags_1_valid) (0.02)	
io_ren_br_tags_0_bits[2:0] (0.01)	Branch tags to model the clock gating for snapshots
DEL(io_ren_br_tags_0_bits[2:0]) (0.01)	
io_ren_br_tags_1_bits[2:0] (0.01)	
DEL(io_ren_br_tags_1_bits[2:0]) (0.01)	
io_brinfo_mispredict (0.02)	Mispredict control signal to denote recovery operation from snapshot
DEL(io_brinfo_mispredict) (0.05)	

Table 1.7: Rename Stage - Freelist features

Features (importances)	**Description**
io_reqs_0 (0.01)	Input request control signals
DEL(io_reqs_0) (0.5)	
io_reqs_1 (0.01)	
DEL(io_reqs_1) (0.28)	
io_alloc_pregs_0_valid (0.01)	Allocation valid control signals
DEL(io_alloc_pregs_0_valid) (0.01)	
io_alloc_pregs_1_valid (0.01)	
DEL(io_alloc_pregs_1_valid) (0.01)	
io_dealloc_pregs_0_valid (0.01)	De-allocation valid control signals
DEL(io_dealloc_pregs_0_valid) (0.01)	
io_dealloc_pregs_1_valid (0.01)	

Features (importances)	Description
DEL(io_dealloc_pregs_1_valid) (0.02)	
io_ren_br_tags_0_valid (0.01)	Branch valid control signals
DEL(io_ren_br_tags_0_valid) (0.16)	
io_ren_br_tags_1_valid (0.01)	
DEL(io_ren_br_tags_1_valid) (0.04)	
io_brinfo_mispredict (0.01)	Branch mispredict control signals
DEL(io_brinfo_mispredict) (0.01)	

Table 1.8: FP Rename Stage - Maptable features

Features (importances)	Description
io_remap_reqs_0_ldst[5:0] (0.02)	Logical destination and delayed version for clock gating modeling at cycle accurate level
DEL(io_remap_reqs_0_ldst[5:0]) (0.01)	
io_remap_reqs_1_ldst[5:0] (0.01)	
DEL(io_remap_reqs_1_ldst[5:0]) (0.01)	
io_remap_reqs_0_valid (0.01)	Input request valid - control signals
DEL(io_remap_reqs_0_valid) (0.01)	
io_remap_reqs_1_valid (0.01)	
DEL(io_remap_reqs_1_valid) (0.06)	
io_ren_br_tags_0_valid (0.37)	Branch valid control signals
DEL(io_ren_br_tags_0_valid) (0.43)	
io_ren_br_tags_1_valid (0.02)	
DEL(io_ren_br_tags_1_valid) (0.05)	
io_ren_br_tags_0_bits[2:0] (0.01)	Branch tags to model the clock gating for snapshots
DEL(io_ren_br_tags_0_bits[2:0]) (0.01)	
io_ren_br_tags_1_bits[2:0] (0.01)	
DEL(io_ren_br_tags_1_bits[2:0]) (0.01)	
io_brinfo_mispredict (0.02)	Mispredict control signal to denote recovery operation from snapshot
DEL(io_brinfo_mispredict) (0.04)	

Table 1.9: FP Rename Stage - Freelist features

Features (importances)	Description
io_reqs_0 (0.01)	Input request control signals

Features (importances)	Description
DEL(io_reqs_0) (0.12)	
io_reqs_1 (0.01)	
DEL(io_reqs_1) (0.41)	
io_alloc_pregs_0_valid (0.01)	Allocation valid control signals
DEL(io_alloc_pregs_0_valid) (0.01)	
io_alloc_pregs_1_valid (0.01)	
DEL(io_alloc_pregs_1_valid) (0.01)	
io_dealloc_pregs_0_valid (0.01)	De-allocation valid control signals
DEL(io_dealloc_pregs_0_valid) (0.01)	
io_dealloc_pregs_1_valid (0.01)	
DEL(io_dealloc_pregs_1_valid) (0.02)	
io_ren_br_tags_0_valid (0.01)	Branch valid control signals
DEL(io_ren_br_tags_0_valid) (0.23)	
io_ren_br_tags_1_valid (0.01)	
DEL(io_ren_br_tags_1_valid) (0.23)	
io_brinfo_mispredict (0.01)	Branch mispredict control signals
DEL(io_brinfo_mispredict) (0.01)	

Table 1.10: Issue unit

Features (importances)	Description
HD(slots_0.slot_uop_fu_code[9:0]) (0.01)	Micro-op functional unit code
HD(slots_1.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_2.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_3.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_4.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_5.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_6.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_7.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_8.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_9.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_10.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_11.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_12.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_13.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_14.slot_uop_fu_code[9:0]) (0.01)	

Features (importances)	**Description**
HD(slots_15.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_0.slot_uop_uopc[8:0]) (0.01)	Micro-op code in slot
HD(slots_1.slot_uop_uopc[8:0]) (0.01)	
HD(slots_2.slot_uop_uopc[8:0]) (0.02)	
HD(slots_3.slot_uop_uopc[8:0]) (0.01)	
HD(slots_4.slot_uop_uopc[8:0]) (0.01)	
HD(slots_5.slot_uop_uopc[8:0]) (0.05)	
HD(slots_6.slot_uop_uopc[8:0]) (0.01)	
HD(slots_7.slot_uop_uopc[8:0]) (0.01)	
HD(slots_8.slot_uop_uopc[8:0]) (0.01)	
HD(slots_9.slot_uop_uopc[8:0]) (0.01)	
HD(slots_10.slot_uop_uopc[8:0]) (0.01)	
HD(slots_11.slot_uop_uopc[8:0]) (0.01)	
HD(slots_12.slot_uop_uopc[8:0]) (0.01)	
HD(slots_13.slot_uop_uopc[8:0]) (0.01)	
HD(slots_14.slot_uop_uopc[8:0]) (0.03)	
HD(slots_15.slot_uop_uopc[8:0]) (0.24)	
HD(slots_0.state[1:0]) (0.03)	State - hamming represents change in occupancy
HD(slots_1.state[1:0]) (0.01)	
HD(slots_2.state[1:0]) (0.01)	
HD(slots_3.state[1:0]) (0.01)	
HD(slots_4.state[1:0]) (0.01)	
HD(slots_5.state[1:0]) (0.02)	
HD(slots_6.state[1:0]) (0.02)	
HD(slots_7.state[1:0]) (0.06)	
HD(slots_8.state[1:0]) (0.03)	
HD(slots_9.state[1:0]) (0.03)	
HD(slots_10.state[1:0]) (0.12)	
HD(slots_11.state[1:0]) (0.03)	
HD(slots_12.state[1:0]) (0.06)	
HD(slots_13.state[1:0]) (0.02)	
HD(slots_14.state[1:0]) (0.06)	
HD(slots_15.state[1:0]) (0.19)	

Table 1.11: Mem issue unit

Features (importances)	**Description**
HD(slots_0.slot_uop_fu_code[9:0]) (0.01)	Micro-op functional unit code
HD(slots_1.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_2.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_3.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_4.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_5.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_6.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_7.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_8.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_9.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_10.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_11.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_12.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_13.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_14.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_15.slot_uop_fu_code[9:0]) (0.01)	
HD(slots_0.slot_uop_uopc[8:0]) (0.01)	Micro-op code in slot
HD(slots_1.slot_uop_uopc[8:0]) (0.01)	
HD(slots_2.slot_uop_uopc[8:0]) (0.01)	
HD(slots_3.slot_uop_uopc[8:0]) (0.01)	
HD(slots_4.slot_uop_uopc[8:0]) (0.01)	
HD(slots_5.slot_uop_uopc[8:0]) (0.01)	
HD(slots_6.slot_uop_uopc[8:0]) (0.01)	
HD(slots_7.slot_uop_uopc[8:0]) (0.01)	
HD(slots_8.slot_uop_uopc[8:0]) (0.01)	
HD(slots_9.slot_uop_uopc[8:0]) (0.01)	
HD(slots_10.slot_uop_uopc[8:0]) (0.01)	
HD(slots_11.slot_uop_uopc[8:0]) (0.01)	
HD(slots_12.slot_uop_uopc[8:0]) (0.01)	
HD(slots_13.slot_uop_uopc[8:0]) (0.01)	
HD(slots_14.slot_uop_uopc[8:0]) (0.01)	
HD(slots_15.slot_uop_uopc[8:0]) (0.01)	
HD(slots_0.state[1:0]) (0.02)	
HD(slots_1.state[1:0]) (0.02)	

Features (importances)	Description
HD(slots_2.state[1:0]) (0.01)	State - hamming represents change in occupancy
HD(slots_3.state[1:0]) (0.06)	
HD(slots_4.state[1:0]) (0.04)	
HD(slots_5.state[1:0]) (0.04)	
HD(slots_6.state[1:0]) (0.03)	
HD(slots_7.state[1:0]) (0.07)	
HD(slots_8.state[1:0]) (0.04)	
HD(slots_9.state[1:0]) (0.02)	
HD(slots_10.state[1:0]) (0.03)	
HD(slots_11.state[1:0]) (0.12)	
HD(slots_12.state[1:0]) (0.25)	
HD(slots_13.state[1:0]) (0.06)	
HD(slots_14.state[1:0]) (0.05)	
HD(slots_15.state[1:0]) (0.17)	

Table 1.12: Iregister file

Features (importances)	Description
HD(io_readports_0_data[63:56]) (0.01)	Read ports data interface
HD(io_readports_0_data[55:48]) (0.01)	
HD(io_readports_0_data[47:40]) (0.01)	
HD(io_readports_0_data[39:32]) (0.01)	
HD(io_readports_0_data[31:24]) (0.01)	
HD(io_readports_0_data[23:16]) (0.01)	
HD(io_readports_0_data[15:8]) (0.01)	
HD(io_readports_0_data[7:0]) (0.01)	
HD(io_readports_1_data[63:56]) (0.01)	
HD(io_readports_1_data[55:48]) (0.01)	
HD(io_readports_1_data[47:40]) (0.01)	
HD(io_readports_1_data[39:32]) (0.01)	
HD(io_readports_1_data[31:24]) (0.01)	
HD(io_readports_1_data[23:16]) (0.01)	
HD(io_readports_1_data[15:8]) (0.01)	
HD(io_readports_1_data[7:0]) (0.01)	
HD(io_readports_2_data[63:56]) (0.01)	
HD(io_readports_2_data[55:48]) (0.01)	

Features (importances)	**Description**
HD(io_readports_2_data[47:40]) (0.01)	
HD(io_readports_2_data[39:32]) (0.01)	
HD(io_readports_2_data[31:24]) (0.01)	
HD(io_readports_2_data[23:16]) (0.01)	
HD(io_readports_2_data[15:8]) (0.01)	
HD(io_readports_2_data[7:0]) (0.01)	
HD(io_readports_3_data[63:56]) (0.01)	
HD(io_readports_3_data[55:48]) (0.01)	
HD(io_readports_3_data[47:40]) (0.01)	
HD(io_readports_3_data[39:32]) (0.01)	
HD(io_readports_3_data[31:24]) (0.01)	
HD(io_readports_3_data[23:16]) (0.01)	
HD(io_readports_3_data[15:8]) (0.01)	
HD(io_readports_3_data[7:0]) (0.02)	
HD(io_readports_4_data[63:56]) (0.01)	
HD(io_readports_4_data[55:48]) (0.01)	
HD(io_readports_4_data[47:40]) (0.01)	
HD(io_readports_4_data[39:32]) (0.01)	
HD(io_readports_4_data[31:24]) (0.01)	
HD(io_readports_4_data[23:16]) (0.01)	
HD(io_readports_4_data[15:8]) (0.01)	
HD(io_readports_4_data[7:0]) (0.01)	
HD(io_readports_5_data[63:56]) (0.01)	
HD(io_readports_5_data[55:48]) (0.01)	
HD(io_readports_5_data[47:40]) (0.01)	
HD(io_readports_5_data[39:32]) (0.01)	
HD(io_readports_5_data[31:24]) (0.01)	
HD(io_readports_5_data[23:16]) (0.01)	
HD(io_readports_5_data[15:8]) (0.01)	
HD(io_readports_5_data[7:0]) (0.01)	
HD(io_write_ports_0_bits_data[63:56]) (0.11)	
HD(io_write_ports_0_bits_data[55:48]) (0.01)	
HD(io_write_ports_0_bits_data[47:40]) (0.78)	
HD(io_write_ports_0_bits_data[39:32]) (0.01)	
HD(io_write_ports_0_bits_data[31:24]) (0.01)	
HD(io_write_ports_0_bits_data[23:16]) (0.03)	

Features (importances)	Description
HD(io_write_ports_0_bits_data[15:8]) (0.01)	Write ports data interface
HD(io_write_ports_0_bits_data[7:0]) (0.02)	
HD(io_write_ports_1_bits_data[63:56]) (0.02)	
HD(io_write_ports_1_bits_data[55:48]) (0.01)	
HD(io_write_ports_1_bits_data[47:40]) (0.01)	
HD(io_write_ports_1_bits_data[39:32]) (0.01)	
HD(io_write_ports_1_bits_data[31:24]) (0.01)	
HD(io_write_ports_1_bits_data[23:16]) (0.01)	
HD(io_write_ports_1_bits_data[15:8]) (0.03)	
HD(io_write_ports_1_bits_data[7:0]) (0.01)	
HD(io_write_ports_2_bits_data[63:56]) (0.01)	
HD(io_write_ports_2_bits_data[55:48]) (0.01)	
HD(io_write_ports_2_bits_data[47:40]) (0.01)	
HD(io_write_ports_2_bits_data[39:32]) (0.01)	
HD(io_write_ports_2_bits_data[31:24]) (0.01)	
HD(io_write_ports_2_bits_data[23:16]) (0.01)	
HD(io_write_ports_2_bits_data[15:8]) (0.01)	
HD(io_write_ports_2_bits_data[7:0]) (0.01)	

Table 1.13: Iregister read

Features (importances)	Description
HD(io_rf_read_ports_0_data[63:56]) (0.01)	
HD(io_rf_read_ports_0_data[55:48]) (0.01)	
HD(io_rf_read_ports_0_data[47:40]) (0.01)	
HD(io_rf_read_ports_0_data[39:32]) (0.01)	
HD(io_rf_read_ports_0_data[31:24]) (0.01)	
HD(io_rf_read_ports_0_data[23:16]) (0.01)	
HD(io_rf_read_ports_0_data[15:8]) (0.01)	
HD(io_rf_read_ports_0_data[7:0]) (0.04)	
HD(io_rf_read_ports_1_data[63:56]) (0.01)	
HD(io_rf_read_ports_1_data[55:48]) (0.01)	
HD(io_rf_read_ports_1_data[47:40]) (0.01)	
HD(io_rf_read_ports_1_data[39:32]) (0.01)	
HD(io_rf_read_ports_1_data[31:24]) (0.01)	
HD(io_rf_read_ports_1_data[23:16]) (0.01)	

Features (importances)	**Description**
HD(io_rf_read_ports_1_data[15:8]) (0.01)	Register file read ports
HD(io_rf_read_ports_1_data[7:0]) (0.05)	
HD(io_rf_read_ports_2_data[63:56]) (0.01)	
HD(io_rf_read_ports_2_data[55:48]) (0.01)	
HD(io_rf_read_ports_2_data[47:40]) (0.01)	
HD(io_rf_read_ports_2_data[39:32]) (0.01)	
HD(io_rf_read_ports_2_data[31:24]) (0.01)	
HD(io_rf_read_ports_2_data[23:16]) (0.02)	
HD(io_rf_read_ports_2_data[15:8]) (0.01)	
HD(io_rf_read_ports_2_data[7:0]) (0.05)	
HD(io_rf_read_ports_3_data[63:56]) (0.01)	
HD(io_rf_read_ports_3_data[55:48]) (0.01)	
HD(io_rf_read_ports_3_data[47:40]) (0.01)	
HD(io_rf_read_ports_3_data[39:32]) (0.01)	
HD(io_rf_read_ports_3_data[31:24]) (0.01)	
HD(io_rf_read_ports_3_data[23:16]) (0.01)	
HD(io_rf_read_ports_3_data[15:8]) (0.01)	
HD(io_rf_read_ports_3_data[7:0]) (0.6)	
HD(io_rf_read_ports_4_data[63:56]) (0.01)	
HD(io_rf_read_ports_4_data[55:48]) (0.01)	
HD(io_rf_read_ports_4_data[47:40]) (0.01)	
HD(io_rf_read_ports_4_data[39:32]) (0.01)	
HD(io_rf_read_ports_4_data[31:24]) (0.01)	
HD(io_rf_read_ports_4_data[23:16]) (0.01)	
HD(io_rf_read_ports_4_data[15:8]) (0.01)	
HD(io_rf_read_ports_4_data[7:0]) (0.01)	
HD(io_rf_read_ports_5_data[63:56]) (0.01)	
HD(io_rf_read_ports_5_data[55:48]) (0.01)	
HD(io_rf_read_ports_5_data[47:40]) (0.01)	
HD(io_rf_read_ports_5_data[39:32]) (0.01)	
HD(io_rf_read_ports_5_data[31:24]) (0.01)	
HD(io_rf_read_ports_5_data[23:16]) (0.01)	
HD(io_rf_read_ports_5_data[15:8]) (0.01)	
HD(io_rf_read_ports_5_data[7:0]) (0.05)	
HD(io_bypass_data_0[63:56]) (0.01)	
HD(io_bypass_data_0[55:48]) (0.01)	

Features (importances)	Description
HD(io_bypass_data_0[47:40]) (0.01)	Bypass data interface
HD(io_bypass_data_0[39:32]) (0.01)	
HD(io_bypass_data_0[31:24]) (0.01)	
HD(io_bypass_data_0[23:16]) (0.01)	
HD(io_bypass_data_0[15:8]) (0.01)	
HD(io_bypass_data_0[7:0]) (0.12)	
HD(io_bypass_data_1[63:56]) (0.01)	
HD(io_bypass_data_1[55:48]) (0.01)	
HD(io_bypass_data_1[47:40]) (0.01)	
HD(io_bypass_data_1[39:32]) (0.01)	
HD(io_bypass_data_1[31:24]) (0.01)	
HD(io_bypass_data_1[23:16]) (0.01)	
HD(io_bypass_data_1[15:8]) (0.01)	
HD(io_bypass_data_1[7:0]) (0.01)	
HD(io_bypass_data_2[63:56]) (0.01)	
HD(io_bypass_data_2[55:48]) (0.01)	
HD(io_bypass_data_2[47:40]) (0.01)	
HD(io_bypass_data_2[39:32]) (0.01)	
HD(io_bypass_data_2[31:24]) (0.01)	
HD(io_bypass_data_2[23:16]) (0.01)	
HD(io_bypass_data_2[15:8]) (0.01)	
HD(io_bypass_data_2[7:0]) (0.01)	
HD(io_bypass_data_3[63:56]) (0.01)	
HD(io_bypass_data_3[55:48]) (0.01)	
HD(io_bypass_data_3[47:40]) (0.01)	
HD(io_bypass_data_3[39:32]) (0.01)	
HD(io_bypass_data_3[31:24]) (0.01)	
HD(io_bypass_data_3[23:16]) (0.01)	
HD(io_bypass_data_3[15:8]) (0.01)	
HD(io_bypass_data_3[7:0]) (0.02)	

Table 1.14: CSR features

Features (importances)	Description
HD(io_rw_wdata[63:56]) (0.01)	
HD(io_rw_wdata[55:48]) (0.07)	

Features (importances)	Description
HD(io_rw_wdata[47:40]) (0.01)	Write data
HD(io_rw_wdata[39:32]) (0.01)	
HD(io_rw_wdata[31:24]) (0.01)	
HD(io_rw_wdata[23:16]) (0.01)	
HD(io_rw_wdata[15:8]) (0.67)	
HD(io_rw_wdata[7:0]) (0.06)	
HD(io_decode_0_csr[11:0]) (0.15)	Instruction from decode stage
HD(io_decode_1_csr[11:0]) (0.04)	

Table 1.15: ALU features

Features (importances)	Description
HD(imul.inPipe_bits_in1[63:56]) (0.01)	Qualified, pipelined input of multiplier
HD(imul.inPipe_bits_in1[55:48]) (0)	
HD(imul.inPipe_bits_in1[47:40]) (0)	
HD(imul.inPipe_bits_in1[39:32]) (0)	
HD(imul.inPipe_bits_in1[31:24]) (0)	
HD(imul.inPipe_bits_in1[23:16]) (0)	
HD(imul.inPipe_bits_in1[15:8]) (0.06)	
HD(imul.inPipe_bits_in1[7:0]) (0.01)	
HD(imul.inPipe_bits_in2[63:56]) (0)	
HD(imul.inPipe_bits_in2[55:48]) (0)	
HD(imul.inPipe_bits_in2[47:40]) (0)	
HD(imul.inPipe_bits_in2[39:32]) (0)	
HD(imul.inPipe_bits_in2[31:24]) (0)	
HD(imul.inPipe_bits_in2[23:16]) (0)	
HD(imul.inPipe_bits_in2[15:8]) (0.01)	
HD(imul.inPipe_bits_in2[7:0]) (0.01)	
HD(alu.io_in2[63:56]) (0.01)	Qualified, pipelined input of ALU
HD(alu.io_in2[55:48]) (0.01)	
HD(alu.io_in2[47:40]) (0.01)	
HD(alu.io_in2[39:32]) (0.01)	
HD(alu.io_in2[31:24]) (0.01)	
HD(alu.io_in2[23:16]) (0.03)	
HD(alu.io_in2[15:8]) (0.01)	
HD(alu.io_in2[7:0]) (0.08)	

Features (importances)	Description
HD(alu.io_in1[63:56]) (0.01)	
HD(alu.io_in1[55:48]) (0.01)	
HD(alu.io_in1[47:40]) (0.02)	
HD(alu.io_in1[39:32]) (0.02)	
HD(alu.io_in1[31:24]) (0.07)	
HD(alu.io_in1[23:16]) (0.01)	
HD(alu.io_in1[15:8]) (0.01)	
HD(alu.io_in1[7:0]) (0.72)	
HD(ifpu.inPipe_bits_in1[63:56]) (0.01)	
HD(ifpu.inPipe_bits_in1[55:48]) (0.01)	
HD(ifpu.inPipe_bits_in1[47:40]) (0.01)	
HD(ifpu.inPipe_bits_in1[39:32]) (0.01)	Qualified and pipelined input of integer to floating point unit
HD(ifpu.inPipe_bits_in1[31:24]) (0.01)	
HD(ifpu.inPipe_bits_in1[23:16]) (0.01)	
HD(ifpu.inPipe_bits_in1[15:8]) (0.01)	
HD(ifpu.inPipe_bits_in1[7:0]) (0.01)	

Table 1.16: CSR Exe Unit features

Features (importances)	Description
HD(alu.io_in2[63:56]) (0.14)	
HD(alu.io_in2[55:48]) (0.01)	
HD(alu.io_in2[47:40]) (0.01)	
HD(alu.io_in2[39:32]) (0.01)	
HD(alu.io_in2[31:24]) (0.01)	
HD(alu.io_in2[23:16]) (0.01)	
HD(alu.io_in2[15:8]) (0.01)	
HD(alu.io_in2[7:0]) (0.75)	Qualified, pipelined input of ALU
HD(alu.io_in1[63:56]) (0.01)	
HD(alu.io_in1[55:48]) (0.01)	
HD(alu.io_in1[47:40]) (0.01)	
HD(alu.io_in1[39:32]) (0.01)	
HD(alu.io_in1[31:24]) (0.01)	
HD(alu.io_in1[23:16]) (0.01)	
HD(alu.io_in1[15:8]) (0.01)	
HD(alu.io_in1[7:0]) (0.02)	

Features (importances)	Description
HD(div.io_req_bits_in1[63:56]) (0.01)	Qualified, pipelined input of divider
HD(div.io_req_bits_in1[55:48]) (0.01)	
HD(div.io_req_bits_in1[47:40]) (0.01)	
HD(div.io_req_bits_in1[39:32]) (0.01)	
HD(div.io_req_bits_in1[31:24]) (0.01)	
HD(div.io_req_bits_in1[23:16]) (0.01)	
HD(div.io_req_bits_in1[15:8]) (0.01)	
HD(div.io_req_bits_in1[7:0]) (0.04)	
HD(div.io_req_bits_in2[63:56]) (0.01)	
HD(div.io_req_bits_in2[55:48]) (0.01)	
HD(div.io_req_bits_in2[47:40]) (0.01)	
HD(div.io_req_bits_in2[39:32]) (0.01)	
HD(div.io_req_bits_in2[31:24]) (0.01)	
HD(div.io_req_bits_in2[23:16]) (0.01)	
HD(div.io_req_bits_in2[15:8]) (0.01)	
HD(div.io_req_bits_in2[7:0]) (0.01)	

Table 1.17: FP Pipeline features

Features (importances)	Description
HD(fregfile.io_read_ports_0_addr[6:0]) (0.01)	
HD(fregfile.io_read_ports_0_data[64:56]) (0.01)	
HD(fregfile.io_read_ports_0_data[55:48]) (0.01)	
HD(fregfile.io_read_ports_0_data[47:40]) (0.01)	
HD(fregfile.io_read_ports_0_data[39:32]) (0.01)	
HD(fregfile.io_read_ports_0_data[31:24]) (0.01)	
HD(fregfile.io_read_ports_0_data[23:16]) (0.01)	
HD(fregfile.io_read_ports_0_data[15:8]) (0.01)	
HD(fregfile.io_read_ports_0_data[7:0]) (0.01)	
HD(fregfile.io_read_ports_1_addr[6:0]) (0.01)	
HD(fregfile.io_read_ports_1_data[64:56]) (0.01)	
HD(fregfile.io_read_ports_1_data[55:48]) (0.01)	
HD(fregfile.io_read_ports_1_data[47:40]) (0.01)	
HD(fregfile.io_read_ports_1_data[39:32]) (0.01)	
HD(fregfile.io_read_ports_1_data[31:24]) (0.01)	
HD(fregfile.io_read_ports_1_data[23:16]) (0.01)	

Features (importances)	**Description**
HD(fregfile.io_read_ports_1_data[15:8]) (0.01)	Floating point register file interface
HD(fregfile.io_read_ports_1_data[7:0]) (0.01)	
HD(fregfile.io_read_ports_2_addr[6:0]) (0.01)	
HD(fregfile.io_read_ports_2_data[64:56]) (0.01)	
HD(fregfile.io_read_ports_2_data[55:48]) (0.01)	
HD(fregfile.io_read_ports_2_data[47:40]) (0.01)	
HD(fregfile.io_read_ports_2_data[39:32]) (0.01)	
HD(fregfile.io_read_ports_2_data[31:24]) (0.01)	
HD(fregfile.io_read_ports_2_data[23:16]) (0.01)	
HD(fregfile.io_read_ports_2_data[15:8]) (0.01)	
HD(fregfile.io_read_ports_2_data[7:0]) (0.01)	
HD(fregfile.io_write_ports_0_bits_addr[6:0]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[64:56]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[55:48]) (0.07)	
HD(fregfile.io_write_ports_0_bits_data[47:40]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[39:32]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[31:24]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[23:16]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[15:8]) (0.01)	
HD(fregfile.io_write_ports_0_bits_data[7:0]) (0.01)	
HD(fregfile.io_write_ports_1_bits_addr[6:0]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[64:56]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[55:48]) (0.02)	
HD(fregfile.io_write_ports_1_bits_data[47:40]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[39:32]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[31:24]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[23:16]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[15:8]) (0.01)	
HD(fregfile.io_write_ports_1_bits_data[7:0]) (0.01)	
HD(fpu.io_req_bits_uop_uopc[8:0]) (0.02)	
HD(fpu.dfma.fma.io_a[64:56]) (0.01)	
HD(fpu.dfma.fma.io_a[55:48]) (0.01)	
HD(fpu.dfma.fma.io_a[47:40]) (0.03)	
HD(fpu.dfma.fma.io_a[39:32]) (0.01)	
HD(fpu.dfma.fma.io_a[31:24]) (0.01)	
HD(fpu.dfma.fma.io_a[23:16]) (0.01)	

Features (importances)	**Description**
HD(fpu.dfma.fma.io_a[15:8]) (0.01)	Qualified, pipelined inputs of fused multiply add unit
HD(fpu.dfma.fma.io_a[7:0]) (0.01)	
HD(fpu.dfma.fma.io_b[64:56]) (0.01)	
HD(fpu.dfma.fma.io_b[55:48]) (0.01)	
HD(fpu.dfma.fma.io_b[47:40]) (0.01)	
HD(fpu.dfma.fma.io_b[39:32]) (0.01)	
HD(fpu.dfma.fma.io_b[31:24]) (0.01)	
HD(fpu.dfma.fma.io_b[23:16]) (0.01)	
HD(fpu.dfma.fma.io_b[15:8]) (0.01)	
HD(fpu.dfma.fma.io_b[7:0]) (0.01)	
HD(fpu.dfma.fma.io_c[64:56]) (0.01)	
HD(fpu.dfma.fma.io_c[55:48]) (0.01)	
HD(fpu.dfma.fma.io_c[47:40]) (0.01)	
HD(fpu.dfma.fma.io_c[39:32]) (0.01)	
HD(fpu.dfma.fma.io_c[31:24]) (0.01)	
HD(fpu.dfma.fma.io_c[23:16]) (0.01)	
HD(fpu.dfma.fma.io_c[15:8]) (0.83)	
HD(fpu.dfma.fma.io_c[7:0]) (0.01)	
HD(fpu.fpiu.in_in1[64:56]) (0.01)	Qualified, pipelined input of floating point to integer unit
HD(fpu.fpiu.in_in1[55:48]) (0.01)	
HD(fpu.fpiu.in_in1[47:40]) (0.01)	
HD(fpu.fpiu.in_in1[39:32]) (0.01)	
HD(fpu.fpiu.in_in1[31:24]) (0.01)	
HD(fpu.fpiu.in_in1[23:16]) (0.01)	
HD(fpu.fpiu.in_in1[15:8]) (0.01)	
HD(fpu.fpiu.in_in1[7:0]) (0.01)	
HD(fpu.fpiu.in_in2[64:56]) (0.01)	
HD(fpu.fpiu.in_in2[55:48]) (0.01)	
HD(fpu.fpiu.in_in2[47:40]) (0.01)	
HD(fpu.fpiu.in_in2[39:32]) (0.01)	
HD(fpu.fpiu.in_in2[31:24]) (0.01)	
HD(fpu.fpiu.in_in2[23:16]) (0)	
HD(fpu.fpiu.in_in2[15:8]) (0)	
HD(fpu.fpiu.in_in2[7:0]) (0)	
HD(fpu.fpmu.inPipe_bits_in1[64:56]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in1[55:48]) (0.01)	

Features (importances)	Description
HD(fpu.fpmu.inPipe_bits_in1[47:40]) (0.01)	Qualified, pipelined input of fpmu
HD(fpu.fpmu.inPipe_bits_in1[39:32]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in1[31:24]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in1[23:16]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in1[15:8]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in1[7:0]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[64:56]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[55:48]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[47:40]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[39:32]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[31:24]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[23:16]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[15:8]) (0.01)	
HD(fpu.fpmu.inPipe_bits_in2[7:0]) (0.01)	

Table 1.18: LSU features

Features (importances)	Description
io_brinfo_mispredict (0.01)	Control signals - load, store mode and valid signals
io_brinfo_valid (0.01)	
io_dis_ld_vals_0 (0.05)	
io_dis_ld_vals_1 (0.01)	
io_dis_st_vals_0 (0.03)	
io_dis_st_vals_1 (0.01)	
io_exe_resp_bits_uop_ctrl_is_load (0.01)	
io_exe_resp_bits_uop_ctrl_is_sta (0.01)	
io_exe_resp_bits_uop_ctrl_is_std (0.01)	
io_exe_resp_bits_uop_fp_val (0.01)	
io_exe_resp_bits_uop_is_load (0.04)	
io_exe_resp_bits_uop_is_store (0.02)	
io_exe_resp_valid (0.01)	
io_forward_uop_fp_val (0.01)	
io_forward_uop_is_load (0.01)	
io_forward_uop_is_store (0)	
io_forward_val (0.01)	
io_fp_stdata_valid (0.01)	

Features (importances)	**Description**
io_mem_ldSpecWakeup_valid (0.02)	
io_memreq_uop_is_load (0.01)	
io_memreq_uop_is_store (0.01)	
io_memreq_val (0.02)	
io_memresp_bits_is_load (0.06)	
io_memresp_valid (0.1)	
HD(io_exe_resp_bits_addr[39:0]) (0.56)	Address and data interface
HD(io_exe_resp_bits_data[15:8]) (0.01)	
HD(io_exe_resp_bits_data[23:16]) (0.02)	
HD(io_exe_resp_bits_data[31:24]) (0.01)	
HD(io_exe_resp_bits_data[39:32]) (0.01)	
HD(io_exe_resp_bits_data[47:40]) (0.01)	
HD(io_exe_resp_bits_data[55:48]) (0.01)	
HD(io_exe_resp_bits_data[63:56]) (0.01)	
HD(io_exe_resp_bits_data[7:0]) (0.02)	
HD(io_forward_data[15:8]) (0.01)	
HD(io_forward_data[23:16]) (0.01)	
HD(io_forward_data[31:24]) (0.01)	
HD(io_forward_data[39:32]) (0.01)	
HD(io_forward_data[47:40]) (0.01)	
HD(io_forward_data[55:48]) (0.01)	
HD(io_forward_data[63:56]) (0.01)	
HD(io_forward_data[7:0]) (0.01)	
HD(io_fp_stdata_bits_data[15:8]) (0.01)	
HD(io_fp_stdata_bits_data[23:16]) (0.01)	
HD(io_fp_stdata_bits_data[31:24]) (0.01)	
HD(io_fp_stdata_bits_data[39:32]) (0.01)	
HD(io_fp_stdata_bits_data[47:40]) (0.02)	
HD(io_fp_stdata_bits_data[55:48]) (0.01)	
HD(io_fp_stdata_bits_data[63:56]) (0.01)	
HD(io_fp_stdata_bits_data[7:0]) (0.01)	
HD(io_memreq_wdata[15:8]) (0.01)	
HD(io_memreq_wdata[23:16]) (0.01)	
HD(io_memreq_wdata[31:24]) (0.01)	
HD(io_memreq_wdata[39:32]) (0.01)	
HD(io_memreq_wdata[47:40]) (0.01)	

Features (importances)	Description
HD(io_memreq_wdata[55:48]) (0.01)	
HD(io_memreq_wdata[63:56]) (0.01)	
HD(io_memreq_wdata[7:0]) (0.03)	

Table 1.19: ROB features

Features (importances)	Description
io_enq_valids_0 (0.42)	ROB enqueue interface
io_enq_valids_1 (0.04)	
HD(io_enq_uops_0_uopc[8:0]) (0.04)	
HD(io_enq_uops_1_uopc[8:0]) (0.26)	
io_brinfo_valid (0.02)	Branch related control signals
io_brinfo_mispredict (0.01)	
io_brinfo_btb_made_pred (0.01)	
io_brinfo_btb_mispredict (0.01)	
io_brinfo_bpd_made_pred (0.01)	
io_brinfo_bpd_mispredict (0.01)	
io_wb_resps_0_valid (0.02)	Writeback valid
io_wb_resps_1_valid (0.03)	
io_wb_resps_2_valid (0.01)	
io_wb_resps_3_valid (0.01)	
io_wb_resps_4_valid (0.02)	
io_lsu_clr_bsy_valid_0 (0.01)	Load store unit control signals
io_lsu_clr_bsy_valid_1 (0.01)	
io_lsu_clr_unsafe_valid (0.02)	
io_commit_valids_0 (0.04)	Commit interface
io_commit_valids_1 (0.03)	
HD(io_commit_uops_0_uopc[8:0]) (0.09)	
HD(io_commit_uops_1_uopc[8:0]) (0.02)	
io_commit_rbk_valids_0 (0.01)	
io_commit_rbk_valids_1 (0)	
io_commit_rollback (0.01)	
io_commit_st_mask_0 (0.01)	
io_commit_st_mask_1 (0.01)	
io_commit_ld_mask_0 (0.01)	
io_commit_ld_mask_1 (0.01)	

Bibliography

[1] A. Bogliolo, L. Benini, and G. De Micheli, "Regression-based RTL power modeling," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 3, Jul. 2000.

[2] D. Sunwoo, G. Y. Wu, N. A. Patil, and D. Chiou, "PrEsto: An FPGA-accelerated power estimation methodology for complex systems," in *International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2010.

[3] J. Yang, L. Ma, K. Zhao, Y. Cai, and T.-F. Ngai, "Early stage real-time SoC power estimation using RTL instrumentation," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2015.

[4] Y. Zhou, H. Ren, Y. Zhang, B. Keller, B. Khailany, and Z. Zhang, "PRIMAL: Power Inference using Machine Learning," in *Design Automation Conference (DAC)*, Jun. 2019.

[5] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing," *ACM Transaction on Architecture Code Optimization (TACO)*, vol. 10, no. 1, Apr. 2013.

[6] W. Lee, Y. Kim, J. H. Ryoo, D. Sunwoo, A. Gerstlauer, and L. K. John, "PowerTrain: A learning-based calibration of McPAT power models," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July. 2015.

[7] Y.-H. Park, S. Pasricha, F. J. Kurdahi, and N. Dutt, "A multi-granularity power modeling methodology for embedded processors," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 19, no. 4, Apr. 2011.

[8] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield, "New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors," *IBM Journal of Research and Development*, vol. 47, no. 5.6, Sep. 2003.

[9] D. Lee and A. Gerstlauer, "Learning-Based, Fine-Grain Power Modeling of System-Level Hardware IPs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 3, Feb. 2018.

[10] Synopsys, "PrimeTime," https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html.

[11] Ansys, "PowerArtist," https://www.ansys.com/products/semiconductors/ansys-powerartist.

[12] Mentor, "PowerPro RTL Low-Power," https://www.mentor.com/hls-lp/powerpro-rtl-low-power/.

[13] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *International-Symposium on Computer Architecture (ISCA)*, 2000.

[14] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. Eickemeyer, "Abstraction and microarchitecture scaling in early-stage power modeling," in *International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2011.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News (CAN)*, vol. 39, no. 2, Aug. 2011.

[16] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015.

[17] M. LeBeane, J. H. Ryoo, R. Panda, and L. K. John, "Watt Watcher: Fine-Grained Power Estimation for Emerging Workloads," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2015.

[18] D. Kim, J. Zhao, J. Bachrach, and K. Asanović, "Simmani: Runtime

Power Modeling for Arbitrary RTL with Automatic Signal Selection," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2019.

[19] W. L. Bircher and L. K. John, "Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events," in *International Symposium on Performance Analysis of Systems Software (ISPASS)*, Apr. 2007.

[20] T. Li and L. K. John, "Run-Time Modeling and Estimation of Operating System Power Consumption," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, Jun. 2003.

[21] Verilator, https://www.veripool.org/wiki/verilator.

[22] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: a full system simulator for multicore x86 CPUs," in *Design Automation Conference (DAC)*, Jun. 2011.

[23] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH computer architecture news (CAN)*, vol. 25, no. 3, Jun. 1997.

[24] M. Donno, A. Ivaldi, L. Benini, and E. Macii, "Clock-tree power optimization based on RTL clock-gating," in *Design Automation Conference (DAC)*, Jun. 2003.

[25] A. K. Ananda Kumar and A. Gerstlauer, "Learning-Based CPU Power Modeling," in *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, Sep. 2019.

[26] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "PULP: A parallel ultra low power platform for next generation IoT applications," in *IEEE Hot Chips 27 Symposium (HCS)*, Aug. 2015.

[27] RI5CY Documentation, https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf/, 2019.

[28] Nangate, "NanGate FreePDK45 Open Cell Library," http://www.nangate.com.

[29] Synopsys, "Design Compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html.

[30] ——, "VCS," https://www.synopsys.com/verification/simulation/vcs.html.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research (JMLR)*, vol. 12, 2011.

[32] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanović, "Boom v2: an open-source out-of-order risc-v core," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017.

[33] BOOM Documentation, https://docs.boom-core.org/, 2019.

[34] Cadence, "Innovus," https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html.

[35] RISC-V Tests., https://github.com/riscv/riscv-tests, RISCV Foundation.