Microprocessor Energy Characterization and Optimization through Fast, Accurate, and Flexible Simulation

by

Ronny Krashinsky

B.S. Electrical Engineering and Computer Science
University of California at Berkeley, 1999

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2001

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by . . . .
Krste Asanović
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Microprocessor Energy Characterization and Optimization through Fast, Accurate, and Flexible Simulation

by
Ronny Krashinsky

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Energy dissipation is emerging as a key constraint for both high-performance and embedded microprocessor designs, requiring computer architects to consider energy in addition to performance when evaluating design decisions. A major limitation is the general difficulty in analyzing the energy impact of architectural and microarchitectural features without constructing detailed implementations and running slow simulations. This thesis first describes the design of a fast, accurate, and flexible circuit simulation tool which enables transition-sensitive studies of microprocessor energy consumption that would otherwise be impossible or impractical. With a simulation infrastructure in place, various optimizations are implemented that target the entire datapath and cache energy consumption. The individual energy optimizations are analyzed in detail, and the microprocessor design is characterized using various energy breakdowns and studies of the bit correlation between data values. This work shows that a few relatively simple energy-saving techniques can have a large impact in the implementation of an energy-efficient microprocessor. By fully characterizing the energy usage, this thesis establishes a coherent vision of microprocessor energy consumption, and serves as a basis and motivation for further energy optimizations.

Thesis Supervisor: Krste Asanović
Title: Assistant Professor

Acknowledgments

First and foremost, I would like to extend my deepest gratitude to Krste for being such a dedicated advisor and great teacher. I would also like to thank the SCALE group at MIT for their endless help and occasional comic relief. Finally, I want to thank Maggie and my family for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Energy usage is increasingly a key constraint for microprocessor designs. Although once only a concern for the fastest supercomputers, energy is now important across a broad range of computer designs. For mobile systems, processor energy consumption is a limiting design factor in terms of battery weight and lifetime. High-performance microprocessors are constrained by peak power usage and the ability to supply current and dissipate the generated heat; these problems directly affect the maximum processor speed. Additionally, energy consumption is crucial in large server farms that are limited by the maximum capabilities of the power infrastructure as well as the cost of this energy. Traditionally, microprocessor designs have focused almost exclusively on performance, but these shifting constraints are requiring architects to consider energy in addition to performance when evaluating design decisions [34]. The new goal is to develop energy-efficient designs which simultaneously have high-performance and low energy consumption [3, 19].

The energy consumption for a system can be optimized at many levels. At the lowest level, improvements in process technology continually make all classes of digital circuits more energy-efficient. At the highest level, individual applications can be optimized to use energy more efficiently or even to do less work by exporting computation to a server. Many circuit level techniques have been explored which target individual structures such as adders, flip-flops, and caches. Some optimizations cross abstraction boundaries; for example, voltage scaling is an effective technique which reduces energy consumption by dynamically changing a circuit's supply voltage based on operating conditions [4]. In general low-level techniques have broader impact and applicability, while high-level techniques can achieve large savings for more limited domains. In creating an energy-efficient solution, it is important to apply optimizations at all levels in a system. This thesis focuses on energy-efficient microarchitectural design.

Microprocessor architectures have traditionally emphasized performance, with recent interest developing in energy-efficient designs. When making design decisions, it is important to have a clear picture of the processor energy consumption and to evaluate the effectiveness of power reduction techniques [20]. A major limitation is the lack of high-level tools to evaluate the energy impact of architectural and microarchitectural features. In Chapter 2, I describe the design of SyCHOSys, a circuit simulation framework which is simultaneously fast, accurate, and flexible. This tool enables detailed studies of microprocessor energy consumption based on full benchmark simulations that would otherwise be impossible or impractical. Chapter 3 describes the microarchitectural design of a basic five-stage pipeline MIPS RISC microprocessor. I implement this design using SyCHOSys including energy models for the caches and datapath components.

With a simulation infrastructure in place, Chapter 4 focuses on reducing energy at the microarchitectural level. Given efficient circuit structures, the challenge is arranging and controlling them to minimize energy usage. I implement various optimizations which target individual portions of the overall energy consumption. These techniques are generally simple, but have a large impact on the total energy usage; combined they reduce the datapath and cache energy by a factor of two. Chapter 5 characterizes and analyzes the microprocessor energy consumption. In this chapter I provide several detailed energy breakdowns which give a unified view of where and why energy is dissipated. I also provide a thorough analysis of the switching activity in the processor, and evaluate how it is affected by intermixing address and data values. The goal is to provide a basis for further energy optimizations, and to investigate factors which may limit the effort to reduce energy.

Chapter 2

Energy Simulation

Architectural studies of microprocessor energy consumption require fast and accurate simulation tools to evaluate a candidate design. Unfortunately, estimating energy dissipation is considerably more difficult than estimating performance. Given a complete functional description of a design, the only missing variable for achieving an accurate performance simulation is the clock frequency, and performance simulation (especially SimpleScalar [6]) has dominated research in computer architecture. However, a functional description of a design is not nearly adequate for energy simulation since every implementation detail impacts energy consumption.

Circuit simulation tools such as SPICE [36] and PowerMill [26] provide accurate energy numbers but require detailed implementations and run much too slowly to evaluate the effect of architectural modifications on large benchmark programs. A number of techniques have been proposed to estimate energy dissipation at higher levels of abstraction. One class of methods make use of *statistical* measures of circuit complexity and/or expected activity to estimate energy usage [21, 37]. Although these methods can represent average behavior and quickly provide estimates, they can give large errors for test inputs that do not match the modeled statistics, and cannot give cycle-by-cycle breakdowns of where energy was dissipated.

Recently, architectural power models have been developed which extend performance simulators with *activity-based* energy models that predict energy consumption based on usage counts and average energy costs for each functional block in a design (e.g., Wattch [1], TEM^{2}P^{2}EST [13], and ALPS [20]). The block energy costs can be based on average measurements for the design under simulation or a similar design. These models are attractive since they simulate energy at a high level of abstraction without requiring implementation details. However, inaccuracies can result if the simulated design differs from that on which the energy models are based [17]. Additionally, activity-based models do not account for the dependence of energy consumption on the actual signal transitions. Like statistical models, activity-based models are based on average behavior and may have large errors if the actual data values differ from those modeled; for this reason, they can not be used to model energy optimization techniques which depend on reducing signal activity. Even more alarming, the accuracy of the SimpleScalar performance simulator on which Wattch and TEM^{2}P^{2}EST are based has been called into question [12].

Transition-sensitive energy models measure the actual signal transitions caused by an input workload and use them to animate energy models [33]. This level of detail is necessary to accurately model circuit blocks whose energy consumption depends on signal activity; such circuits include almost all structures in a processor except for memory arrays implemented using differential circuits. Transition-sensitive simulators require a cycle-accurate structural model which tracks the values on all important nodes in the design. This is more detailed than traditional behavioral register-transfer-level (RTL) simulators which only model the register values each cycle. A structural model is necessary because behavioral models fundamentally lack the implementation details which are required to estimate energy. SimplePower [48] is a tool which enables transition-sensitive energy simulation by parameterizing an activity-based model. Every cycle, it uses a lookup table based on previous and current input values to determine each block's energy.

This chapter describes the implementation of a fast, flexible, and accurate energy simulation tool [29]. SyCHOSys (Synchronous Circuit Hardware Orchestration System) automatically compiles a structural netlist description of a design together with behavioral descriptions of the components and statistics gathering code into a cycle-accurate energy and performance simulator. Instead of basing energy consumption on usage counts for functional blocks or table lookups, SyCHOSys uses automatically generated bit transition statistics and arbitrary per-block statistics to drive transition-sensitive energy models which are based on the circuit structure and physical layout of each block. However, once these block models are established, architectural studies can arrange them in arbitrary configurations. SyCHOSys is used as the basis for the microprocessor energy studies in this thesis.

2.1 SyCHOSys Simulation

SyCHOSys generates cycle simulators from flattened structural netlists, as shown in Figure 2-1. A custom language describes the structural netlist, and C++ is used as the behavioral modeling language for netlist leaf cells. A cycle scheduler takes the structural netlist as input, and statically schedules evaluation of the behavioral blocks. It outputs C++ code containing calls to the blocks' behavioral methods. These calls are combined with the C++ component behavioral methods to form a SyCHOTick simulation object. Arbitrary additional simulator code instantiates a simulation object and drives the simulation by calling the statically scheduled evaluation methods. An optimizing C++ compiler (gcc 2.95.2 in this thesis) is used to generate the final SyCHOTick simulator executable.

To help explain the operation of SyCHOSys, I show a synchronous circuit in Figure 2-2 as an example. This circuit implements Euclid's greatest common divisor (GCD) algorithm shown in Figure 2-3.

2.1.1 Structural Netlist

The netlist representation of the circuit is shown in Figure 2-4. All nets in the design are declared at the top along with their bit-widths. Next, each component in the design is declared. A component declaration specifies the name of the component, the behavioral type of the component, and ordered lists of the component's inputs and outputs. Additionally, components such as flip-flops, latches, and dynamic logic which have clock-dependent be-

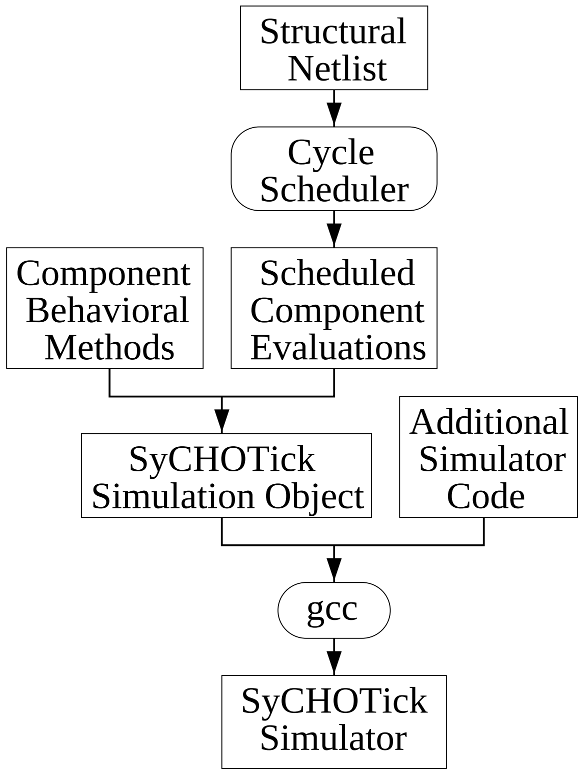

Structural Netlist
Cycle Scheduler
Component Behavioral Methods
Scheduled Component Evaluations
SyCHOTick Simulation Object
Additional Simulator Code
gcc
SyCHOTick Simulator

Figure 2-1: SyCHOSys framework.

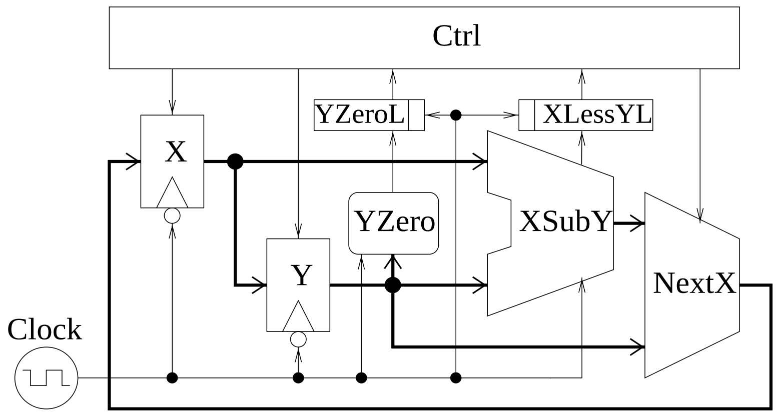


Figure 2-2: GCD circuit. `X` and `Y` are negative-edge-triggered flip-flops which receive enable signals from the control (`Ctrl`), and `YZeroL` and `XLessYL` are high-enabled latches. `YZero` is a dynamic zero comparator, and `XSubY` is a dynamic subtracter.

```
GCD(x, y) {
  if (x < y)     return GCD(y, x);
  else if (y!=0) return GCD(x-y, y);
  else           return x;
}
```

Figure 2-3: Euclid's greatest common divisor algorithm.

```
wire 32 x, y, nextx, xsuby;
wire xen, yen, xsel, zero, zero_l, less_l;

X       { N-CLK             FF_En<32> } (nextx, xen)     (x);
Y       { N-CLK             FF_En<32> } (x, yen)         (y);
NextX   {                   Mux2<32>  } (y, xsuby, xsel) (nextx);
XSubY   { (CLK-L.Precharge,
            CLK-H.Evaluate)  Sub<32>   } (x, y)           (xsuby);
YZero   { (CLK-L.Precharge,
            CLK-H.Evaluate)  Zero<32>  } (y)              (zero);
YZeroL  { H-Latch           Latch<1>  } (zero)           (zero_l);
XLessYL { H-Latch           Latch<1>  } (xsuby[31])      (less_l);
Ctrl    {                   GCDCtrl   } (zero_l, less_l) (xen, yen,
                                                          xsel);
```

Figure 2-4: SyCHOSys netlist for GCD circuit.

havior are tagged as such in the netlist. For example, X and Y have `N-CLK` tags since they are negative-edge-triggered flip-flops. Untagged components are assumed to be combinational logic blocks. Although not yet implemented, the netlist could be machine-generated from a hierarchical design description such as structural Verilog.

The SyCHOSys netlist format also allows bit-indexing into nets and net concatenation. To enable this, SyCHOSys processes the netlist and automatically inserts artificial components to extract and combine the nets as needed.

2.1.2 Behavioral Models

The behavioral models for components are in the form of C++ objects. Each C++ behavioral component defines an `Evaluate()` method which maps inputs to outputs. Figure 2-5 shows the Mux2 class. These methods can be parameterized using the C++ template mechanism, e.g. to accommodate variable bit-widths. When parameterized components are included in a netlist, the template parameters are specified, as with the Mux2 shown in Figure 2-4. Additionally, components can define more than one evaluation method; for example, dynamic logic components define a `Precharge()` method in addition to the `Evaluate()` method.

```
template<int bits>
class Mux2 : public Sycho_Block {
public:
  Mux2(){};
  inline void Evaluate(BitVec<bits>& input0, BitVec<bits>& input1,
                       BitVec<1>& select, BitVec<bits>& output) {
    if (select)  output = input1;
    else         output = input0;
  }
};
```

Figure 2-5: Example SyCHOSys behavioral model: Mux2. This component models a *bits* wide two input multiplexor.

Bit-vectors

In order to simplify the behavioral models, all nets in the design are implemented as bit-vector objects. This abstraction makes the models independent of the native machine data width, and allows for convenient arithmetic and logical operations on small and large bit-vectors that require special handling to mask overflows and handle carries. It also allows behavioral models to be parameterized based on the bit-width as in the mux example in Figure 2-5.

Since a simulation primarily consists of manipulating bit-vectors, the representation of these objects is very performance critical. For SyCHOSys, I modified a version of the C++ Standard Template Library `bitset` container [39] to support necessary arithmetic operations such as adds and shifts. An advantage of this template implementation is that the bit-widths are available at compile time and most of the overhead for providing a bit-vector abstraction is eliminated. For example, `bitset` is optimized so that the compiler uses only a single memory word to store bit-vector objects that are 32 bits or less, in contrast to implementations which must store the bit-width in addition to the value or which use a pointer to access an array of data. And, although manipulation of bit-vectors may require masking based on the bit-width, the compiler determines the necessary operations statically; for example, the masking operations are completely eliminated for operations on 32-bit bit-vectors.

2.1.3 Cycle Scheduler

A cycle-based schedule is determined for a netlist by constructing and topologically sorting dependency graphs. The dependencies between components are established based on the data dependencies and clock tags in the netlist. Each component can define up to four evaluation methods, one for each region of a clock period (rising, high, falling, low). For example, in Figure 2-4 the `N-CLK` tag of component X is shorthand for indicating that its `Evaluate()` method should be called during the clock-falling period. Table 2.1 shows how the methods are defined for the standard clock tags. Additionally, the table

	Rising	High	Falling	Low
Combinational		Evaluate		Evaluate
P-CLK	Evaluate			
N-CLK			Evaluate	
H-LATCH		Evaluate		
L-LATCH				Evaluate
Dynamic Adder		Evaluate		Precharge
RegFile		Read	Write	

Table 2.1: Evaluation methods defined for various component types for each region of a clock period.

shows how components can define arbitrary methods, such as a register file which defines `Read()` and `Write()` methods. These are declared explicitly in the netlist; for example, in the Figure 2-4 the declaration for the dynamic subtracter (`XSubY`) indicates that its `Precharge()` method should be called while the clock is low and its `Evaluate()` method should be called while the clock is high. All of these methods map inputs to outputs and are supplied in the behavioral description of the component.

Based on the clock tags in the netlist, the scheduler constructs four graphs, one for each region of the clock period. These are shown for the GCD circuit in Figure 2-6. Only the components with an evaluation method defined for a given clock region are part of the associated scheduling graph. During the level regions of the clock period (high and low), if component B gets an input from component A, then A should evaluate before B $(A \rightarrow B)$. During the edge regions of the clock period (rising and falling), if component B gets an input from component A, then B should evaluate before A $(A \leftarrow B)$.

The reverse ordering for clock edges prevents a value from propagating through a chain of flip-flops in one clock cycle. An alternative solution adopted in many RTL simulators is to use a temporary storage location for every output of edge-clocked components. The components are evaluated in an arbitrary order at the clock edge, but they only update the temporary location so that values do not propagate through multiple components. After all the edge-clocked components have evaluated, the temporary values are copied into the actual outputs which are used by the other circuit components. The method adopted in SyCHOSys uses static scheduling to eliminate the overhead of accessing temporary storage locations; however, it has the drawback that a closed chain of edge-triggered components has no possible correct evaluation ordering. To fix the problem, a single combinational component (e.g. a buffer) must be inserted to break the cycle.

To determine a correct cycle-based scheduling, each dependency graph is topologically sorted using a depth-first search (DFS) algorithm. Any combinational logic cycles are detected and reported as an error at this stage. In general, all combinational logic blocks will be evaluated during both the clock-high and clock-low periods. However, I implement an optimization which prunes the scheduling graphs so that a combinational component is only evaluated during clock periods in which at least one of its inputs may change. This is accomplished while topologically sorting the scheduling graphs by excluding certain components from the base set of vertices used as starting points for the DFS algorithm: in

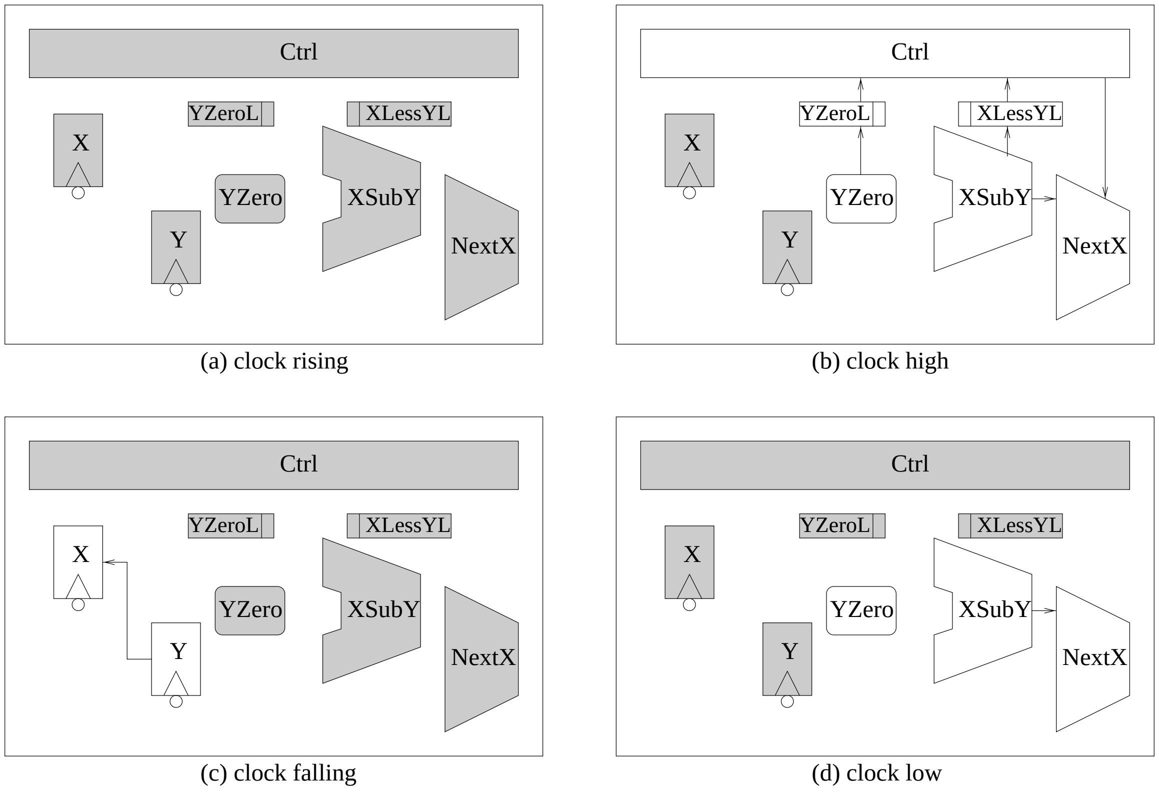


Figure 2-6: Scheduling graphs for the GCD circuit. The arrows indicate scheduling dependencies, and the shaded components are not scheduled for evaluation.

the clock-high graph, combinational components which do not get inputs from rising-edge-clocked components are excluded, and similarly in the clock-low graph with falling-edge-clocked components. This is why `Ctrl` is not scheduled in Figure 2-6(d).

Figure 2-7 shows the scheduled component evaluation functions which result from topologically sorting the graphs in Figure 2-6.

2.1.4 SyCHOTick

A SyCHOTick simulation object is constructed by combining the scheduled component evaluation calls with the component behavioral models. The main interface is a `clock_tick()` method which calls each of the clock region evaluation methods in turn. SyCHOTick simulation objects also provide convenient methods to print out the values of nets each cycle or output a VCD (value change dump) trace [27].

Additional application-specific C++ code instantiates the SyCHOTick simulation object and drives the simulation. It also provides a user interface to the simulator which may include such things as initializing the circuit and providing debugging output. A powerful feature of SyCHOSys is that arbitrary C++ code can be included with the simulator. For example, the CPU simulator includes code to first load a program into memory and then to service Unix system I/O calls on the simulated program's behalf. All nets and component blocks in a design are public members of the simulation object, with the same name as in

```
SychoTick_GCD::clock_rising() {
}

SychoTick_GCD::clock_high() {
    YZero.Evaluate(y, zero);
    YZeroL.Evaluate(zero, zero_l);
    XSubY.Evaluate(x, y, xsuby);
    xsuby_31_31_sb_BitSlice.Evaluate(xsuby, xsuby_31_31);
    XLessYL.Evaluate(xsuby_31_31, less_l);
    Ctrl.Evaluate(zero_l, less_l, xen, yen, xsel);
    NextX.Evaluate(y, xsuby, xsel, nextx);
}

SychoTick_GCD::clock_falling() {
    Y.Evaluate(x, yen, y);
    X.Evaluate(nextx, xen, x);
}

SychoTick_GCD::clock_low() {
    YZero.Precharge(y, zero);
    XSubY.Precharge(x, y, xsuby);
    xsuby_31_31_sb_BitSlice.Evaluate(xsuby, xsuby_31_31);
    NextX.Evaluate(y, xsuby, xsel, nextx);
}
```

Figure 2-7: Scheduled component evaluation calls for the GCD circuit. Note that the component `xsuby_31_31_sb_BitSlice` has been inserted automatically by SyCHOSys to extract bit 31 from `xsuby`.

the netlist. This provides easy access for the driver code, and an interface which is much easier to use and more efficient than a Verilog/PLI interface.

The final simulator is compiled using an optimizing C++ compiler which inlines the component evaluation calls. The representation of a simulation object is designed to leverage the compiler's optimization capabilities. All nets in the design are represented by bit-vector objects, and these are declared as members of the simulation object. Once the code for the behavioral evaluation methods is inlined, accesses to the bit-vector objects turn into simple memory references which get at the net values with no overhead. For example, the evaluation method for a flip-flop will turn into a load followed by a store. In some cases the compiler is even able to eliminate the loads of the bit-vector values by retaining them in registers across behavioral evaluation methods. Additionally, declaring all the bit-vector objects together in the simulation object results in high spatial-locality during simulation.

2.2 Energy Models

Developing models for energy dissipation involves a tradeoff between simulation speed and energy accuracy. SyCHOSys is designed to gather energy numbers for large benchmark programs requiring perhaps billions of execution cycles; this is only tractable with very simple runtime statistics gathering. This section describes how SyCHOSys takes advantage of the limited application domain of well-designed high-performance low-power microprocessors to increase the accuracy of simple transition-sensitive energy models.

The three major sources of power dissipation in a digital CMOS circuit are summarized by [10]:

$$P_{total} = aC_l V_{swing} V_{dd} f + I_{sc} V_{dd} + I_{leakage} V_{dd}$$

where the first term is the dynamic switching component of the energy, the second term is from short-circuit currents, and the last term is from leakage currents including diode and sub-threshold leakage.

Dynamic switching is the primary source of energy dissipation in CMOS circuits, where a is the average number of positive-edge transitions per cycle (the activity factor), C_l is the effective load capacitance, V_{swing} is the signal swing on the node, which is often equal to V_{dd}, the supply voltage, and f is the clock frequency. Short circuit current is generated during signal transients as both NMOS and PMOS transistors turn on simultaneously in a static CMOS gate, and is a function of signal rise-fall times and circuit state. In well-designed circuits, short circuit energy should be less than 10% of dynamic switching energy. During active operation, the leakage currents, $I_{leakage}$, are usually much smaller than the other terms and can be neglected. Signal swing, clock frequency, and supply voltage can usually be treated as fixed quantities. The difficult part of accurate energy modeling with a cycle simulator is modeling dynamic signal activity and effective load capacitance.

A limitation of a cycle simulator is that it does not model transient glitches which can cause additional power dissipation. The SyCHOSys simulator is only capable of modeling up to two transitions per cycle for clock-controlled circuitry such as precharged dynamic logic. In a well-designed low-power processor, however, glitch energy should be minimal. Dynamic circuit blocks, such as the adder, must avoid glitches for correct operation; and, control lines are usually registered to avoid glitches and driver conflicts in tristate busses. Glitches are more of a concern in larger datapath units such as multipliers and shifters, and in control logic blocks. For these blocks, each unit's energy model can estimate glitch activity based on input values. One factor which can help in accounting for glitch energy is that glitches are often caused by transition activity, so some glitch energy can be estimated as a percentage of the switching energy.

2.2.1 Dynamic Switching Energy

Microprocessors can be split into three main types of circuit blocks — memory arrays, datapaths, and control logic — connected together by global wires. The energy dissipation on global wires can be determined solely by the total capacitance and transition frequency. For the various circuit blocks, the task is simplified by considering each type of component separately.

Memory Arrays

Memory arrays are one of the largest consumers of energy in a typical microprocessor but are straightforward to model because of their regular layout structure and simple activation pattern. Once calibrated with a few test patterns, a cycle by cycle address and data trace is sufficient to model the energy dissipation of a large memory array to acceptable accuracy. For example, the low-power cache design used in this thesis employs multiple levels of cache sub-banking and self-timed low-voltage-swing techniques, yet the cycle-based energy model captures energy dissipation to within 3% of SPICE simulation based on extracted layout.

Datapaths

Datapaths also consume considerable energy in a processor but are more complicated to model because of their complex interconnect structure, the wide variety of circuit types, and the richer set of activation patterns. The energy dissipated in a unit is a function of the input and output signal statistics and the effective internal load capacitances. The internal capacitances are independent of a specific datapath floorplan, and can be determined once when the unit is designed. The energy dissipated on the nets between datapath components is modeled based on the capacitance and switching frequency, just as with global wires.

The most common cells in a processor datapath are muxes and latches, and for these functions transmission gate designs, as shown in Figure 2-8, have among the best overall energy-delay product while also being simple and robust [41]. Coincidentally, these cells have properties that enable simple but accurate energy modeling; their internal energy consumption can be estimated with only the transition counts at their inputs and outputs. For example, node IA in the mux circuit transitions as often as input node A, and node IF transitions as often as output node F. Similarly, in the latch circuit, node ID transitions as often as node D, and nodes IQ and IQB transition as often as the output Q.

More complex datapath blocks have custom energy models derived from their internal structure. For example, for the carry-skip adder used in this thesis, the bitwise XOR and

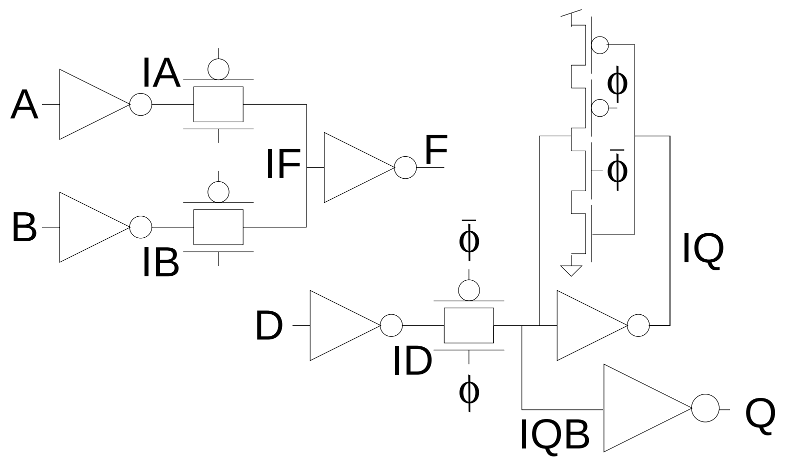


Figure 2-8: Transmission gate mux and latch designs.

AND of the input vectors determine the values for the propagate and generate gate outputs on a per-cycle basis, while the carry chain values are determined by XOR-ing the adder output with the internal propagate value. The carry skip values are computed from the individual carry bits. By using bit-parallel arithmetic in this way, the switching on all internal nodes in the adder can be calculated rapidly.

Control Logic

Control logic also has a complex structure (often automatically synthesized, placed, and routed) and a rich set of activation patterns, but is usually fashioned from a small set of static CMOS standard cells. For simple pipelined RISC and VLIW processors, control logic is usually responsible for less than 10% of total processor energy [5], particularly if energy expended in control line drivers is accounted for in the datapath components. Modeling control logic energy consumption accurately will become increasingly important in low-power processor designs because many techniques (e.g., clock gating or eliminating cache tag checks) reduce datapath and cache energy but require additional control logic. This thesis does not include modeling of control logic energy, but planned future work will add support in SyCHOSys for gate-level modeling of standard cell control blocks with load capacitances extracted from placed and routed output.

Effective Load Capacitance

The energy models for global wires and datapaths multiply transition counts with corresponding effective load capacitance values. The capacitance values can either be extracted from layout or estimated using some interconnect length model if layout is not available. For this thesis, circuit extraction is performed using the SPACE 2D extractor [45] which extracts layout parasitics including capacitance to the substrate, fringe capacitance, capacitance between parallel wires, and crossover wire capacitance. SPACE produces a SPICE netlist of wire caps plus transistors.

A custom tool, `mergecap`, reads the extracted SPICE netlist and for each net returns a single equivalent capacitance to ground obtained by summing all capacitances connected to the net. As part of this process, the effective capacitance contribution of any transistor gates or drains connected to the net are estimated. These capacitances can be difficult to model because they are voltage dependent. However, in a well-designed circuit, signal rise-fall times are usually restricted to a narrow range around the natural fanout-of-four (FO4) rise-fall times; therefore the effective transistor gate and drain capacitances can be determined by constructing circuits with typical rise-fall times. The coupling capacitance between two nets also varies dynamically depending on their relative switching. A cycle simulator cannot determine the relative timing of two signals, and even if that were possible, tracking inter-signal interactions would require excessive compute time and storage. `Mergecap` makes the approximation that two coupled signals never switch simultaneously, and simply sums all inter-node capacitances into a single equivalent capacitance to ground.

2.3 SyCHO Energy Analysis

Conventional RTL simulations only need to accurately model the register values each cycle. One motivation for specifying a SyCHOSys design as a structural netlist of components is to enable cycle-accurate simulation for all inter-component net values in the design. As described above, SyCHOSys generates a simulation that accurately tracks the input and output values of each block. These values are valid for each period of the clock; this includes nodes which take on two distinct values per clock period, for example the output of a dynamic logic block and any combinational logic which is connected to it.

The structural nature of the netlist description simplifies the process of energy estimation. I divide the problem into two parts, the energy dissipated on the nets which connect components together, and the internal energy used by each component. To determine the energy of the external nets, counters are added into the simulation code to keep track of the total transitions for each bit; these counters are generated automatically based on the structural netlist description. Energy is calculated using this number and the capacitance of the node as extracted from layout or estimated.

Each component is responsible for calculating its own internal energy. A `CalcStats()` method can be defined in the behavioral component model; this method is called after every `Evaluate()`. Based on its new inputs and outputs, the block calculates any internal statistics which it needs to determine energy consumption. For example, an adder might count the transitions in each portion of the carry chain, while a register file block could count the total number of reads and writes. In addition to defining a statistics gathering routine, each component type defines a method which interprets these statistics to determine energy.

As described above, the internal energy usage of many components can be calculated based only on capacitance numbers and the transition counts of their input and output nodes. Since the transitions on all external nets are counted automatically, these are made available to components for their internal energy calculation routines. This avoids duplicating the statistics tracking on the external net and in the internal statistics gathering of the component. Additionally, when two or more components share a common input, the transitions are only counted once but both components can make use of the numbers in determining their internal energy usage.

In order to minimize simulation time, SyCHOSys only tracks switching statistics while running a simulation. All of the remaining energy analysis happens as a data processing step after the statistics have been gathered. This is in contrast to simulators like Simple-Power [48] and Wattch [1] which calculate energy usage every cycle. Usually, the total energy numbers for a simulation are most useful, but SyCHOSys could output snap-shots of the statistics as often as every cycle to give more fine-grained energy numbers. Another feature of the SyCHOSys design is that energy statistics can be gathered for a subset of the components or a selected subset of simulation time to improve simulation speeds.

2.3.1 Transition Counting

Rapid signal transition counting is at the heart of fast energy-performance simulation. The transitions on a bus are determined by the logical XOR of the current and previous values; accumulating a count of the number of ones for each bit in this value gives the total bus activity. A simple method to count bus transitions is to use a series of shifts, masks, and adds to accumulate each bit separately as in the following code:

```
trans = val1 ^ val2;
if (trans==0) return; // optimization
for (i=0; i<n; i++) {
  bit_count[i] += ((trans >> i) & 0x1);
}
```

The check for zero is an optimization so that the rest of the loop can be avoided. The memory layout for this scheme is shown in Figure 2-9 by the horizontal boxes.

To improve simulation speed, SyCHOSys maintains transition counts for each bit in the bus using an alternative transposed memory layout (the vertical boxes in Figure 2-9) and uses the following faster bit-parallel ripple carry algorithm:

```
carries = val1 ^ val2;
for (i=0; carries != 0; i++) {
  temp = sig_bit[i];
  sig_bit[i] ^= carries;
  carries &= temp;
}
```

This method has the advantage of terminating as soon as `carries` becomes zero, but has the disadvantage that it can waste memory for narrow busses. To alleviate this effect, SyCHOSys handles single-bit nets using a single memory location and simple transition counting. When taking energy statistics for the GCD example, using the transposed counting method increased the overall simulation speed by a factor of four over the simple method, and a factor of two over the simple method with the additional zero check.

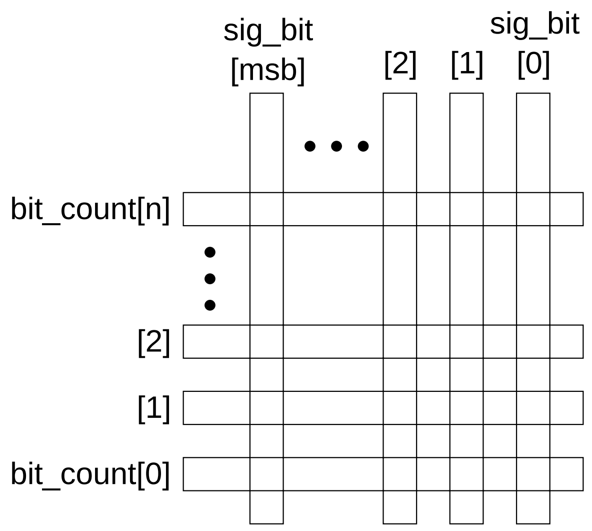


Figure 2-9: Alternative memory layouts for counting the bit transitions of an n-bit bus.

2.4 Energy-Performance Model Evaluation

To evaluate the performance and accuracy of the SyCHOSys datapath energy models, the GCD circuit was implemented using a TSMC 0.25 μm process. The circuit contains a mixture of various datapath units including flip-flops, latches, muxes, and adders, and both static and dynamic logic.

This circuit was modeled with several different simulators as shown in Table 2.2. The hand-tuned C code (an iterative version of Figure 2-3) runs extremely quickly, taking only three processor cycles per loop iteration. A comparable behavioral Verilog simulation is around 200 times slower. I also implemented a structural Verilog simulation which wires together components such as the registers, subtracter, and mux, which were modeled as separate behavioral blocks; this version ran around 30% slower than the behavioral design. The SyCHOSys structural simulation models the design at a comparable level to the structural Verilog design, yet runs over 20 times faster. When the energy statistics tracking is added to the SyCHOSys simulator, a 40-fold slow down is observed. However, the SyCHOSys energy simulator is still 7 orders of magnitude faster than an HSpice simulation, and 5 orders of magnitude faster than PowerMill.

The SyCHOSys energy simulation kept statistics for a total of 300 nodes (bits), including 4 32-bit external busses, 8 1-bit external signals, 5 32-bit internal busses for the subtracter, and 4 1-bit internal signals for the control. The total number of nodes in the design layout was 1278; the energy equations account for the switching activity on almost all of these nodes by taking advantage of the fact that the transition frequencies of internal nodes often mirror those of the input or output nodes.

Table 2.3 shows a comparison of energy usage as calculated by HSpice, PowerMill, and SyCHOSys. Seven input vectors were chosen to exercise the GCD circuit differently. The results illustrate the importance of transition-sensitive modeling, e.g., the difference in data values causes the first case to dissipate almost twice the energy of the second case even though both complete the GCD calculation in the same number of cycles. The SyCHOSys

Simulation model	Compiler / Simulation Engine	Simulation Speed (Hz)
C-Behavioral	gcc -O3	109,000,000.00
Verilog-Behavioral	VCS -O3 +2+state	544,000.00
Verilog-Structural	VCS -O3 +2+state	341,000.00
SyCHOSys-Structural	gcc -O3	8,000,000.00
SyCHOSys-Energy	gcc -O3	195,000.00
Extracted Layout	PowerMill	0.73
Extracted Layout	Star-HSpice	0.01

Table 2.2: Comparison of various simulation speeds for the GCD circuit. All simulations were run on a 333 MHz Sun Ultra-5 workstation under Solaris 2.7.

GCD inputs		GCD cycles	HSpice	Power-Mill	SyCHO-Sys
X	`0x04000000`	18	0.946	0.8163	0.9560
Y	`0x40000000`			(-13.7%)	(+1.06%)
GCD	`0x04000000`				
X	`0x00ffffff`	18	0.555	0.5211	0.5444
Y	`0x0ffffff0`			(-6.11%)	(-1.91%)
GCD	`0x00ffffff`				
X	`0x05555555`	22	1.095	1.001	1.021
Y	`0x6aaaaaa4`			(-8.58%)	(-6.76%)
GCD	`0x05555555`				
X	`0x0487ab00`	26	1.198	1.102	1.195
Y	`0x3b9aca00`			(-8.01%)	(-0.250%)
GCD	`0x003d0900`				
X	`0x01fffffe`	45	1.267	1.158	1.236
Y	`0x50ffffaf`			(-8.63%)	(-2.48%)
GCD	`0x00ffffff`				
X	`0x053ec600`	46	2.059	1.910	2.125
Y	`0x34f7e020`			(-7.24%)	(+3.21%)
GCD	`0x00004e20`				
X	`0x01000000`	66	3.266	2.953	3.494
Y	`0x40000000`			(-9.58%)	(+6.98%)
GCD	`0x01000000`				

Table 2.3: Comparison of simulated energy usage for several GCD computations using various energy simulators. All energy numbers are in nJ. The percent differences from HSpice are shown for PowerMill and SyCHOSys.

energy model differed from HSpice by a maximum of 7%. In all cases, it gave numbers within PowerMill's error; although it should be noted that PowerMill had a lower variance in error.

2.5 Processor Model Development

The main goal in developing SyCHOSys is to aid research in the design of low power processors. To this end, I have been using the tool to model a MIPS R3000-compatible RISC microprocessor with a five-stage pipeline described further in the next chapter. SyCHOSys enables a top-down design methodology in which one can successively refine a working design. In the following, simulation speeds are for tests run on a 333 MHz Sun Ultra-5 workstation running Solaris 2.7.

As an initial step, I designed a behavioral RTL simulator using SyCHOSys. This design modeled each pipeline stage as a large behavioral block, and was cycle-accurate at the registers between stages. The memory was modeled as a "magic" memory which performed all operations in a single cycle. I compiled this design using the SyCHOSys framework, and achieved a simulation speed of around 400 kHz. Comparable Verilog models compiled using industry tools run at 3 kHz to 35 kHz depending on the tool and the level of optimization. For reference, a C++ ISA interpreter which does not model the pipeline state or cache runs at about 3 MHz.

The next step was to break the behavioral blocks up into adders, muxes, shifters, etc. The control was still modeled as a large behavioral block, and the design still used a "magic" memory and a "magic" multiplier/divider unit with a single cycle latency. This design contained around 135 components in the netlist, and ran at a speed of 280 kHz. This simulation is cycle-accurate at each block used in the design, yet the performance is still comparable to an RTL simulation, and far better than compiled behavioral Verilog simulations. This is evidence of the speed advantages of statically scheduling a cycle-based simulation and leveraging a powerful general purpose compiler.

The processor simulation used in this thesis includes instruction and data caches and a cycle-accurate model of the off-chip memory system. I have also implemented an iterative multiplier/divider. The processor pipeline supports user/kernel mode and precise exception handling. On the 333 MHz Sun workstation the simulation runs at around 40 kHz, and executes large benchmark programs. When the energy statistics tracking is enabled, the simulation rate is around 11 kHz while tracking transition statistics for over 5000 bits. Note that the slowdown to include energy modeling is much less for the processor than for the GCD circuit because the CPU model complexity is much greater relative to the number of nodes that need to be tracked. This simulation rate is adequate for simulating a billion cycles in a CPU day.

Most of the simulations for this thesis were run on a set of 10 dual processor 850 MHz Pentium III systems (note that each simulation runs on only one processor). On these, the processor performance simulator runs at 130 kHz and the energy simulations run at 45 kHz. There are some know inefficiencies which could be optimized, for example some of the net transition counts could be eliminated, but so far this has not been necessary. The energy simulation rate is adequate for simulating almost four billion cycles in a CPU day.

2.6 Processor Energy Models

SyCHOSys allows considerable flexibility in developing the component energy models. For the GCD evaluation in Section 2.4, the energy models were based on extracted and merged capacitance values as described in Section 2.2.1. These are relatively easy to generate, and they can be calibrated to accurately represent average (e.g. fanout-of-four) switching characteristics [23]. However, the energy models for many components can be improved if they are based on SPICE simulations.

For the microprocessor simulations discussed in this thesis, SPICE simulations are used to characterize many of the components, including the caches, register file, flip-flops, latches, and muxes. The caches use analytical models which determine an energy cost for the tag check, data read or write, and data bus; the address bus energy depends on transition statistics. The register file uses a transition-sensitive model for the bit-lines; while the energy dissipated on the word-line, precharge, and enable signals is based on the total number of reads and writes. Although initially based on extracted and merged capacitance values, the models for the flip-flops, latches, and muxes have been updated to use energy numbers based on SPICE simulations. The models are simple to generate since these components are bit-independent [48]; that is, the operation of each bit-slice is independent of the other bit-slices. To obtain the energy models, a single-bit structure is simulated in detail to determine an effective energy cost for transitions on the input, output, and clock/select signals. The only component energy models in the microprocessor simulation which are not derived from SPICE simulations are the ALU, shifter, and adders; based on the simulated numbers presented in the following chapters, these components are estimated to account for only around 10% of the total energy.

The component energy models used in SyCHOSys are independent of any particular configuration, so the components can be arranged to model arbitrary microprocessor organizations. Additionally, the component energy models developed for this thesis are parameterized based on bit-width when possible; this is the case for the flip-flops, latches, muxes, and adders. Future work will also parameterize the cache and register file energy models. Since the component energy models are configuration-independent and parameterizable, it is relatively straightforward to evaluate alternative microprocessor designs using an established library of components.

2.7 Summary

In this chapter I described a framework that enables fast, accurate, and flexible simulation of energy usage in microprocessor designs.

SyCHOSys is fast since it statically schedules component evaluations and leverages compile-time optimization. The use of a separate netlist description as in [8] simplifies cycle scheduling by making the timing for the synchronous circuit explicit. Static scheduling eliminates much of the overhead associated with event-driven simulators which are typically used to model circuits. The framework is designed so that the compiler can inline the component evaluation methods and eliminate much of the overhead associated with accessing the net values that are represented as generic bit-vector objects. For energy simulation, SyCHOSys maintains minimal transition counts at run-time with carefully tuned code. Using SyCHOSys, a detailed pipelined MIPS processor model runs at 45 kHz while tracking statistics sufficient to model almost all datapath and memory energy consumption.

SyCHOSys provides accurate energy simulation since it uses transition-sensitive models based on extracted capacitance values or simulated SPICE measurements. Since the energy models are driven by the simulated transition counts, there is no need to make any assumptions about average switching statistics when developing the models. The block models themselves can be based on detailed SPICE simulations if they are independent of input transitions, as with differential memory structures; or if they have have simple energy models, as with flip-flops, latches, and muxes. More complex structures can be based on extracted and merged effective node switching capacitances which assume average fanout-of-four characteristics. For memory structures, the models are within 3% of SPICE simulation based on extracted layout, and a simulation of an example datapath circuit was found to be within 7% of SPICE energy estimates.

SyCHOSys is flexible in that it supports simulation at all levels of detail from purely behavioral to gate level. This enables a top-down design methodology in which functionality is at first modeled using behavioral code, then refined into a a structural implementation. In addition, rather than generate a closed stand-alone simulator, SyCHOSys produces a C++ object that can itself be linked with arbitrary C++ code. In general, circuits require an external interface, and this organization simplifies the task of driving the simulation. The separate netlist description allows SyCHOSys to automatically handle the scheduling overheads involved in writing and maintaining a hand-written C/C++ simulator. Additionally, the netlist allows SyCHOSys to generate much of the energy statistics gathering code automatically.

Chapter 3

Vanilla Pekoe Microprocessor Microarchitecture

This chapter describes the microarchitectural design of the Vanilla Pekoe processor. A SyCHOSys simulation of Vanilla Pekoe is used as the basis for the energy studies in this thesis.

Vanilla Pekoe is an energy-efficient processor being built by the SCALE group [25] at the Laboratory for Computer Science at MIT. It is a single-issue five-stage pipeline RISC processor which implements the MIPS-II instruction set [28]. Vanilla Pekoe is intended to serve as a base-line general-purpose processor design against which more radical processor architectures can be compared. Its design point is analogous to the well-known StrongARM microprocessor [14].

One of the most important goals in creating an energy-efficient design is to have high performance [3, 19]. It is easy to lower energy consumption at the expense of performance, e.g., by lowering a circuit's supply voltage. However, any interesting application will demand both high performance and low energy. A processor design can optimize the execution of a program in two ways: by reducing the CPI (cycles-per-instruction), or by increasing the clock frequency (cycles-per-second).

In order to optimize CPI, Vanilla Pekoe is pipelined and can execute up to one instruction per cycle. Additionally, bypassing avoids stalls by making results in the pipeline available to dependent instructions, and branch-prediction attempts to keep the pipeline filled with useful instructions. Highly associative single-cycle instruction and data caches are used to provide fast access times and high hit rates so that fewer cycles are spent waiting for instructions and data. CPI can be further improved by executing multiple instructions in a single clock cycle, as superscalar architectures do. However, these designs require vast amounts of speculation and additional complexity to achieve relatively modest performance improvements; this is detrimental to the overall energy-efficiency [19].

The Vanilla Pekoe pipeline is optimized for speed by incorporating fast circuit structures and microarchitecture design. Critical components such as the ALU, register file, and caches use high-speed dynamic and self-timed logic structures. Latches are used on critical paths in preference to flip-flops. Although they can be more difficult to design with, latches enable faster circuits by making critical paths less sensitive to the clock skew. Latches allow values to propagate between pipeline stages as they become ready, providing

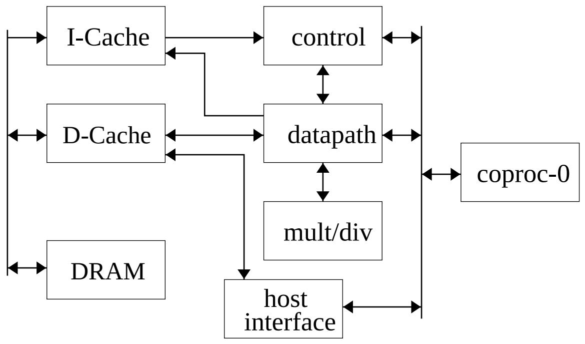


Figure 3-1: Vanilla Pekoe block diagram.

time-borrowing and less stringent timing constraints than a flip-flop based design in which critical paths start and end at flip-flop clock edges.

3.1 Processor

Figure 3-1 shows a block diagram of the Vanilla Pekoe processor. The datapath contains a program counter unit, a 32-word general-purpose register file, and a 32-bit ALU and shifter. The instruction and data caches are each 16 KB and 32-way set-associative with 32 byte lines. They are designed using content addressable memory (CAM) tags and provide single-cycle access times. The multiplier and divider unit operates asynchronously with respect to the datapath pipeline. The system coprocessor provides the system status and control registers and interfaces with both the datapath and control logic. Vanilla Pekoe is designed to connect to a host system and external DRAM; these components are not discussed in detail in this thesis.

3.2 Datapath

The pipeline diagram for Vanilla Pekoe is shown in Figure 3-2. The stages consist of: program-counter generation (P), instruction fetch (F), instruction decode (D), execute (X), memory access (M), and result write-back (W). Each of these is further split into clock high (h) and clock low (l). Dynamic logic components in Figure 3-2 are shown with a shaded bar which indicates when they evaluate.

A conventional five stage pipeline diagram begins with the instruction fetch stage. Vanilla Pekoe has a five stage pipeline starting from instruction fetch, but it is most conveniently described with the program-counter generation as a separate stage. There is no obvious rule governing how the pipeline stages should be divided; for instance, the program-counter logic gets inputs from stages F, D, X, and M.

During stage P, the next sequential program counter (PC) is calculated by incrementing the PC of the previous instruction. The PC-relative branch target address is also calculated using the sign-extended immediate field of the instruction in stage D. Based on the status of the instructions in the pipeline (see Section 3.6), a PC is selected and sent to the instruction

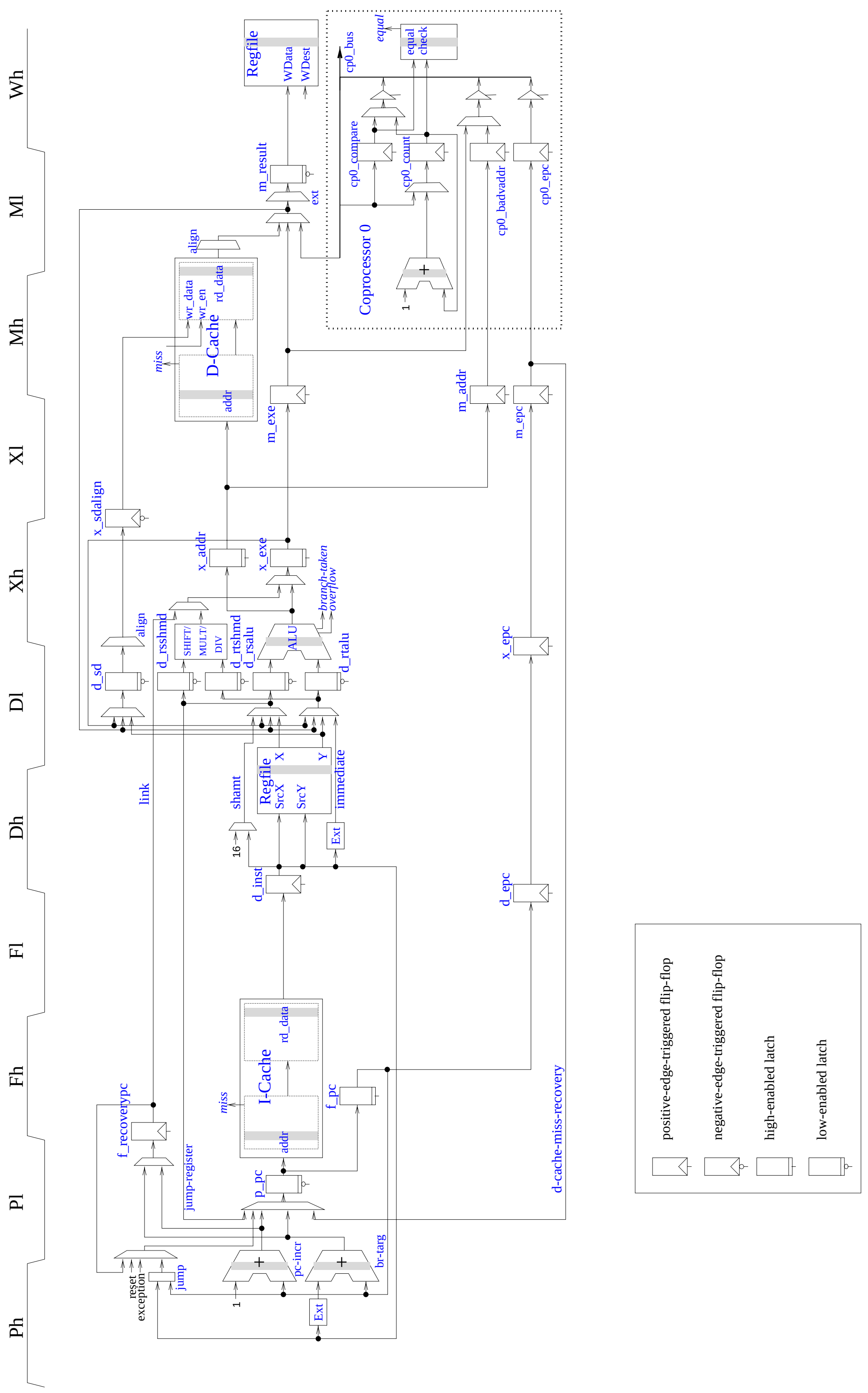


Figure 3-2: Vanilla Pekoe pipeline diagram.

cache (I-Cache) during Pl. If the instruction in stage D is a conditional branch instruction, it is predicted to be either taken or not taken as described in Section 3.6.1; the f_recoverypc register is used to save the PC which is not chosen to recover from a misprediction. This same register is also used to store PC+8 if the instruction in stage D is a jump-and-link instruction.

The F stage is for the instruction cache access. The instruction cache CAM tags evaluate during Fh and determine whether the access is a hit or miss. If it is a hit, the instruction word is read out during Fl and clocked into the instruction flip-flop at the end of the cycle.

The D stage decodes the instruction, generates the bypass network control signals, and detects any pipeline interlocks. During Dh the register file (Regfile) is precharged, and the register values are read during Dl. The bypass muxes select the register values from the appropriate source. The register values are latched, and used to setup the ALU carry-chain by the end of Dl.

The X stage is used to execute arithmetic, logical, shift, and branch check operations, and also to calculate addresses for memory access instructions. The ALU dynamic carry chain and zero detection logic evaluate, producing overflow and branch-taken signals during Xh. The shifter also evaluates its operands, and muxes select between the ALU, shifter, and multiplier/divider values in time to be latched during Xh. This result value is then available to be bypassed to the instruction in stage D. Memory addresses are also computed using the ALU, and sent to the data cache (D-Cache) during Xl.

The M stage executes memory access operations. The data cache CAM tags evaluate during Mh, producing a hit or miss signal. In the case of a hit, data values are read or written during Ml. Loads of bytes and half-words are aligned, and the load data is muxed with the execution data for bypassing to the D stage. Sign-extension for signed loads of bytes and half-words is done after the bypassing mux, and the result value is latched by the end of Ml.

The result value is written back to the register file during Wh, allowing an instruction in stage D to read the value during Dl.

3.3 Cache

Figure 3-3 diagrams the cache layout. The instruction and data caches are each 16 KB and 32-way set-associative. A cache is divided into 16 1 KB banks. The line size is 32 bytes, and each bank contains 32 lines. A bank constitutes one cache set, and the placement of lines in a bank is fully-associative. Tags are maintained in a 32 entry content-addressable memory (CAM). The cache uses a round-robin (FIFO) replacement policy in each bank.

One cache bank is activated during each access. The upper bits of the address are broadcast to the selected bank's CAM tag array during the first phase of a cache access. If the tag is present in the CAM a match signal indicates the location of the line in the data array. During the second phase, the data SRAM is read or written.

The cache uses a write-back policy, meaning that store data is cached and only written to memory when a cache line is evicted. The cache is also write-allocate, meaning that during a store miss the entire line is brought into the cache. A store buffer could be used to hold the data and allow the store instruction to complete, but this optimization is not

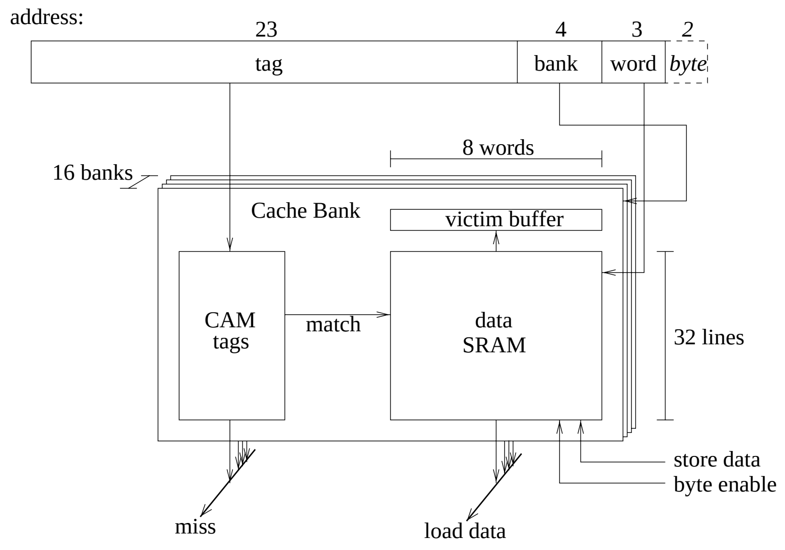


Figure 3-3: Cache diagram.

implemented; the store instruction must stall until the cache miss is resolved. A dirty bit per line is used to indicate whether any words have been written to.

During a cache miss, a victim line is chosen based on the round-robin replacement policy. If the line is dirty it is written into the single line victim buffer so that it can be written back to main memory in the background after the miss has been serviced. A subsequent miss to the same bank will stall until the victim buffer drains. Next, the data is fetched from second level memory which is planned to be DRAM for Vanilla Pekoe. The latency for fetching data from DRAM will depend on the relative clock rate of the processor and on the details of the DRAM operation. In this thesis, the base cache miss penalty is modeled as 17 cycles, and 2 extra cycles are required if the victim line is dirty. Additionally, the instruction and data caches arbitrate for the DRAM, and cache misses must wait if data from a victim buffer is being written back to DRAM.

After a cache miss has resolved, the original memory access must be re-executed. Optimizations such as returning the critical data word as soon as it arrives are possible, but not implemented in this thesis. Additionally, a miss is always serviced to completion once it begins, even if the instruction is killed. Thus, a cache line is allocated even for a mispredicted branch instruction. It might be worthwhile to avoid this overhead, but often the cache line will be used eventually even if not right away.

The pipeline diagram in Figure 3-2 might seem slightly unbalanced: the instruction cache address is available late in Pl and the instruction word is not required until the end of Fl, while the data cache address is available by the beginning of Xl but the load data must be available early in Ml for bypassing. This is a common problem, and it can be resolved by clocking the data cache with a clock slightly in advance of the datapath clock and clocking the instruction cache with a clock slightly delayed from the datapath clock [14].

3.4 System Coprocessor

The system coprocessor (Coprocessor-0) includes the system status and control registers. These are implemented partially in the datapath and partially in the control logic. A bus extends between the datapath and control logic to provide a unified access point for reading and writing the registers during stage M of the pipeline; there is no bus conflict because a single instruction can not both read and write the registers. Coprocessor-0 also includes a count register which is incremented every cycle and a compare register which is compared against the count register to generate a timer interrupt.

3.5 Multiplier and Divider

The multiplier and divider unit contains a single 34-bit adder and uses a radix-2 booth recoding algorithm to complete a 32-bit multiply in 17 cycles and a 32-bit divide in 32 cycles. Signed division with negative operands requires up to 3 extra cycles to conditionally negate the operands and result values. When there is a multiply or divide instruction in stage X of the pipeline, operands are broadcast to the unit through the same latches which are used as inputs to the shifter. Results are read by move-from-high (mfhi) or move-from-low (mflo) instruction during the same pipeline stage.

3.6 Control

This section describes how hazards and exceptional conditions are handled in the processor. These conditions may require the pipeline operation to be interrupted. In particular, pipeline stages may stall or instructions in the pipeline may need to be killed. In order to enable a high-speed pipeline implementation, instructions never stall after they leave D. This is necessary because, for example, stalling an instruction in X would usually require the ALU inputs to be preserved; however, by the time an instruction reaches Xl the ALU is already operating on the instruction in Dl. A killed instruction must not be allowed to change any of the ISA-visible processor state (the register file, memory, and system coprocessor registers). To support this, instructions only update this state during Ml or W, and instructions are never killed after Mh; that is, instructions commit on the negative clock edge between Mh and Ml.

3.6.1 Control Hazards

Control hazards occur when the correct program counter can not be determined due to an unresolved branch instruction. The MIPS instruction set has one branch delay slot which allows the branch determination to be delayed until stage D. However, in Vanilla Pekoe branches are not resolved until the instruction obtains its register operands and uses the ALU in stage X. Therefore, if the program counter unit waits for the branch to resolve before fetching the next instruction, one cycle will be wasted with every branch. To avoid this performance penalty, branch prediction is used to speculatively fetch an instruction.

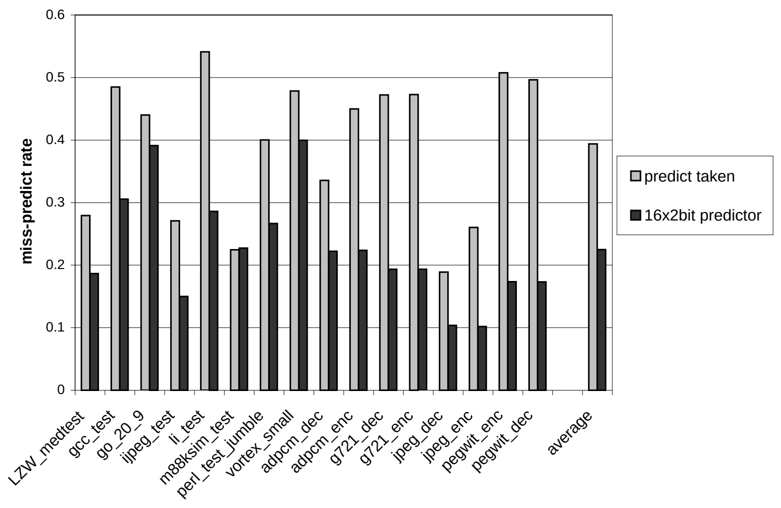


Figure 3-4: Branch mispredict rate.

An extra register in the program counter unit (f_recoverypc) saves the PC which was not chosen. If a branch misprediction is discovered in X, the speculative instruction (in F) is killed and the saved recovery PC is fetched during the next cycle.

The simplest prediction scheme is to always predict that branches are taken; this is the case for around 60% of all branches, as shown in Figure 3-4 (Section 3.7 describes the benchmarks). The Vanilla Pekoe design uses a small 16-entry 2-bit predictor [40] to achieve better accuracy. Figure 3-4 shows that the predictor reduces the average misprediction rate to around 22% . An overhead of allowing branches to be predicted either taken or not taken is that an extra mux is necessary to select the recovery PC. The energy measurements in the next chapter include this mux, but the predictor itself is considered part of the control logic which is not modeled. More analysis is necessary to evaluate the energy-efficiency of this branch prediction.

3.6.2 Data Hazards

Data hazards occur when an instruction requires a register value which is not yet computed. The Vanilla Pekoe design includes value bypassing so that back-to-back dependent operations can use values in the pipeline which have not been written back to the register file. However, when an instruction in D requires the result of a load in X, there is a hazard since the load data will not be available until the next cycle. Vanilla Pekoe detects this hazard, and stalls the instruction in stage D. Sign extension for signed load-byte and load-halfword instructions is performed after the bypass to avoid lengthening this critical path; thus, other instructions which use the result of these load instructions must wait until the value is written back to the register file. Additionally, a bypass from a load to a jump-register instruction is disallowed since it would lengthen the critical path.

An instruction which reads the multiplier/divider result values encounters a hazard if the multiplier/divider unit is in the process of computing a result. The instruction will stall in D until the data is available.

3.6.3 Exceptions and Interrupts

Exceptions occur when an instruction causes an error condition such as an arithmetic overflow or an illegal memory address fault. Various exceptional conditions may be detected in all stages of the pipeline from F through M. However, the exception must not be taken until the instruction commits at the end of stage M; this is because previous instructions must be allowed to commit, and may themselves have exceptions. When an exception is taken at the end of M, all later instructions in the pipeline are killed and the PC is set to the location of the kernel's exception handling routine. Additionally, the Coprocessor-0 registers are updated to indicate the cause of the exception and the PC of the instruction that took the exception. This PC comes from the EPC chain which tracks the PC of instructions as they move down the pipeline.

If the instruction in a branch delay slot takes an exception, the EPC is set to the PC of the branch itself and a status bit informs the kernel that the exception occurred in a branch delay slot. This is so that the program can be resumed by re-executing the branch instruction (all branch instructions in the MIPS ISA are restartable). To handle this, a single-bit flip-flop chain tracks whether the instruction in each stage is in a branch delay slot; if it is the EPC flip-flop in that stage is clock-gated so that it retains the PC of the branch instruction.

Interrupts are caused by events which occur asynchronously to the program execution; for example, a timer expiration or a device signaling that it needs attention. To handle an interrupt, it is associated with an instruction in the M stage and dealt with in exactly the same was as if that instruction had an exception. The current design ensures that a valid restart PC is in the EPC chain by only taking an interrupt when a live instruction is in stage M. It would be more difficult to ensure that the EPC chain always contained a valid restart PC; for example, when a speculative instruction is killed its PC is not a valid restart point. This policy means that an interrupt may need to wait until an outstanding cache miss is resolved. Depending on the system requirements, this may not be acceptable. If necessary, a waiting interrupt could be detected, and a valid PC could be sent down the EPC chain to allow the interrupt to be taken.

3.6.4 Cache Misses

Instruction cache misses are handled by stalling stage F and inserting bubbles into the pipeline until the miss resolves. Data cache misses are more difficult to handle given the decision not to allow instructions to stall after they leave stage D. The solution I adopt is to flush the pipeline during a data cache miss and re-fetch the instruction which caused the miss. Then, it is stalled in D until the miss resolves.

Conveniently, the M stage of the EPC chain contains the valid restart PC for the memory access instruction, so the only hardware overhead for the data cache miss policy is a bus from the M stage EPC flip-flop to the PC mux. Furthermore, this solution handles the tricky situation when a memory instruction in a branch delay slot has a cache miss. The value in the EPC chain is the PC for the branch, and the miss is handled correctly by re-executing the branch instruction.

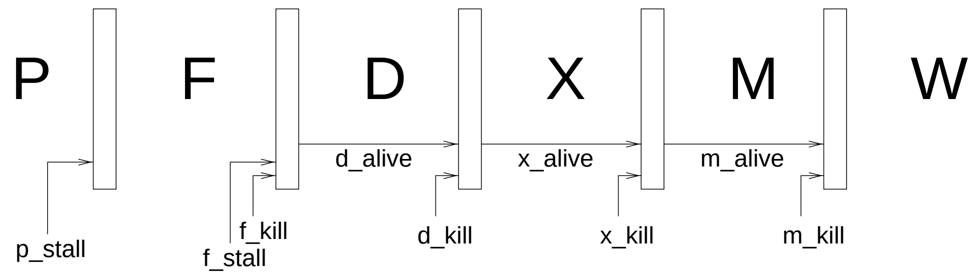


Figure 3-5: Stall, kill, and alive control signals.

3.6.5 Branches

The delay slot instruction following a not taken branch-likely instruction must be annulled, and the second instruction after a branch must be killed if the branch was mispredicted. Additionally, the PC generation for the second instruction after a branch depends on the immediate field of the branch instruction itself. To support these requirements, the Vanilla Pekoe design stalls branch instructions in D if the delay slot instruction has an instruction cache miss. This ensures that the branch remains available for calculating the program counter in stage P. Additionally, it maintains the invariant that the delay slot instruction is in D when the branch resolves in X; this facilitates killing the delay slot or speculative instruction.

3.6.6 Implementing Stalls and Kills

The pipeline control is accomplished using *stall* and *kill* signals for the various stages, as shown in Figure 3-5. A stall signal prevents an instruction from advancing to the next stage, and a kill signal inserts a bubble into the next stage on the subsequent clock cycle. For example, if f_stall and d_kill are asserted, the instruction in D will stall and a bubble will be inserted into X. These signals interact with both the control logic and the datapath.

To support the stall signal, datapath and control registers must be clock-gated to preserve their values. In particular, the program counter must not change when p_stall is asserted, and the instruction register must not clock in new values while f_stall is asserted.

The kill signal is mostly handled in the control logic by maintaining an *alive* bit which indicates the status of instructions. A stage with the alive signal de-asserted indicates a pipeline bubble, and is never allowed to update the permanent state of the processor. If a store instruction is alive during X but must be killed during M (this happens if the store has an address error, or if there is an interrupt), the data cache memory update is aborted by de-asserting the byte enable signals before Ml. The cache circuitry itself prevents a store from updating memory if there is a cache miss.

Pipeline bubbles also must not be allowed to affect the operation of subsequent instructions. For example, result values should never be bypassed from instructions which have been killed; this is handled by setting the register destination specifiers for instructions killed in F or D to zero, but could also be governed by the alive signals. Bypassing from pipeline bubbles is not an issue for instructions killed in X or M because the control maintains an invariant that instructions are only killed in these stages when the entire pipeline is flushed.

Additionally, pipeline bubbles must not be allowed to affect the program counter. If the instruction in D is not alive, it should not be interpreted as a branch instruction; to ensure this, the instruction register is set to a no-operation (nop) value when f_kill is asserted, although the alive signal could be used instead. If a live branch instruction in D is to be killed (d_kill), it is not allowed to affect the program counter generation (this is important when it is in an annulled branch-likely delay slot).

For reference, Figure 3-6 shows behavioral C code which sets the stall and kill signals based on the various hazards and exceptional conditions discussed in this section. The inputs signals are: instruction cache miss (`f_icache_miss`); branch instruction in stage D (`f_branch_delay_slot`), which is not asserted if the instruction in D is itself in a branch delay slot; D stage interlock (`d_interlock`), which includes load-use data hazards, multiplier/divider data hazards, and memory access instructions waiting for an unresolved data cache miss; mispredicted branch in X (`x_br_miss_predict`); not-taken branch-likely in X (`x_br_kill_likely`); exception in M (`m_exception`); and, data cache miss in M (`m_dcache_miss`). Several of these signals may be asserted at the same time; note that more than one of the `if` blocks in Figure 3-6 may be executed. In some cases stall and kill signals will be set by the earlier conditions but reset by later ones; for example, any time there is an exception or data cache miss in M the entire pipeline is flushed regardless of the status of instructions in the other stages.

3.7 Benchmarks

The analyses in this thesis are based on eight SPECint95 [11] and eight MediaBench [31] benchmarks shown in Table 3.1. These applications include a variety of processing requirements and have characteristics which are important for energy-efficient high-performance and embedded microprocessors. Together, the benchmarks constitute almost 13 billion simulated processor cycles.

3.8 Performance

Figure 3-7 shows the simulated CPI rates for various benchmarks on Vanilla Pekoe. Each bar represents the total cycles executed and is broken down into cycles spent executing instructions and cycles wasted for various pipeline bubbles. The bars are scaled so that the number of instructions (including user, kernel, and nop instructions) is 1.0. The pipeline bubbles include cycles spent waiting for an instruction cache miss to resolve (*icache miss*); waiting for a data cache miss to resolve (*dcache miss*); speculative instructions killed due to a branch misprediction (*branch miss-pred*); instructions killed due to an annulled branch-likely delay slot (*branch kill-likely*); stalls due to data hazards (*load-use*); waiting for a multiply or divide operation to complete (*mult/div wait*); and flushing the pipeline during an exception (*exception*).

The benchmarks show a range of performance and are impacted differently by the various stall conditions. On average, the CPI is 1.4 (29% of cycles are stalls) and the stall

```
p_stall = 0;
f_stall = 0;
f_kill  = 0;
d_kill  = 0;
x_kill  = 0;
m_kill  = 0;

if (f_icache_miss) {
  p_stall = 1;
  if (f_branch_delay_slot) {
    f_stall = 1;
    d_kill  = 1;
  }
  else
    f_kill  = 1;
}
if (d_interlock) {
  p_stall = 1;
  f_stall = 1;
  d_kill  = 1;
}
if (x_br_miss_predict) {
  p_stall = 0;
  if (!d_interlock) {
    f_stall = 0;
    f_kill  = 1;
  }
}
if (x_br_kill_likely) {
  d_kill  = 1;
  if (d_interlock) {
    f_stall = 0;
    f_kill  = 1;
  }
}
if (m_exception || m_dcache_miss) {
  p_stall = 0;
  f_stall = 0;
  f_kill  = 1;
  d_kill  = 1;
  x_kill  = 1;
  m_kill  = 1;
}
```

Figure 3-6: Behavioral C code for pipeline stall and kill control.

Benchmark Name	Instruction Count (millions)	Cycle Count (millions)	Description
SPECint95			
LZW_medtest	165.9	313.3	Lempel-Ziv compression (optimized SPECint95 compress)
gcc_test	1580.2	2720.4	GNU C compiler version 2.5.3
go_20_9	408.7	632.8	Artificial intelligence; plays the game of ”Go”
ijpeg_test	585.3	783.3	Graphic compression and decompression
li_test	1227.1	1567.9	A LISP interpreter
m88ksim_test	517.8	598.5	Motorola 88K microprocessor simulator
perl_test_jumble	2630.2	3494.0	A Perl Interpreter
vortex_small	1348.4	1954.2	An object-oriented database program
MediaBench			
adpcm_dec	5.9	6.6	Adaptive differential PCM voice decompression
adpcm_enc	7.0	8.2	Adaptive differential PCM voice compression
g721_dec	290.5	364.5	CCITT G.721 voice compression
g721_enc	298.1	374.9	CCITT G.721 voice decompression
jpeg_dec	5.1	6.4	JPEG image decompression
jpeg_enc	16.3	20.9	JPEG image compression
pegwit_enc	33.9	54.9	public key encryption
pegwit_dec	19.3	30.9	public key decryption
total	9139.7	12931.7	

Table 3.1: Benchmarks and descriptions.

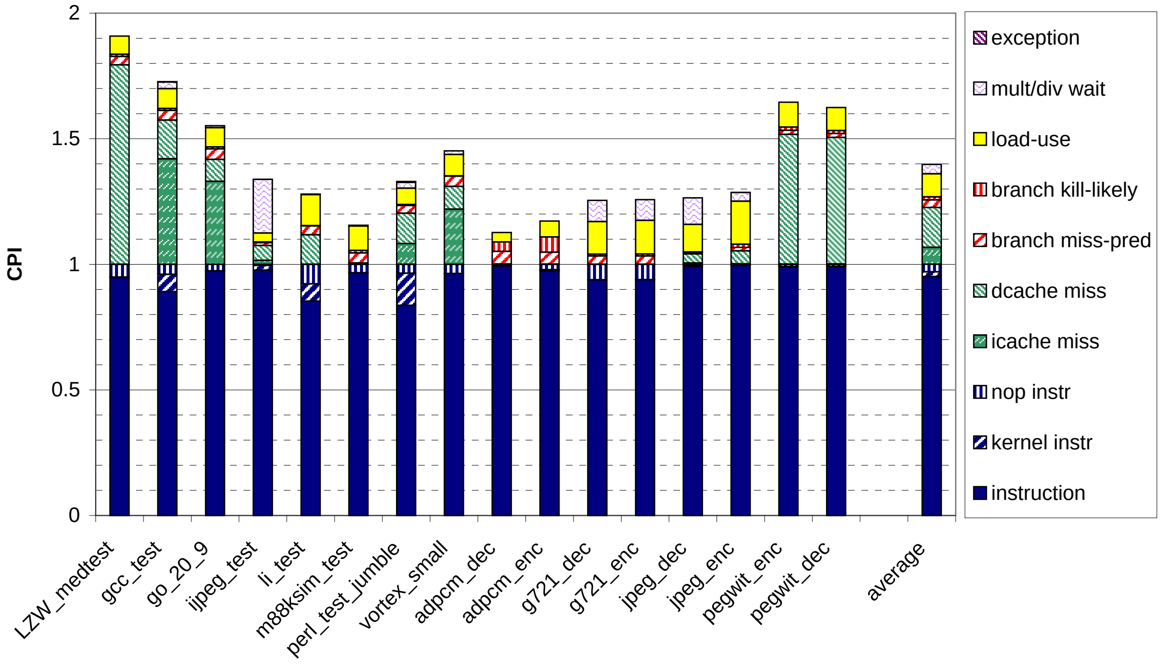


Figure 3-7: Vanilla Pekoe CPI.

cycles are broken down as: 17% instruction cache misses, 40% data cache misses, 8% branch mispredictions, 3% annulled branch-likely delay slots, 23% load-use interlocks, and 9% multiplier/divider wait cycles.

Chapter 4

Energy Optimization

In this chapter I apply various energy optimizations to the baseline microprocessor design. The baseline design uses energy-efficient circuit components such as the flip-flops, caches, and register file. This chapter targets the composition and control of these components.

Figure 4-1 shows the energy consumption of the unoptimized processor. The energy per cycle is shown for each benchmark and broken down into various sub-units; Figure 4-2 diagrams the sub-unit breakdown which is used throughout this chapter and the next. All energy numbers in this thesis are based on a 0.25 μm TSMC CMOS process technology under nominal conditions of Vdd=2.5 V and T=25 ^{o}C; the models are based on the N9BM wafer run. None of the energy numbers in this thesis include the control logic, global clock driver, multiplier/divider unit, or external DRAM.

The breakdown shows that the processor energy consumption is spread across many sub-units in the datapath and caches. Thus, the energy optimizations which I present target different portions of the total processor energy consumption. Evaluating the overall impact of an energy saving technique depends very much on the degree to which the rest of the system has been optimized. Therefore, in presenting each optimization, I limit the energy analysis to the portions of the processor which it affects. A combined breakdown is shown at the end of this chapter, and the next chapter concentrates on a more holistic view of the energy breakdown.

An important property of energy consumption is that it is extremely dependent on the details of a particular circuit implementation. This limits the degree to which results are transferable to other designs, and also makes modeling energy very difficult and error prone. For these reasons, simulated raw energy numbers tend to be uninteresting and lack credibility; nevertheless, many energy optimization studies only report raw energy totals. For each optimization technique in this chapter, I present architectural properties such as transition counts in addition to the simulated energy measurements. These show how the savings are achieved and they are mostly independent of a particular implementation. Additionally, I break the energy numbers into relevant sub-units to show where the reductions are obtained.

In order to have shorter simulation times and less data to manage, the results in the remainder of this chapter are based on the harmonic mean for a subset of six benchmarks: LZW_test, go_20_9, m88ksim_test, adpcm_dec, jpeg_enc, and pegwit_enc. I chose these based on their wide variety of performance characteristics demonstrated in Figure 3-7.

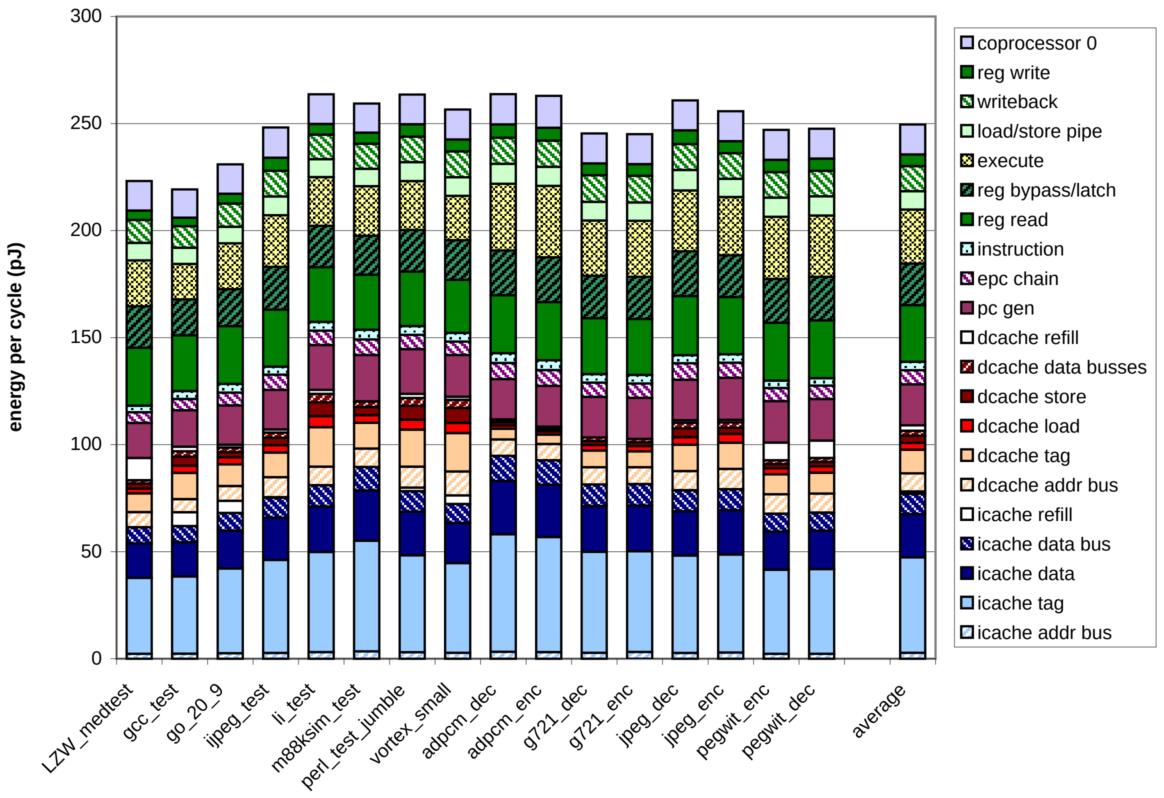


Figure 4-1: Unoptimized energy per cycle.

These benchmark runs constitute 1.6 billion simulated processor cycles, and their mean CPI is 1.44.

4.1 General Clock Gating

Clock gating is an important technique in minimizing the energy consumption of a processor [43]. When clock gating is applied, the clock input to flip-flops, latches, or dynamic logic components is prevented from toggling. This saves energy by reducing the data transitions in down-stream functional units, and also by reducing the clock activity in the gated units.

The application of clock gating is very dependent upon the datapath clocking strategy. The clocking method used in Vanilla Pekoe is shown in Figure 4-3. A global clock driver distributes the clock signal to local drivers which buffer the signal and drive multi-bit (e.g. 32-bit) flip-flops, latches, and dynamic logic components. Clock gating is achieved by incorporating an enable signal in the local clock drivers; if the enable signal is not asserted, the clock output of the drivers will not toggle. The overhead for clock gating is minimal since it is amortized over an entire multi-bit component. Additionally, in designs that distribute the clock signal in a tree structure, gating can be applied further up the clock tree so that a single signal can gate an entire region of the chip.

In order for the clock gating circuitry to operate correctly, the enable signal must be-

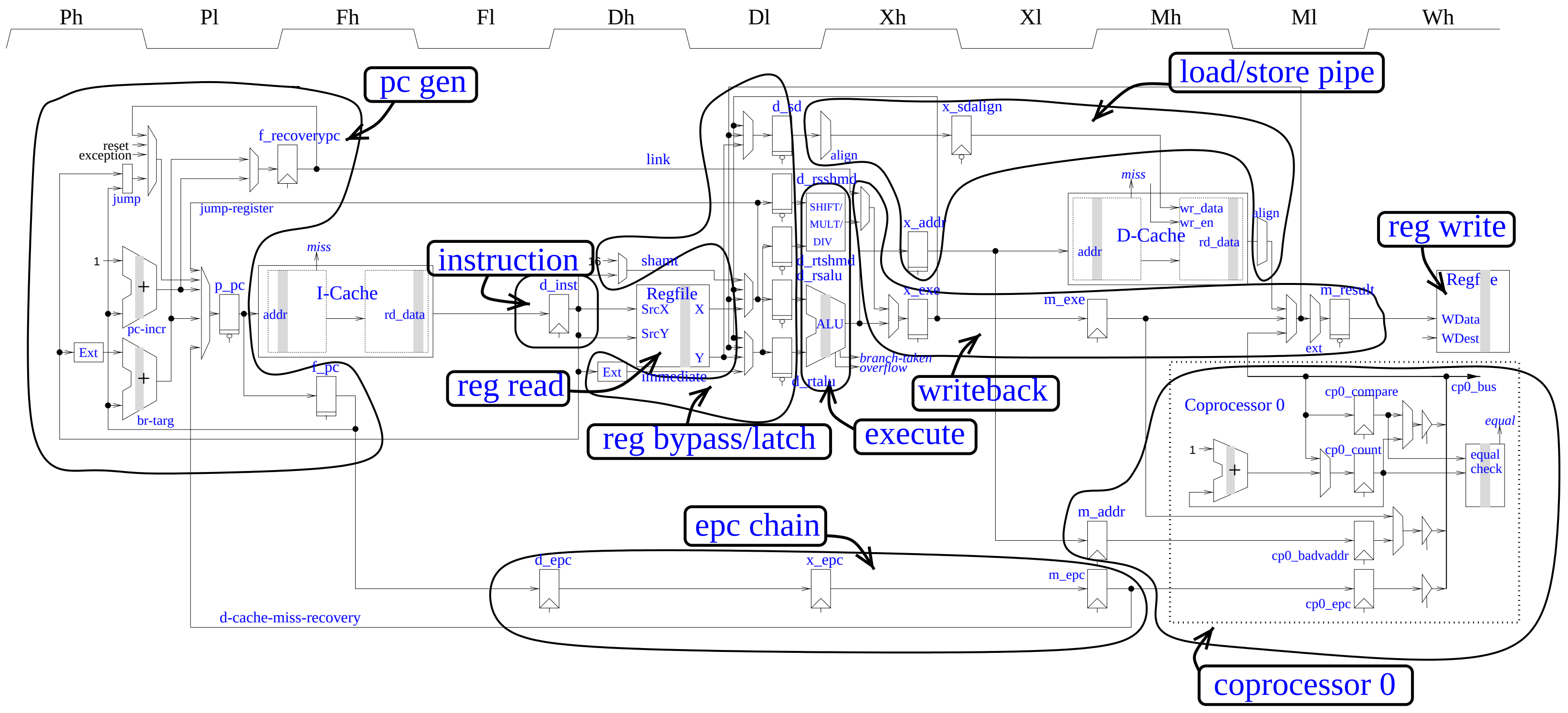


Figure 4-2: Datapath subunits diagram. This is based on the pipeline diagram in Figure 3-2.

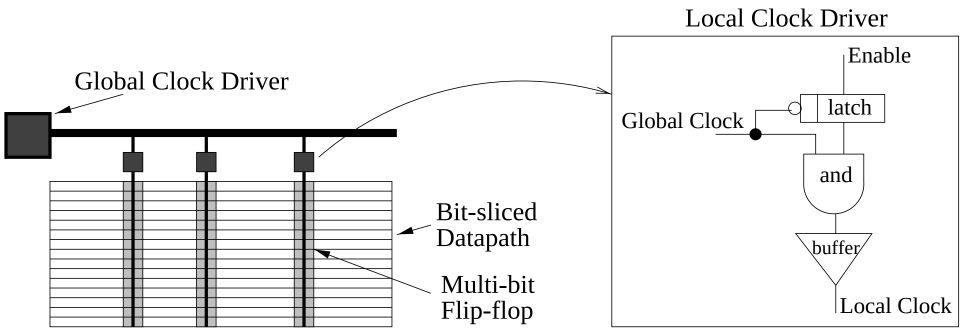


Figure 4-3: Datapath clocking strategy.

come valid a certain amount of time before the active clock edge (e.g., the positive edge for a positive-edge-triggered flip-flop or a high-enabled latch). This will prevent flip-flops from clocking in new values and latches from becoming transparent. For dynamic logic components, clock gating is used to keep components in the precharge mode.

In this section, I consider three levels of clock gating:

- *Minimal clock gating* is required for correct operation. For example, during pipeline stalls, the input register values for the stalled stage must be preserved. Also, the exception program counter chain uses clock gating to ensure that the correct PC is available if the instruction in a branch delay slot takes an exception. Additionally, the Coprocessor-0 exception registers are only activated during an exception.

- *Basic clock gating* is performed based on the operand types of instructions. For example, the latches controlling the shifter inputs (d_rsshmd and d_rtshmd) need only be activated for shift instructions. Another example of clock gating based on operand type is that the dynamic branch target adder must only evaluate when there is a branch instruction. The ability to gate components in the decode stage may be timing critical depending on the latency of decoding the instruction word and calculating the enable signals; however, there should be no timing constraints for applying clock gating to components in subsequent stages.

- *Liveness clock gating* is performed based on the liveness of instructions in the pipeline. When there are pipeline interlocks or cache misses, bubbles are inserted into the pipeline providing an opportunity for clock gating. Since instructions stall for interlocks and data cache misses in stage D of the pipeline, it is important to disable the latches in this stage for stalled instructions. For example, consider an instruction stalled waiting for a data cache miss to resolve; the memory instruction computes its address using the ALU, so if clock gating is not applied the latch inputs to the ALU and the ALU itself will be clocked every cycle. Clock gating can also be applied to the instruction flip-flop during an instruction cache miss. The current design clocks a NOP value into this flip-flop when the instruction in stage F is to be killed; however, when this happens again on the next cycle, the clock can be gated. A similar technique could be applied to the control logic flip-flops although they are not modeled in this thesis.

	flip-flops and latches clocked (707 total)	input data transitions	output data transitions
minimal	559.8	159.8	88.3
basic	304.6	141.7	46.9
liveness	271.4	138.0	45.1

Table 4.1: Total flip-flop and latch clock and data activity per cycle with various levels of clock gating. Note that input transitions are counted more than once if they connect to more than one flip-flop or latch.

Figure 4-4 shows the clock and input data activity for the flip-flops and latches in the datapath as the various levels of clock gating are applied successively. Table 4.1 summarizes the total clock and input data activity for all 707 individual flip-flops and latches in the datapath. Most of the clock gating activity reduction is achieved by the basic clock gating, with the liveness gating providing some additional benefit. Considering one example, basic clock gating reduces the clock activity for the shifter and multiplier/divider input latches (d_rsshmd and d_rtshmd) from 2.0 to 0.4; liveness clock gating further reduces this activity to 0.23. The activity reduction due to basic clock gating will vary depending on the instruction mix, while liveness clock gating will be affected by the overall CPI. Together, these reduce the number of flip-flops and latches clocked per cycle by over a factor of two. After these optimizations, less than 9 32-bit flip-flops and latches are clocked per cycle on average, and only around 45 flip-flop and latch output bits toggle per cycle. The optimizations presented in subsequent sections will further reduce the clock activity and exploit the low data activity to reduce energy.

The activities in Figure 4-4 reveal some interesting behavior that may not be intuitive. For example, one might think that the ALU should be active almost every cycle since it is used for all arithmetic, logical, conditional branch, and memory access instructions. However, its actual activity is less than 60% since there are many pipeline stalls and the shifter is isolated with separate latches. This demonstrates a benefit of duplicating the RS and RT latch inputs for the shifter and ALU; clock gating prevents the ALU inputs from toggling during shift instructions. Another interesting observation is that the X and M stage latch and flip-flop on the write-back path are active for only 43% of all cycles. The reason for this relatively low activity is that conditional branch and memory access instructions do not need to clock values down this path.

It can be somewhat elusive to determine all the possible opportunities for clock gating, but in many cases the gating signals are already available in the control logic. For example, the write-back path can be gated with the existing bit in the control logic which indicates whether or not an instruction writes the register file. However, things are not quite this straightforward since instructions which write to the Coprocessor-0 registers also use the write-back path and load instructions write the register file, but do not need to clock the ALU output through the X and M stage write-back path. That said, a preliminary datapath design took the bad-virtual-address input from the write-back path so that memory addresses had to be clocked down this path, at least until the address check was performed during phase Xl. Taking the bad-virtual-address input from the an extra register on the

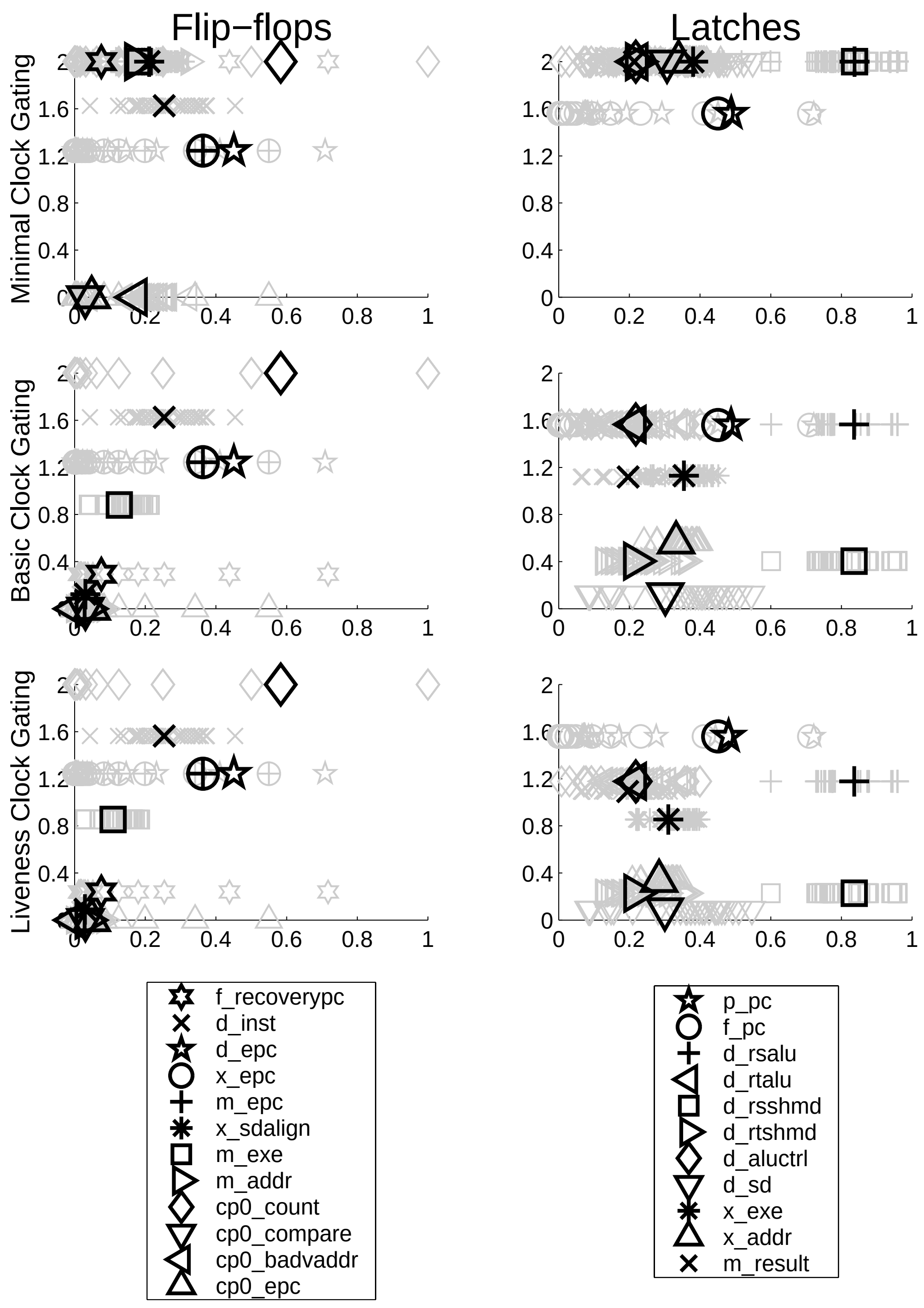


Figure 4-4: Clock and data activities for flip-flops and latches with various levels of clock gating. The clock activity is shown on the y-axis and input data activity on the x-axis. For each, the activity is the number of transitions per cycle; the maximum clock activity is 2.0 because the clock can make up to two transitions per cycle (positive and negative edge). The gray markers represent individual bits, while the black markers are averages for each multi-bit flip-flop or latch. Flip-flops are shown in the left column, and latches in the right. The top row shows the activity under minimal clock gating, the middle shows basic clock gating based on operand type, and the bottom shows the activity when clock gating is additionally based on the liveness of instructions in the pipeline.

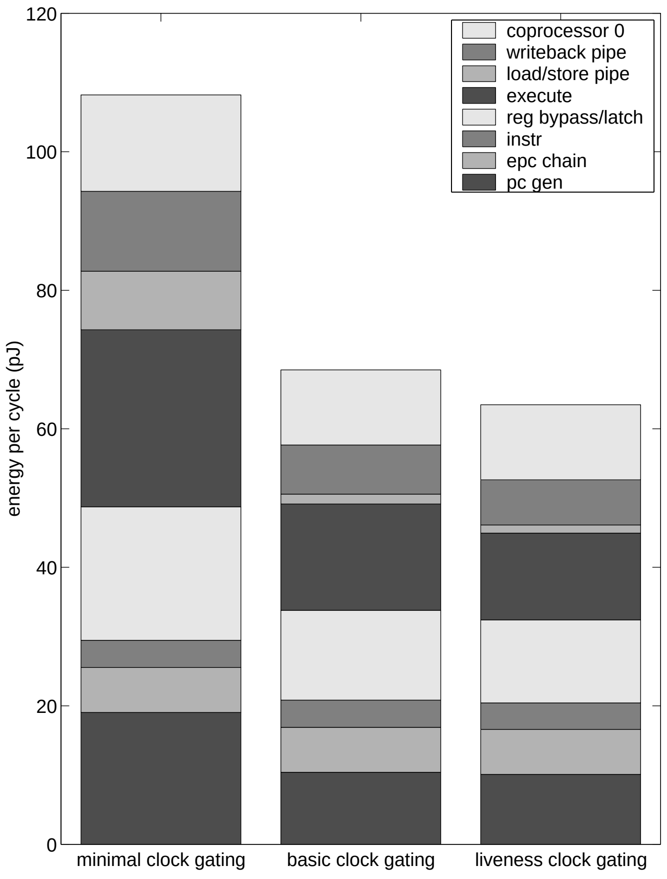


Figure 4-5: Clock gating energy savings. Each bar depicts the energy for the entire datapath excluding the register file.

address path (m_addr) eliminated this problem and also lowered the input data activity for the bad-virtual-address flip-flop.

Figure 4-5 shows the energy savings achieved in the datapath (minus register file) using clock gating for the flip-flops, latches, and dynamic adders. Significant energy savings are obtained throughout the datapath due to the decreased clock and data activities. Figure 4-6 shows the clock and data energy reduction for the flip-flops, latches, and dynamic branch target adder. It is interesting that more than half of the energy savings from clock gating come from the reduced clock activity itself rather than the decreased data activity achieved by preventing values from propagating down the pipeline. This is due to the predominantly low data transition activity in the pipeline. Out of the total 45 pJ per cycle energy reduction, around 20 pJ is saved by not clocking flip-flops and latches, and around 5 pJ is saved by not clocking dynamic adders.

When clock gating flip-flops with a master-slave latch structure, normally the master latch will be gated open and the slave latch will be gated closed (see Figure 4-19(c) for a diagram); this is what happens when the clock input to a positive-edge-triggered flip-flop remains low. The disadvantage of this situation is that any transitions on the flip-flop's input signals propagate through the master latch and consume energy. However, if the enable signal is available early enough, both latches can be gated closed. Essentially this

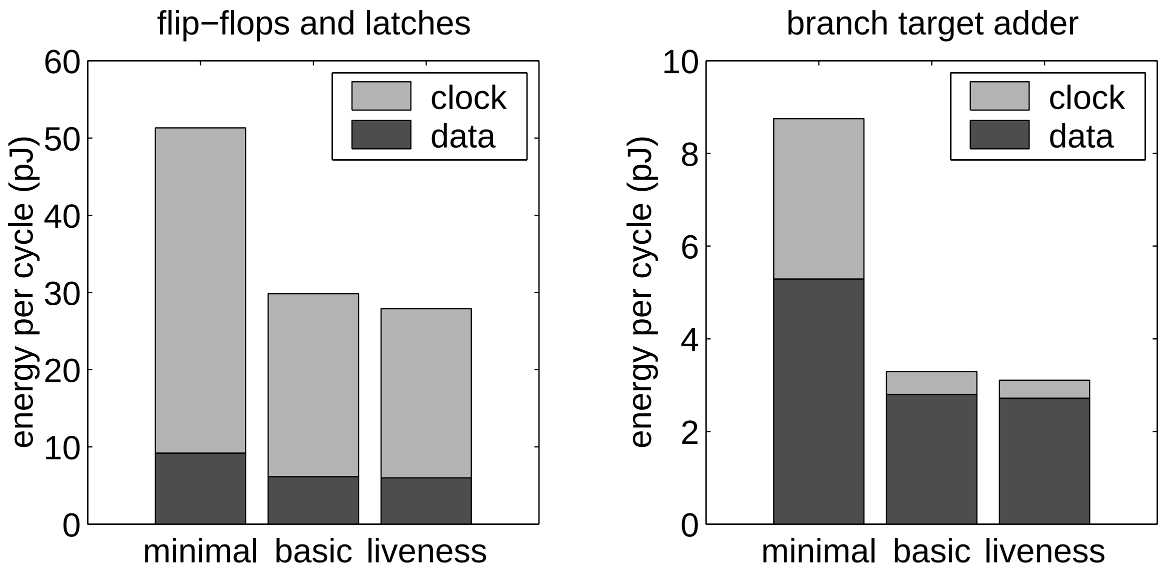
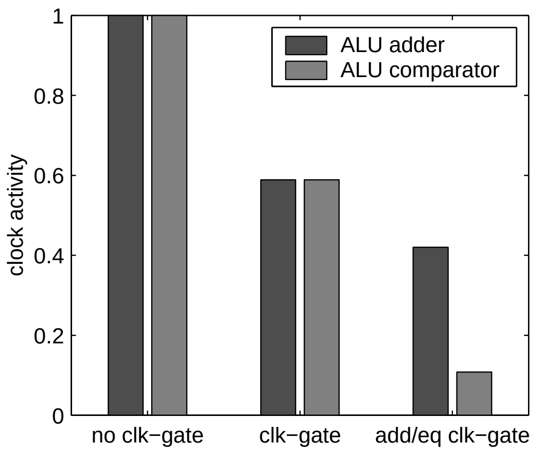


Figure 4-6: Clock versus data energy with clock gating in flip-flops, latches, and branch target adder.

is done by separating the flip-flop into two latches and clock gating both of them. For the flip-flops and latches used in this thesis, this technique reduces the input transition energy by a factor of 6 while the clock is gated. Thus, this optimization can be important for flip-flops which are usually gated but have high input transition frequencies. For example, the first conception of the Vanilla Pekoe datapath used a flip-flop followed by a latch for the store-data path. In the design shown in Figure 3-2, I use a latch followed by a flip-flop to minimize the input data energy; these energy savings are included in the baseline design.

4.2 ALU

The clock gating presented in Section 4.1 treats the ALU as a unified component that can be either enabled or gated. However, the ALU contains a dynamic adder and a dynamic equality comparator which can be gated separately based on the instruction type. Figure 4-7 shows the activity and energy for the adder clock and the equality comparison circuitry when they are ungated, clock gated together, or clock gated separately. Most instructions use only the adder or the comparator but not both, and the individual clock gating reduces the adder activity by 29% and the comparator activity by 82%. When active, the comparator precharges a heavily loaded dynamic node and discharges it whenever the inputs are not equal. This consumes a significant amount of energy which is eliminated by the clock gating. For the adder, clock gating saves energy by reducing the clock energy itself and preventing the output nodes from toggling. Overall, the individual clock gating reduces the ALU energy by 20% compared to the unified clock gating.



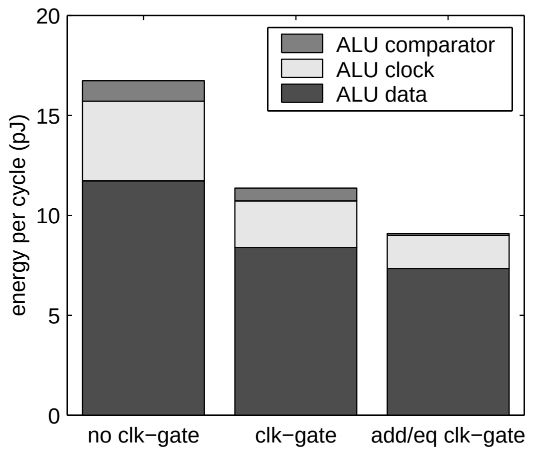


Figure 4-7: ALU adder and equality comparator clock gating.

4.3 Register File

The register file and bypass network consume around 40% of the unoptimized datapath energy. In this section I analyze several techniques for reducing register file reads and switching activity in the bypass network. Most of these techniques were previously presented in [44], but the analysis in this section is based on my implementation of the optimizations in the SyCHOSys processor simulation.

In the base case, every instruction reads two operands from the register file; however, many of these values are never used. Precise read control (PRC) eliminates reads based on instruction type. For example, instructions which operate on immediate values do not use RT, and shift instructions with immediate shift amounts do not use RS. Figure 4-8 shows that around 30% of register file reads are eliminated using PRC.

Another opportunity for eliminating register file reads is when operands are supplied by the bypass network. Bypass skip (BS) uses the bypass mux control signals to determine when register values will be bypassed from instructions in the pipeline. In this case these reads can be eliminated, and Figure 4-8 shows that almost 60% of register file reads are eliminated when BS is combined with PRC.

Register file reads can also be eliminated using liveness gating (LVG), as discussed in section 4.1. In the base case, instructions stalled in the decode stage (waiting for a data cache miss, for example) read from the register file every cycle; however, the reads can be avoided as long as the instruction is stalled. Figure 4-8 shows that around 70% of register file reads are eliminated when LVG is used along with BS and PRC. Figure 4-9 shows that this reduction varies from about 60% to 75% across the benchmarks.

The base register file design precharges the data bitlines high during the first clock phase. During the second phase, the rx read-port bitlines evaluate to the value of the RS register and the ry read-port bitlines evaluate to the inverted value of the RT register (and are subsequently inverted to the positive value by static inverters). Since a high percentage of the bits read out of the register file are zeros, most of the time the bits in the rx read-port precharge high and evaluate low. Figure 4-10 shows the transition frequencies per read for each bit of the rx and ry register file outputs. As shown by the figure, with the standard

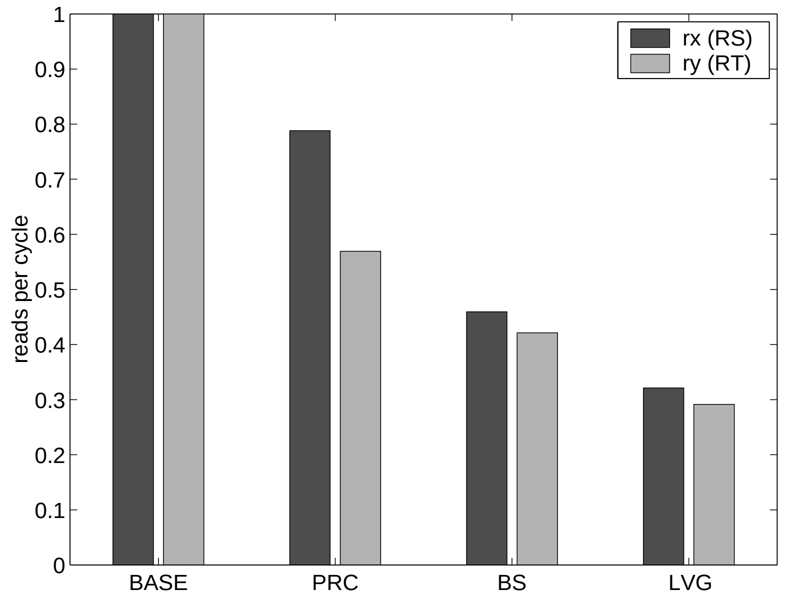


Figure 4-8: Register file read reduction successively applying precise read control, bypass skip, and liveness gating.

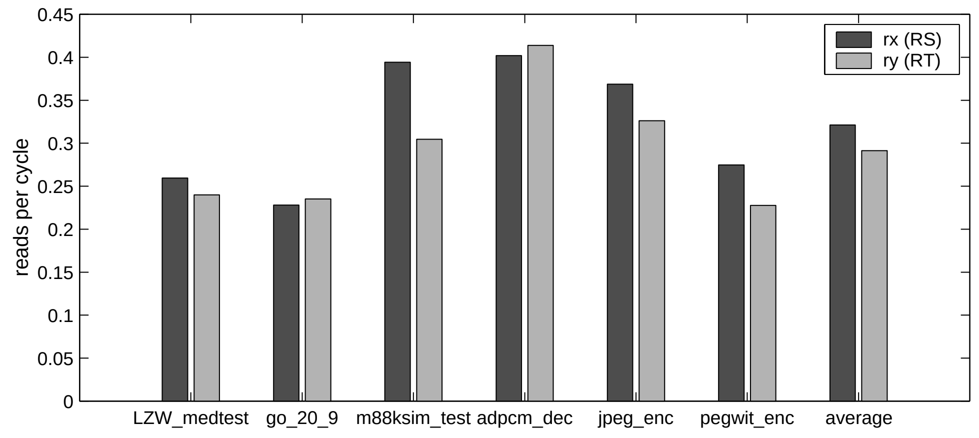


Figure 4-9: Register file read reduction for various benchmarks.

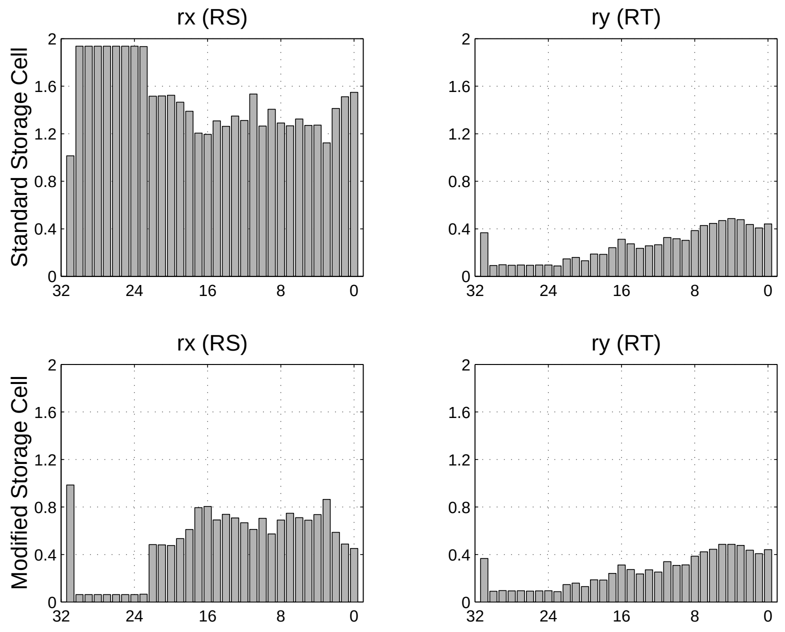


Figure 4-10: Register file bit transition activity reduction using a modified storage cell. To isolate the effect of the read polarity, the frequencies have been scaled by the actual number of register reads. The upper plots show bit activity per read for the rx and ry read ports using a standard storage cell, and the lower plots show the activities using a modified storage cell.

storage cell many of the rx read-port bits transition almost twice per read.

A modified storage cell (MSC) is described in [44]. With minimal overhead, this design reads inverted data for rx in addition to ry, so that none of the register file bitlines discharge when zero bits are read from the register file. The lower half of Figure 4-10 shows the dramatic reduction in bit activity for the rx read-port; its activity becomes similar to ry. This leads to energy savings in the register file as well as the muxes, latches, and busses that it connects to.

The combined effect of these techniques on the energy consumption of the register file and bypass network is shown in Figure 4-11. In this analysis, I assume that clock gating has already been applied to the bypass latches. The combined techniques reduce the register file read energy consumption by 83%, and the combined energy by 68%.

4.4 Program Counter

Each cycle the next program counter (PC) is computed, and previous program counters are saved in the exception program counter (EPC) chain; this activity accounts for around 20% of the unoptimized datapath energy consumption. An opportunity arises for eliminating some of this energy consumption since the data activity of the program counter decreases exponentially from the low-order to high-order bits. To take advantage of this unbalance, the program counter components can be segmented into lower and upper sections; most of the time the upper sections need not be clocked. This technique targets the clock energy

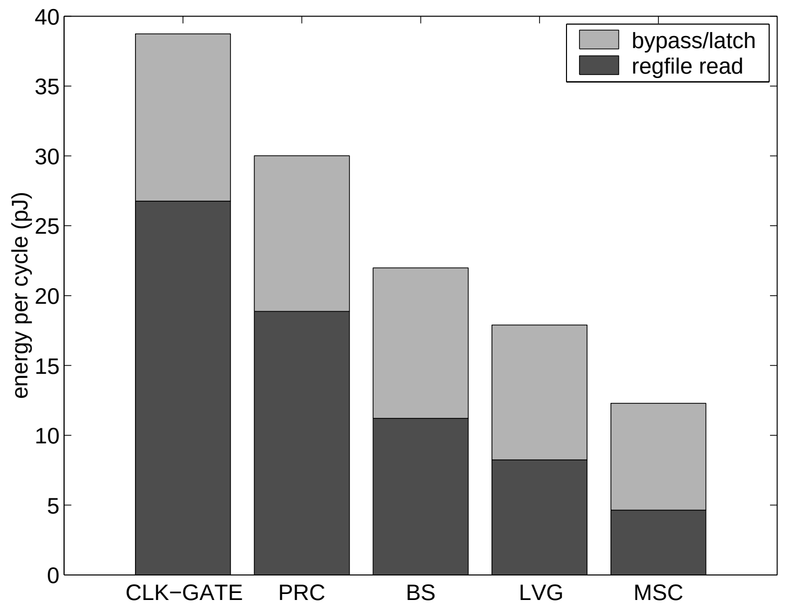


Figure 4-11: Register file and bypass network energy reduction with successive optimizations.

which is dominant due to the low data activity.

I first segment the dynamic PC incrementer (refer to Figure 3-2) into upper and lower sections. The lower section uses static logic and increments the lower PC bits during Ph. If there is a carry-out, the dynamic incrementer for the upper PC bits is activated during Pl to compute the new upper PC. This simple scheme adds very little complexity since the carry signal is already present in the logic. The timing overhead is expected to be minimal since the static lower incrementer operates in parallel with the precharge phase of the dynamic upper incrementer, and the length of the dynamic carry chain is reduced; furthermore, a small increase in latency is tolerable since the incrementer has available timing slack.

If the instruction flow is sequential (not a branch or jump) and there was no carry out from the lower PC incrementer, the upper sections of the PC latches will not change and can be gated. For simplicity, the upper part is clocked during every branch or jump, even though its value may not change for very small branch offsets. This clock gating is based on the existing PC mux select signals (to determine sequential flow) and incrementer carry bit, so it adds minimal overhead.

The upper and lower sections of the three 32-bit registers in the EPC chain are clocked for every branch or jump instruction (and never for the instruction in the branch delay slot) as it goes down the pipeline. For sequential instructions between branches, the upper segments need only be clocked if the upper portion of the PC changes, as determined by the clock enable signal for the PC latch. One minor complexity is that there may be stalls such as an instruction cache miss after the upper PC changes but before the instruction advances down the pipeline. To handle this case, I use a storage element which is set whenever the upper PC changes, but not reset as long as the instruction is stalled in stage F. As the instruction goes down the pipeline, a chain of single-bit registers tracks whether or not the upper PC has changed, and these are used to enable the clocking of the upper EPC registers.

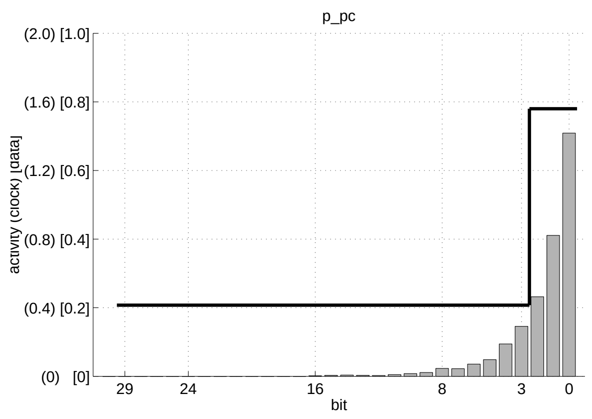


Figure 4-12: Segmented PC latch clock and data activity. The solid line shows the clock activity for each bit in the latch; 2.0 is the maximum clock activity. The bars show the data activity for each output bit of the latch; these are on a maximum scale of 1.0 transitions per cycle so that the data activity matches the clock activity even though it makes at most half as many transitions.

I implemented the PC segmentation in the Vanilla Pekoe datapath with the components split between the lower 3 bits and the upper 27 bits (including the incrementer, PC latches, and EPC chain). The reason for this split width will be made apparent in Section 4.5.

Figure 4-12 shows the clock and data activity for each bit in the segmented PC latch. The lower 3 bits are clocked almost 80% of the time, while the clock activity for the upper 27 bits is around 20%. This figure demonstrates a tradeoff in segmenting the latch between how often the upper bits must be clocked and how many bits are clocked almost every cycle in the lower bits.

For the upper part of the PC incrementer, the clock activity is reduced from 100% to around 10%. One might think that the maximum reduction should be to 1/8th the original value (this is the case for purely sequential operation); however, when there are program loops that operate entirely within 3 bits of address space (8 words), the lower PC adder never has a carry-out and the upper part of the PC incrementer never needs to be clocked. Additionally, the PC incrementer does not need to be clocked when the PC comes from the branch mispredict recovery register, from the EPC chain (during a data cache miss), or from a jump instruction.

Figure 4-13 shows the energy breakdown for the relevant program counter components before and after segmenting the low order bits. This technique lowers the clocking energy for the high order bits but does not affect the data transition energy. The clocking energy for the upper flip-flops and latches is lowered by almost four times corresponding to the decrease in activity shown in Figure 4-12. The clocking energy for the PC incrementer is reduced by around ten times according to its reduction in clock activity. After these optimizations, the upper 27 bits of the program counter components use less energy than the lower 3 bits; therefore the chosen segmentation size is reasonable. The energy breakdown also includes overhead for clock gating. In the base case, the clock gating overhead consists of 3 single-bit flip-flops to hold the enable signals for the PC latches and EPC chain; with the segmentation, this overhead increases to 8 single-bit flip-flops (3 additional flip-flops

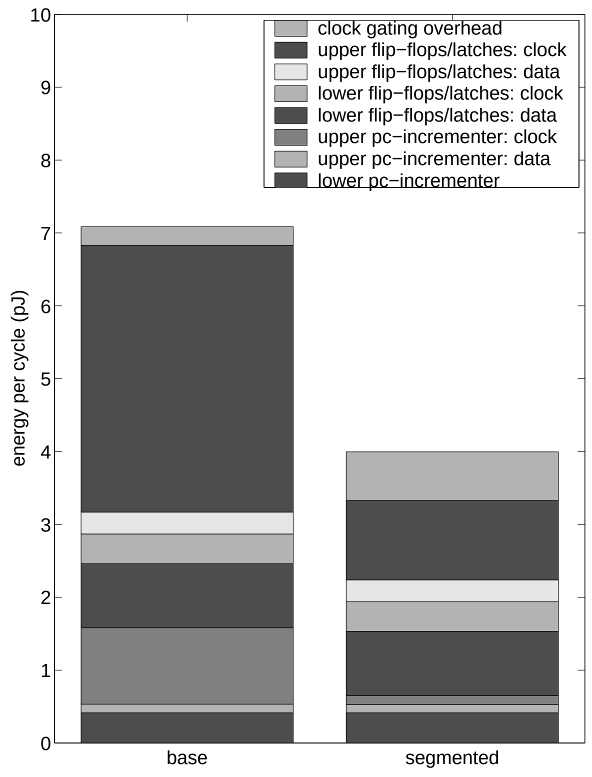


Figure 4-13: Energy for PC components with segmentation. Each bar includes the energy for the incrementer, PC latches, and EPC chain.

for the upper EPC gating, and 2 for keeping track of whether the upper PC changed during an instruction cache miss).

It should be noted that there are many circuit and architectural techniques that can be used to lower the program counter energy. [9] describes a mechanism which segments the PC into four 8-bit blocks and performs the increment in a byte-serial manner, stalling if there is a carry-out. The minimum clocking activity for that scheme is 8 bits per cycle, about the same as the average activity of 7.8 achieved using the scheme described in this section (80% times 3 bits plus 20% times 27 bits). The mechanism described here is probably simpler though since it does not require any extra stall cycles, however it does require the use of a dynamic adder to meet delay constraints.

4.5 Cache

Caches often consume a majority of the total chip energy, and thus are important targets for energy optimization. However, even in the base case, the cache design has already been highly optimized for low energy usage [49]. It is split into 16 banks, and during each access only one bank is activated; without this optimization the energy consumption would increase by nearly a factor of 16. Additionally, the cache design uses segmented word lines and low-swing bit-lines to lower the access energy. The energy consumption of the cache is also very sensitive to the bank size. Reducing the cache bank size by a factor of two cuts the

	total (pJ)	address bus (average)	tag search	data access	data bus
I-Cache load	98	3 (3%)	57 (58%)	26 (27%)	12 (12%)
D-Cache load	106	11 (10%)	57 (54%)	26 (25%)	12 (11%)
D-Cache store	133	11 (8%)	57 (43%)	53 (40%)	12 (9%)

Table 4.2: Cache access energy breakdown.

access energy in half although there is some fixed and additional energy overhead and also area overhead. Initially, the Vanilla Pekoe cache was to use 2 KB banks, but reducing the bank size to 1 KB was found to lower the energy considerably. Set-associative caches usually have significantly higher hit rates than direct-mapped caches, and are therefore more energy-efficient overall despite their overhead. For caches with a set-associativity greater than two, highly associative content addressable memory (CAM) based designs are more energy-efficient than designs which store the tags in SRAM [2, 49]; the modeled cache is a CAM based design in which each bank is one 32-way fully-associative set. The cache configuration used in this thesis is very similar to that in the energy-efficient StrongARM processor [14] which also uses 16 KB 32-way set-associative instruction and data caches split into 16 banks, and its successor XScale [15] which uses 32 KB 32-way set-associative instruction and data caches. Additionally, [2] uses a 16 KB 32-way set-associative unified cache split into 16 banks, which was found to be the most energy-efficient configuration. The energy breakdown for the modeled cache is shown in Figure 4.2.

In the CAM based cache design which was modeled, the tag energy is far more than that of reading out the data word as shown in table 4.2. However, if an access is to the same cache line as the previous access the tag check can be eliminated. This is done in the cache circuitry by saving the match line output of the CAM tag array (which has a small energy overhead that is not modeled here) and then accessing the cache line data again on the next cycle. [2] describes an implementation of this technique for a unified instruction and data cache; in order to determine if an access is to the same line as the previous access, the new tag is compared with the previously saved tag to avoid activating the entire tag array. However, this comparison adds energy overhead and adversely affects the latency of the cache access.

For an instruction cache, accesses are sequential by default and in most cases it is very simple to determine when an instruction fetch is from the previously accessed cache line [35, 38]. In fact, the segmented program counter implementation described in Section 4.4 satisfies this requirement perfectly. Since I split the program counter between the low order 3 bits and the high order 27 bits, whenever the upper segment is not clocked the access is guaranteed to be to the same 8 word cache line. Thus, tag checks for all intra-line-sequential (ILS) accesses can be eliminated. This reduces the CAM tag check rate from about 78% to 24% (a 70% reduction). Further optimizations are possible, but only with significantly greater complexity; for example, way-memoization is a technique which can eliminate tag checks for inter-line-sequential and non-sequential accesses [32].

Figure 4-14 demonstrates the energy impact of the ILS optimization. For reference, the energy for doing a tag search every cycle is shown, although the base-line design only

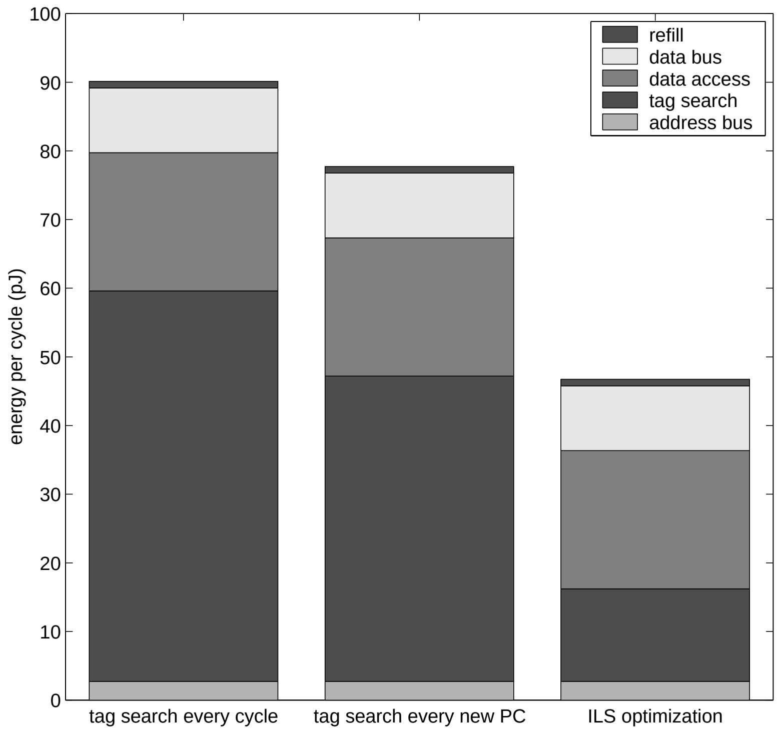


Figure 4-14: Instruction cache energy consumption while eliminating intra-line-sequential tag searches. The base-line design does a tag search for every new PC; the energy for searching tags every cycle is shown for reference.

performs a tag search for every new PC (and not during instruction cache misses). The tag energy dominates the access energy before the ILS optimization, but afterwards it is less than 30% of the total instruction cache energy. Overall the instruction cache energy savings are about 40% compared to the base-line design (and 50% compared to a design which does a tag search every cycle).

Cache tag searches also dominate the data cache energy consumption, and potentially these can be eliminated if they are to the last line accessed. Figure 4-15 shows statistics for the percentage of data cache accesses which are to the last-line accessed (36% on average) as well as the percentage which are to the last-line accessed for a given bank (78% on average). The data shows that a significant portion of the tag checks could be eliminated using a scheme such as the one presented in [2], or ideally a compiler could identify many of these redundant tag searches [47]. Potentially, the address calculation (including register file reads, addition in the ALU, and pipeline flip-flops) could even be eliminated just as the upper portion of the program counter generation was eliminated in Section 4.4. Another data cache energy optimization is to use compression to take advantage of the prevalence of zero bits in data cache accesses [46]. Energy savings for these optimizations are not included in this thesis.

For completeness, Figure 4-16 shows the percentage of instruction fetches which are to the last-line accessed as well as the percentage which are to the last-line accessed for a given bank.

Switching activity statistics can help determine whether busses should use static or dy-

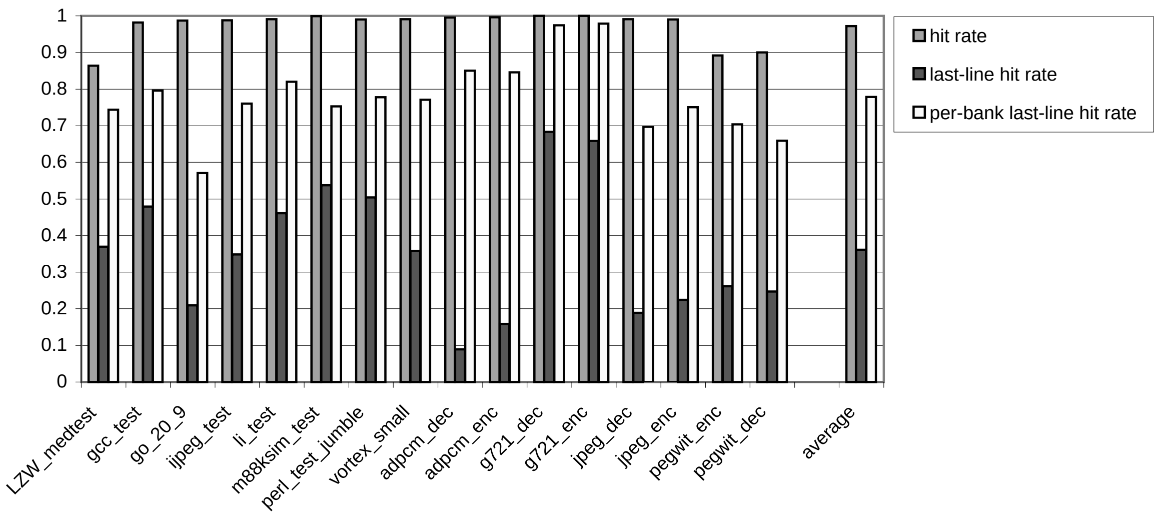


Figure 4-15: Data cache hit rates including last-line and per-bank last-line hit rates.

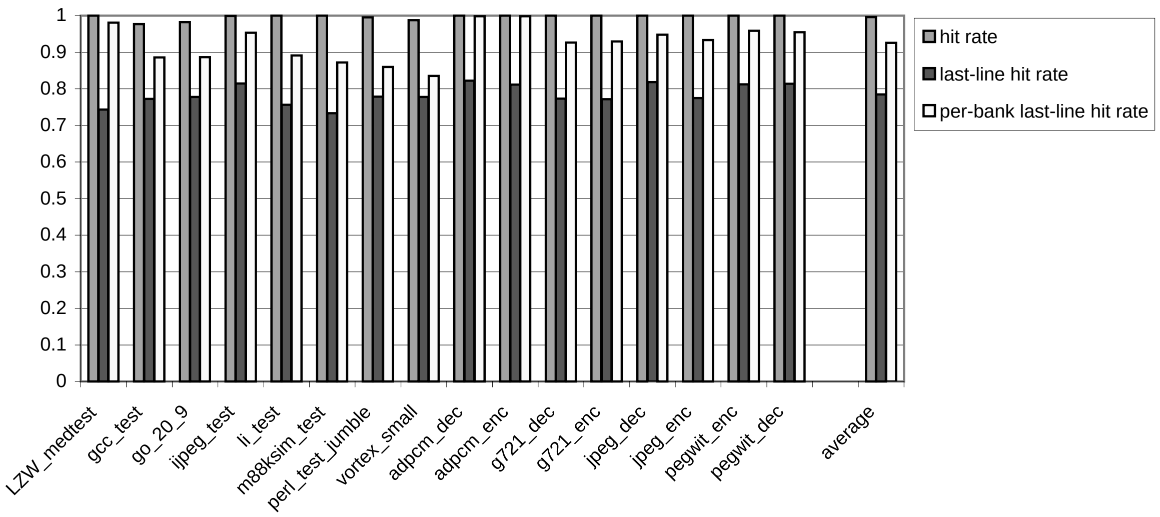


Figure 4-16: Instruction cache hit rates including last-line and per-bank last-line hit rates.

	bits	average bit transitions	dynamic bus energy (pJ)	static bus energy (pJ)
I-Cache address	30	2.26	11.4	3.52
I-Cache data	32	9.97	12.2	15.58
D-Cache address	30	6.42	11.4	10.03
D-Cache load data	32	9.11	12.2	14.23
D-Cache store data	32	6.30	12.2	9.85

Table 4.3: Cache dynamic differential versus static bus energy per access.

namic circuits. Dynamic differential busses have a fixed energy cost, while the energy usage of static busses depends on bit transitions. For the bus between the cache and datapath, a dynamic differential bus always uses 0.38 pJ per bit while a static bus uses 1.56 pJ per bit transition. The static bus uses up to 4 times more energy per bit, but Table 4.3 shows that with low transition frequencies it can be much more energy-efficient. The cache design modeled in this thesis uses static busses for addresses, and dynamic busses for data.

4.6 System Coprocessor

The majority of the energy in the system coprocessor (Coprocessor-0) is spent in the counter and comparator. There are many ways to minimize this energy consumption, including techniques at the circuit through operating system levels. Here, I use a simple technique similar to the one in Section 4.4 to eliminate most of the energy usage.

I segment the 32-bit dynamic count adder into a 3-bit static adder for the low-order bits and a 29-bit dynamic adder for the upper bits. The upper section is only clocked when there is a carry-out from the lower adder. This signal is also used to determine whether the upper bits of the count flip-flop should be clocked. I also segment the dynamic equality comparator into a 3-bit static comparator for the low-order bits and a 29-bit dynamic comparator for the high-order bits. The dynamic comparator is only activated when the low-order bits are equal.

Figure 4-17 shows the clock and output data activity for the count flip-flop; the clock activity for the upper 29 bits is reduced to 1/8th. Figure 4-18 shows the energy impact of these optimizations; the energy usage for the upper 29 bits is reduced to be less than that of the lower 3 bits. Overall the Coprocessor-0 energy is reduced by 63%.

4.7 Flip-flops and Latches

Flip-flops and latches (collectively timing elements, or TEs) are major consumers of energy in the datapath. Energy savings can be achieved by making two key observations: first, many TEs have considerable timing slack; and second, the clock and data activation patterns for different TEs vary considerably. In previous work, [24] provided a full energy characterization of a variety of flip-flop and latch circuit structures. In that work, I used the SyCHOSys Vanilla Pekoe simulator to gather detailed transition statistics for the flip-flop

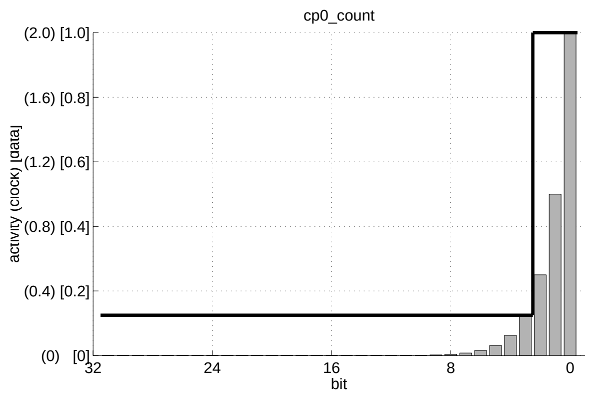


Figure 4-17: Segmented Coprocessor-0 counter flip-flop clock and data activity. The solid line shows the clock activity for each bit in the latch; 2.0 is the maximum clock activity. The bars show the data activity for each output bit of the latch; these are on a maximum scale of 1.0 transitions per cycle so that the data activity matches the clock activity even though it makes at most half as many transitions.

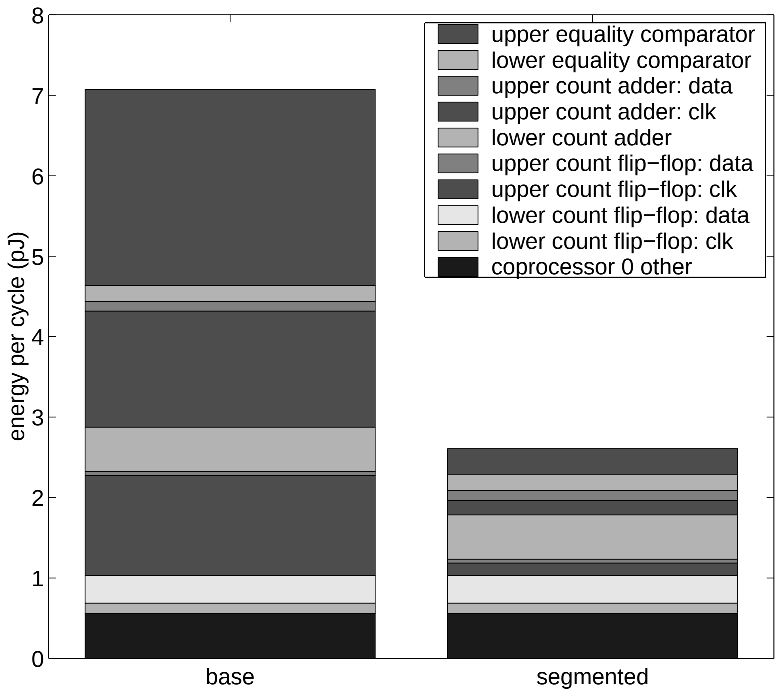
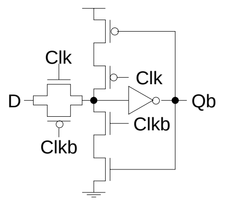
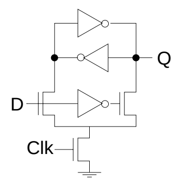


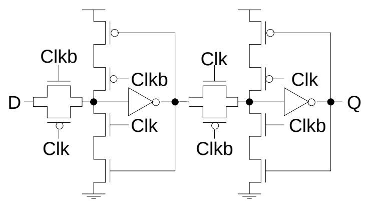
Figure 4-18: Segmented counter and comparator components energy. The bars show a breakdown of the entire Coprocessor-0 energy.



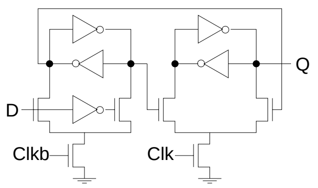
(a) PPCLA



(b) SSALA



(c) PPCFF



(d) SSAFF

Figure 4-19: Flip-flop and latch structures.

and latch clock and data signals. By choosing the structures which minimized energy for each non-critical TE in the datapath, large savings were achieved in comparison to a design which used fast TE structures uniformly. Here, I revisit this technique using a simplified approach.

Figure 4-19 shows the subset of flip-flop and latch structures I consider in this thesis, and Figure 4-20 shows their energy breakdowns. To allow arbitrarily low clock frequencies and to allow clocks that can be gated in either phase, the designs are restricted to be fully static. The transistors in each design were sized separately for both high-speed (hs) and low-power (lp) operation. PPCLA and PPCFF are based on the PowerPC 603 design, which is known to be reasonably fast and energy-efficient [41]. The other structures considered are the static sense-amp latch and master-slave flip-flop (SSALA and SSAFF). Compared to the PPC designs, the SSA TEs trade higher data transition energy for lower clock transition energy; this is intuitive since they have only 1/4 as many transistors driven by the clock. The SSA structures are also significantly slower than the PPC designs.

Figure 4-21 shows the final clock and data transition statistics for the datapath flip-flops and latches after applying the various optimizations discussed in this chapter. It is apparent that the various TEs have a wide range of activation patterns. This is interesting considering that previous analyses of flip-flop and latch structures (e.g. [41]) have usually assumed that the clock switches every cycle.

All critical paths in the design should always use the fastest flip-flop or latch available, in this case the high-speed PPC designs. For non-critical TEs, I choose the lowest energy structure based on the local clock and data activation patterns. Since most TEs in the datapath have lower data activity than clock activity the SSA structures are often an energy efficient choice. Some non-critical latches have low clock activity but somewhat high input data activity, for example the inputs to the the shifter (d_rsshmd, d_rtshmd) and the store-data path (d_sd). For these, PPCLA-lp provides the lowest energy consumption.

Figure 4-22 demonstrates the effect of using activity sensitive selection (AS) to choose flip-flop and latch structures for both the unoptimized design with minimal clock gating,

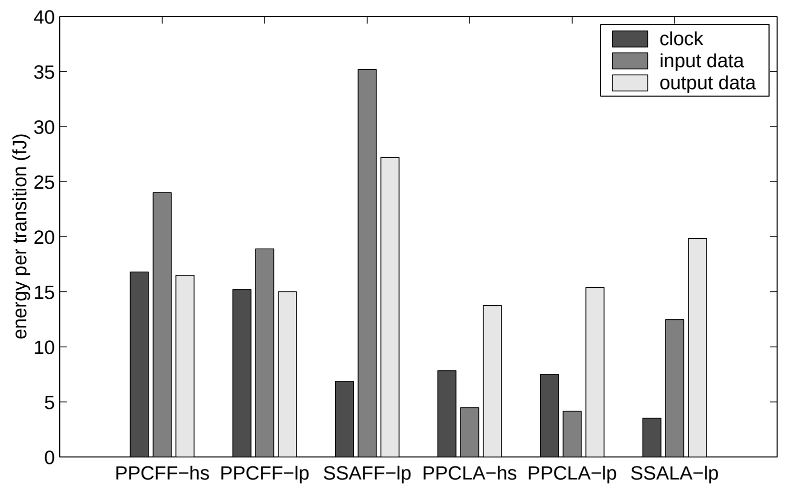
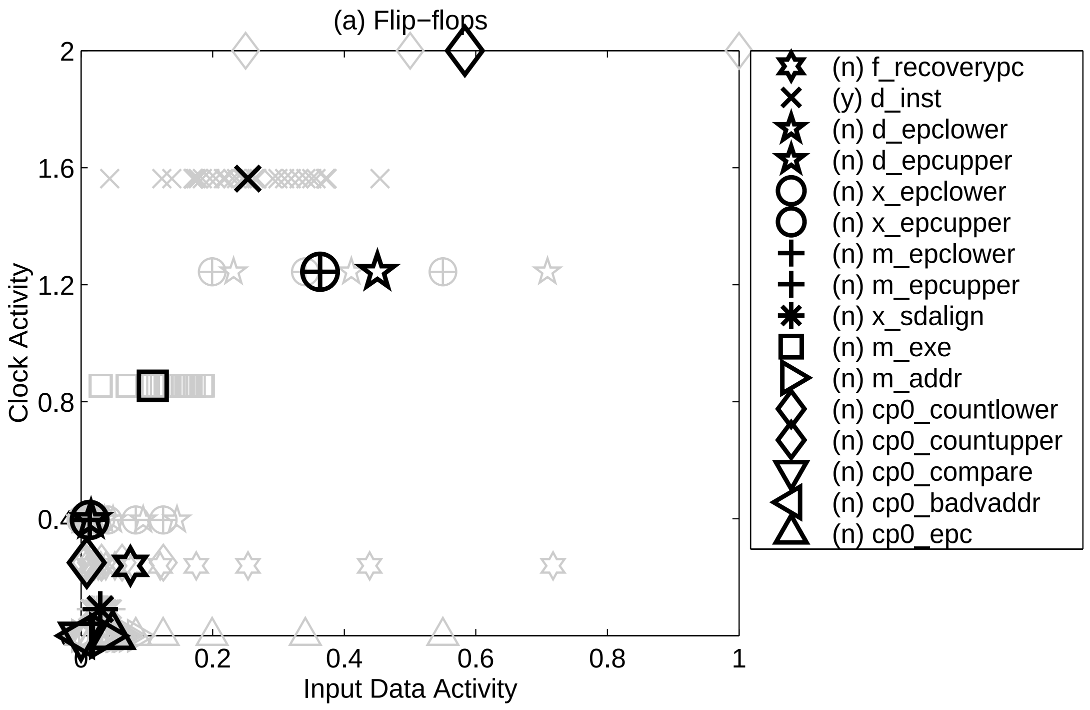


Figure 4-20: Flip-flop and latch energy breakdowns for various structures.

and the optimized design which includes the optimizations presented in this chapter. For each, the energy is divided between critical and non-critical TEs as well as clock and data. It can be seen that on average, the effect of AS selection is to tradeoff a small increase in data energy for a larger reduction in clock energy; this is the result of switching from PPC to SSA structures. An important point here is that an alternative structure is necessary to achieve this energy reduction; traditional transistor sizing would not be sufficient. It is also apparent that the energy savings are smaller when applying the technique to the more optimized design (14% compared to 23%) since a lot of the potential energy savings have been achieved through the other methods. Additionally, a larger energy savings is observed when comparing against a flip-flop structure that is faster but less energy-efficient [24]. In this thesis, an energy-efficient design was chosen for the baseline; however, an advantage of using a heterogeneous selection of TE designs is that critical paths can use the fastest design available even if it is not energy-efficient.

4.8 Combined Energy Optimizations

Figure 4-23 shows the per cycle energy for the processor with all the optimizations discussed in this chapter. Compared to the unoptimized energy consumption in Figure 4-1, the various optimizations combine to reduce the energy to around half of its original value. Figure 4-24 shows the savings with a side-by-side comparison for each benchmark, and Figure 4-25 shows the average savings broken down by sub-unit. In achieving this overall reduction, all of the optimizations play an important role. Comparing the optimized and unoptimized breakdowns, the energy savings are distributed across the collection of processor sub-components. The next chapter further analyzes the energy consumption of the optimized design.



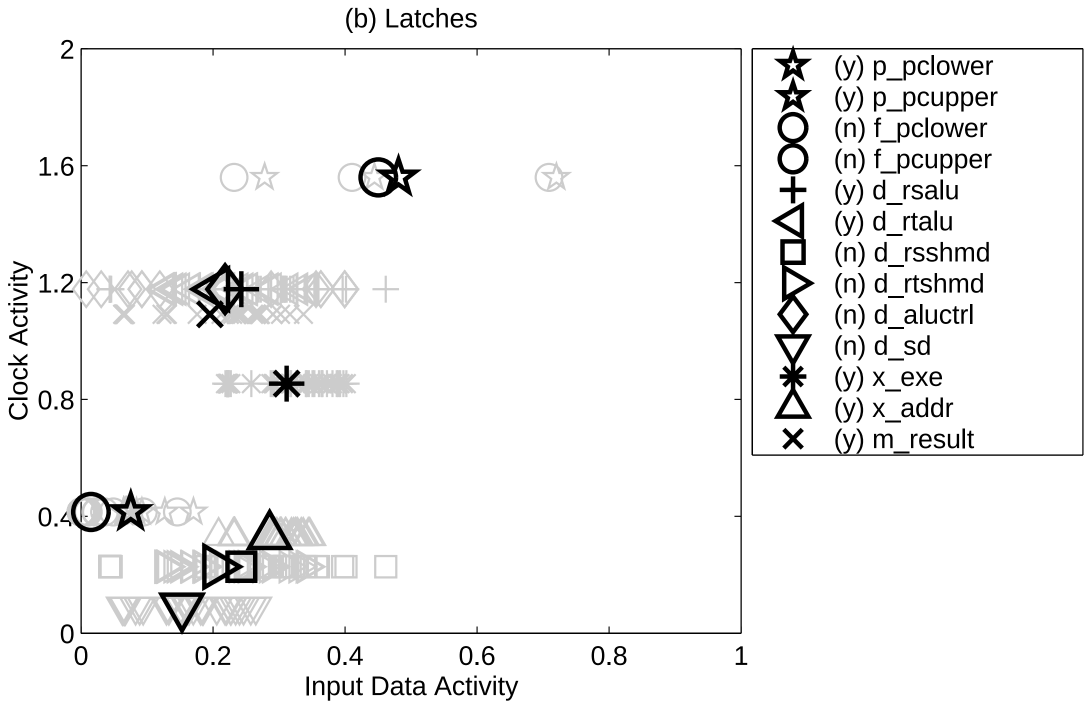


Figure 4-21: Flip-flop and latch clock and data activity in optimized datapath. Each plot shows the clock activity on the y-axis and input data activity on the x-axis; for each the activity is shown as the number of transitions (positive or negative edge) per cycle. The gray markers represent individual bits, while the black markers are averages for each multi-bit flip-flop or latch. In addition to the names of the TEs, the legends indicate whether or not each has critical timing (y/n).

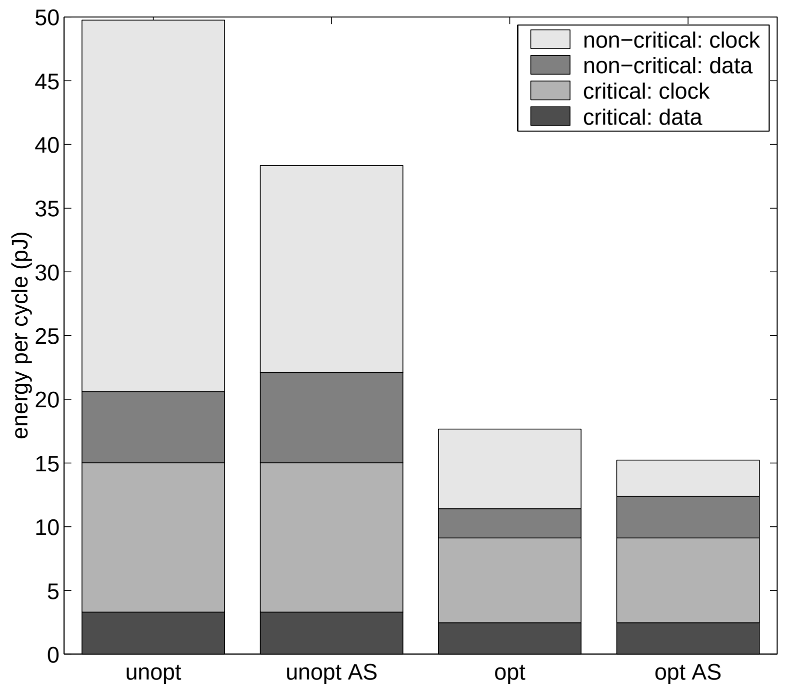


Figure 4-22: Flip-flop and latch activity-sensitive selection energy savings. The first two bars show the total flip-flop and latch energy for an unoptimized design both before and after applying AS; the second two bars show the same for an optimized design. Each bar is broken into clock and data energy for critical and non-critical flip-flops and latches.

4.9 Summary

In this chapter I have presented various energy optimizations that reduce energy consumption across the microprocessor design. General clock-gating based on instruction operand types and the liveness of instructions in the pipeline reduces the datapath (minus register file) energy by over 40%, mostly by spending less energy clocking flip-flops, latches, and dynamic adders. Clock-gating can be applied at different levels of granularity in some cases; for example, the ALU which has a 20% energy reduction when the adder and comparator are individually gated. Register file reads can also be gated based on instruction opcode, bypass logic, and the liveness of instructions in the pipeline to eliminate 70% of all reads and reduce the total register file plus bypass mux energy by 54%. Dynamic logic components like the register file can cause excessive switching activity since in the worst case their outputs have two transitions per cycle. However, reading inverted data from the register file exploits the prevalence of zero bits and reduces the register file plus bypass mux energy by an additional 31%. Specialized clock-gating in the high-order bits of the program counter components and the system counter reduces the energy for the upper 27 or 29 bits of these components to be less than that of the lower 3 bits. Eliminating intra-line-sequential tag checks in the instruction cache reduces its energy by 40%. Finally, taking advantage of the low data activity and timing slack for non-critical flip-flops and latches reduces their energy by 14%. Combined, these techniques reduce the processor datapath and cache energy by a factor of two.

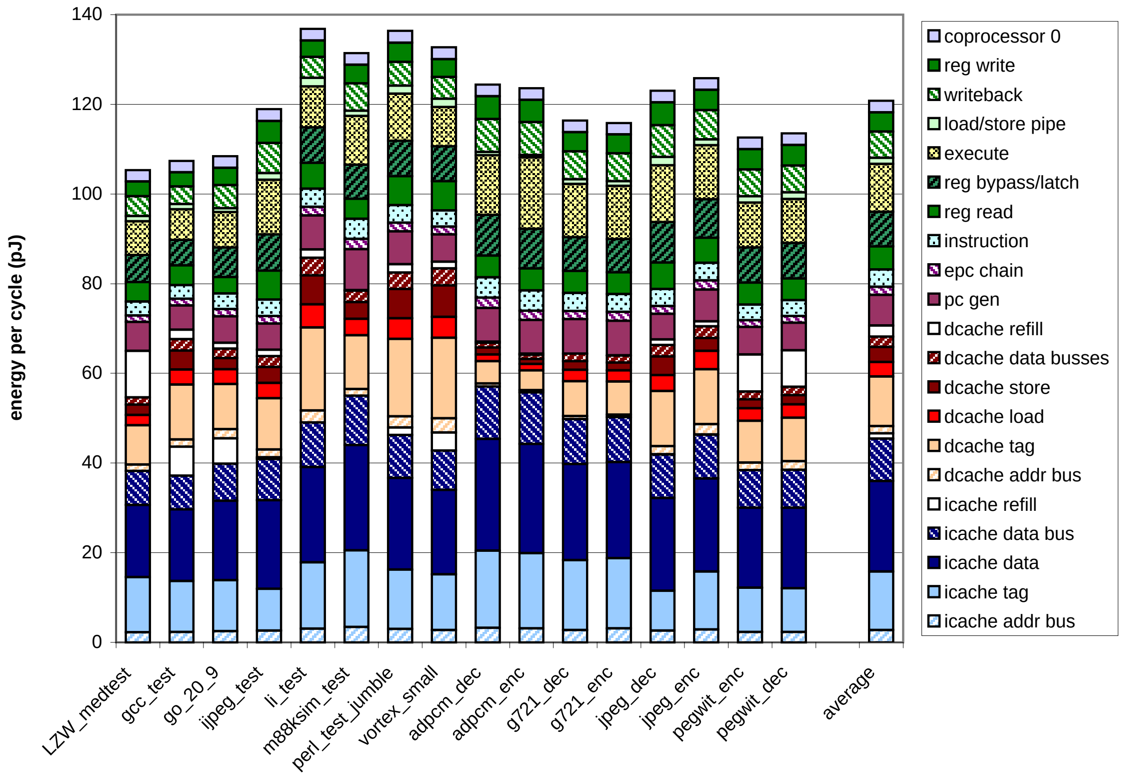


Figure 4-23: Optimized energy per cycle.

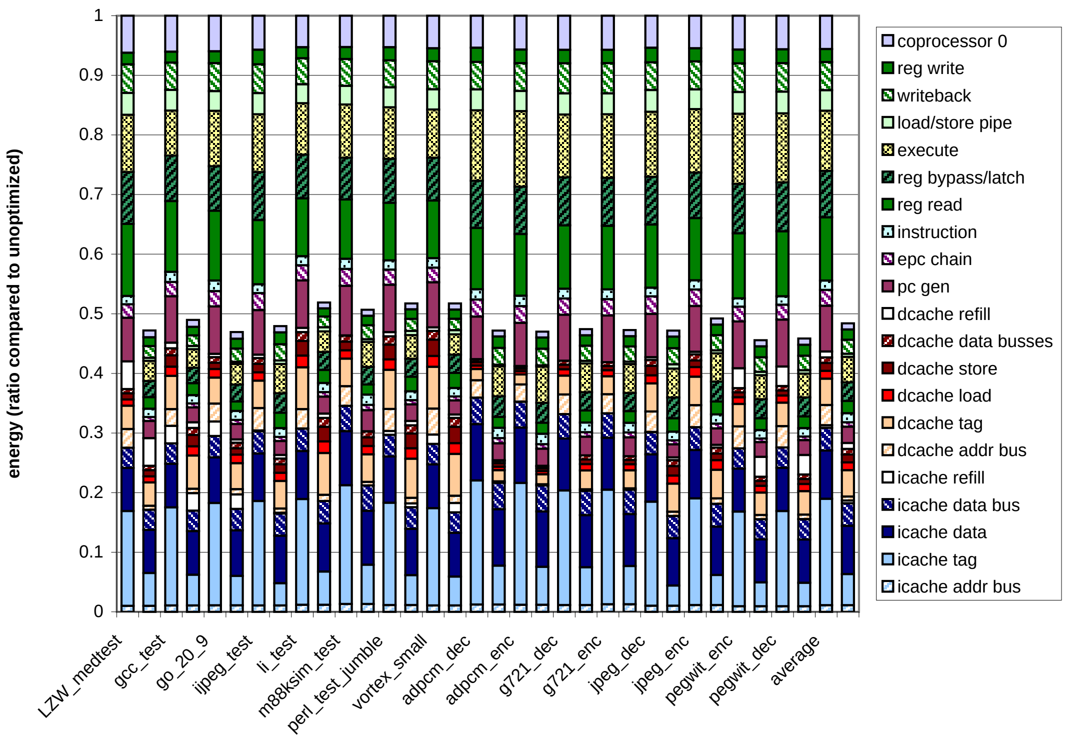
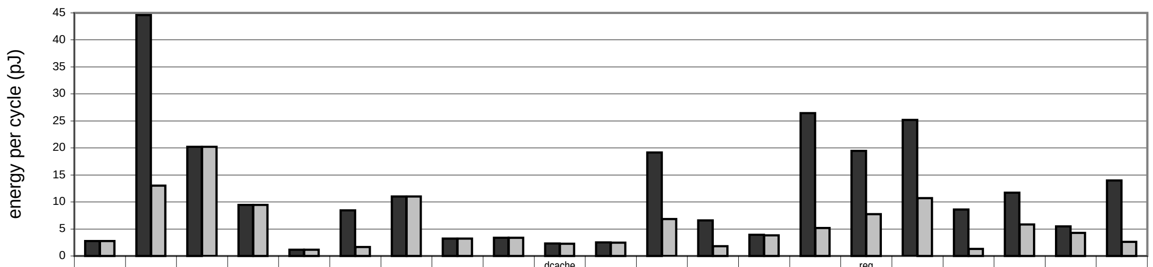


Figure 4-24: Energy savings with optimizations. The energy for each test is normalized to the unoptimized energy consumption.



	icache addr bus	icache tag	icache data	icache data bus	icache refill	dcache addr bus	dcache tag	dcache load	dcache store	dcache data busses	dcache refill	pc gen	epc chain	instr-uction	reg read	reg bypass/latch	exec	load/ store pipe	write-back	reg write	cp0
■ unoptimized	2.76	44.57	20.19	9.46	1.15	8.44	11.03	3.24	3.36	2.29	2.49	19.16	6.61	3.94	26.46	19.44	25.17	8.62	11.73	5.46	13.99
▨ optimized	2.76	13.03	20.19	9.46	1.15	1.68	11.02	3.24	3.36	2.29	2.49	6.83	1.81	3.82	5.18	7.74	10.69	1.32	5.86	4.29	2.61

Figure 4-25: Average energy savings breakdown.

The clock and data activities for the flip-flops and latches are good indicators for the general activity in the datapath. Figure 4-21 shows the final activity after applying the various optimizations. The general clock-gating described in Section 4.1 reduced the number of flip-flops and latches clocked per cycle to 271 out of 707 in the datapath. Adding the segmented gating for the PC, EPC, and system coprocessor counter further lowered the clock activity to 181 per cycle (26%). This is quite low, especially considering that clocking the instruction register, two ALU inputs, ALU control register, and three write-back registers involves 209 flip-flops and latches. Perhaps even more more remarkable is the low data activity; only 43 flip-flop and latch output bits toggle per cycle. There can be many ways to exploit the low data activity to achieve energy savings. The activity-sensitive flip-flop and latch selection presented in Section 4.7 and [24] is one example. The segmented PC components exploit low activity in the high-order bits of the program-counter values. Significance compression is an interesting technique which seeks to similarly gate the high-order bits of data values in the pipeline [9].

It is important to realize that there may be several overlapping ways of obtaining the same reduction in energy. Any proposed technique should be evaluated not only on the basis of its energy reduction, but also in comparison to other methods of achieving the same reduction. An example is the energy consumed in the exception program counter chain. Initially these 32-bit flip-flops are clocked almost every cycle and consume a considerable amount of energy. Segmentation (Section 4.4) is a microarchitectural technique that takes advantage of the fact that the high-order bits change much less frequently than the low-order bits, and partitions the flip-flops so as to avoid clocking the high-order bits most of the time. Activity-sensitive flip-flop selection (Section 4.7) is a microarchitectural/circuit technique which takes advantage of the low data activity and available timing slack to choose structures which minimize the energy consumption. Finally, an architectural technique alters the semantics of instructions so that the exception program counter must only be retained for a fraction of the executed instructions [22]. Some of these techniques may be complimentary, but a given quantity of energy can only be eliminated once and applying more than one technique will yield diminishing returns.

A significant consideration when reducing energy consumption is the performance impact. Complicated optimizations which reduce energy but significantly impact performance are generally useless since simply lowering a circuit's supply voltage has the same effect.

None of the optimizations presented in this chapter affect the processor's CPI rate. The processor clock frequency is also predicted to be largely unaffected since many of the optimizations exploit time slack to lower the energy consumption on non-critical paths. If energy reduction techniques do affect the critical path in a final implementation, it may not be worthwhile to include them. One option in this situation may be to make the optimization less critical; for example, it may be feasible to apply clock gating the second cycle after a cache miss even if there is not enough time during the first cycle.

Another important point about the tradeoff between energy and performance is that slowing down a component on a critical path in order to reduce its energy affects the performance of the entire circuit. Thus, even a large energy reduction in a critical component may not be worthwhile if it affects performance. A case in point is flip-flop and latch structures. Previous studies of energy-efficient flip-flops and latches have aimed to minimize the energy-delay product of these devices. A better solution is to use a heterogeneous selection of designs, as described in Section 4.7. Compared to a design which uses heterogeneous TE structures, a design which uses TE structures with optimal energy-delay product will have slower critical paths and will use more energy on non-critical paths; thus, it will actually be slower and consume more energy!

I found SyCHOSys to be a very useful tool in performing the energy studies in this chapter. I was able to use a single simulator executable with runtime options to enable the various optimizations. However, both a positive and negative implication of implementing a cycle-accurate structural simulation is that the model must be detailed and correct enough to actually work. It was non-trivial to implement such features as clock-gating and PC segmentation; but doing so uncovered some interesting problems and made the final result more credible.

Chapter 5

Energy Characterization and Analysis

In this chapter I evaluate a design which incorporates all of the individual optimizations described in the previous chapter. The goal is to provide a full characterization and analysis of an energy-efficient microprocessor, and to investigate the potential for further energy reductions.

In comparison to many previous studies, this chapter provides more detailed energy breakdowns which are based on longer simulations. Raw simulated energy numbers are not very useful in architectural and microarchitectural studies. It is important to develop a sense of where the energy is consumed, and to evaluate the causes of the energy usage. Additionally, a full energy characterization is necessary in evaluating the impact of a proposed improvement. Furthermore, many aspects of a processor's energy consumption can only be revealed by analyzing the average behavior observed over full benchmark simulations. Therefore, the results in this chapter show both individual and harmonic mean breakdowns for a large set of benchmark programs.

Ultimately, dynamic energy consumption in a processor is caused by circuit nodes toggling. Some of this activity is inherent to the overheads involved in fetching and executing instructions, while some activity depends on the actual data values moving through the circuit. In this chapter I provide a detailed analysis of the bit transition activities on a variety of nodes in the processor. I also evaluate the bit transition overhead caused by time-multiplexing different data types on the same physical hardware, and I investigate spatial separation to eliminate this overhead.

5.1 Energy Breakdowns

5.1.1 Overall

Figure 5-1 shows the energy per instruction for each benchmark program. This measurement is more useful than energy per cycle in analyzing the energy efficiency of a program execution because stall cycles reduce the per cycle energy but adversely affect performance. The energy breakdowns exhibit relatively little variance across the different benchmarks. A primary reason for this is that RISC instruction sets are designed so that all instructions tend to have similar complexities. Each instruction is the same size, and the energy per instruc-

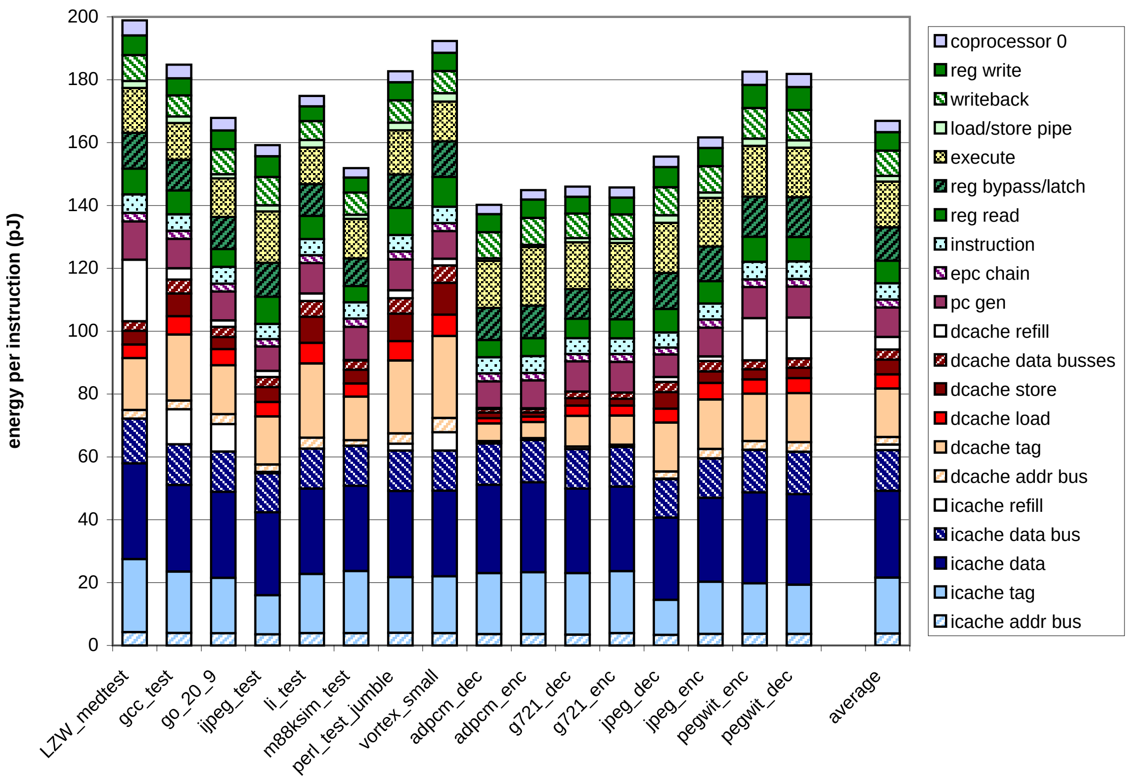


Figure 5-1: Optimized energy per instruction.

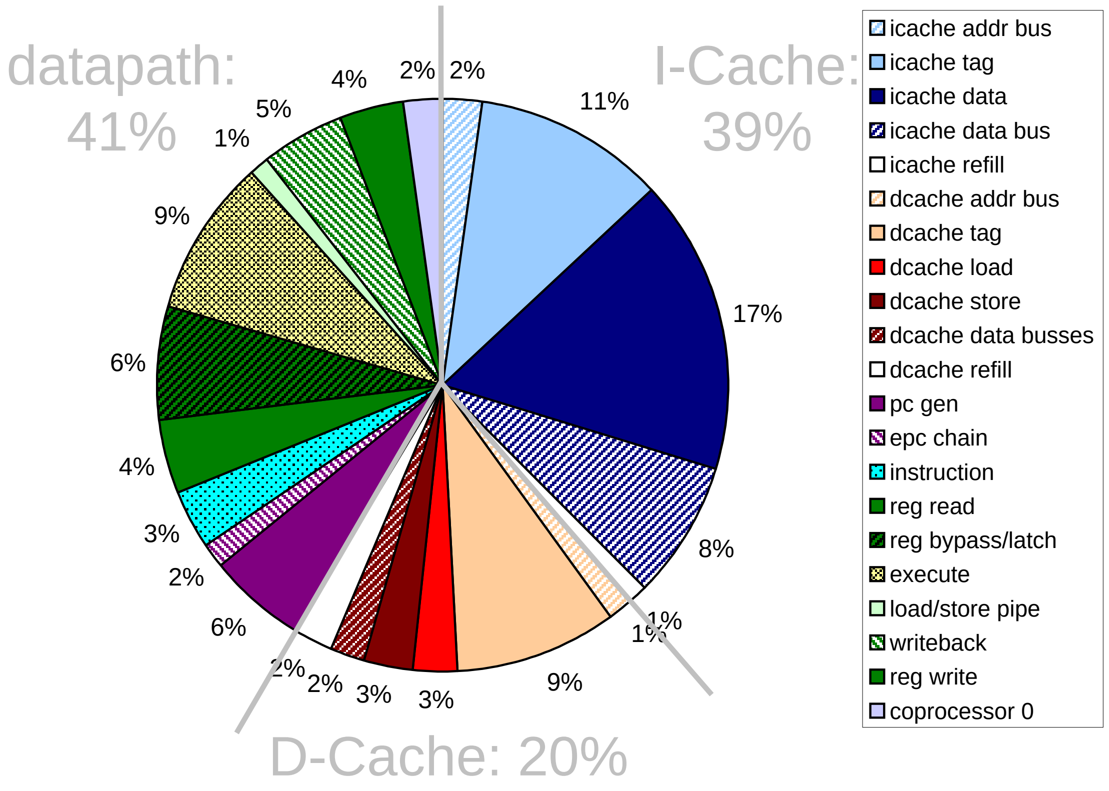


Figure 5-2: Optimized average energy breakdown.

tion spent in the instruction cache varies only a small amount across the benchmarks. Most instructions read two values from the register file, perform a simple operation, and write a result back to the register file; so, the datapath energy numbers also tend to be consistent for the different benchmarks. The biggest variance between the benchmarks is due to the data cache energy numbers. For example, vortex spends around five times more energy per instruction in the data cache compared to the adpcm benchmarks. This is simply caused by the percentage of load and store instructions; these have a big impact on the average energy per instruction since they use far more energy than other instructions.

Figure 5-2 shows the average energy breakdown for the various benchmarks. Broken down coarsely, around 40% of the energy is spent in the instruction cache, 20% in the data cache, and 40% in the datapath. This breakdown can be compared to the unoptimized breakdowns shown in Figure 4-1; in these the split is about 31% instruction cache, 12% data cache, and 57% datapath. Even though the absolute instruction cache energy was reduced by about 40% (Section 4.5), its relative energy has increased. The energy cost just to read the instruction data word from the cache and send it to the datapath is around 25% of the total energy per instruction. This is a fixed cost which is very hard to improve without changing the instruction set. It is intrinsic to general purpose programmable RISC processors and will be a bottleneck in further reducing energy consumption. The remaining energy is widely distributed across the processor sub-components. Optimizations which target individual sub-components can have limited impact when performed in isolation.

It is interesting in the energy breakdown figures to look at the *instruction* sub-component which represents a single 32-bit flip-flop that is clocked approximately once per instruction. The energy consumption for the entire datapath plus caches is 31.6 times this reference; a very low number, especially considering that the chip area could accommodate around 3,500 32-bit flip-flops. This result is expected since most of the chip is occupied by cache banks which are inactive most of the time. However, it hints at the degree to which leakage currents will be important in future energy-efficient processors. Static leakage currents are the only form of energy consumption while a processor is inactive, and as dynamic energy is reduced they become more significant during active operation [7].

Notably absent from these breakdowns is the control logic energy consumption. Most processor energy studies ignore the control logic energy since it is hard to model and is assumed to be a small fraction of the total energy. Although possibly a small fraction of the unoptimized processor, the control logic energy will certainly become significant when the rest of the system is optimized. For example, control accounts for 25%–30% of the energy in the XScale processor (with 32 KB instruction and data caches) [16], and 21% in a processor based on an ARM8 core (with 16 KB unified cache) [2]. The energy optimizations presented in this thesis have not lowered the control logic energy at all, and in some cases have added extra control overhead. Implementing a synthesis path for the control logic and modeling its energy consumption will be a subject of future work. It will be important to optimize the control logic, but this can be difficult due to its irregularity.

5.1.2 Datapath Units

The breakdowns in the previous section group many components together in each subdivision. For reference, Table 5.1 shows a more detailed energy breakdown for the average

	energy per cycle (pJ)	datapath percent
pc gen: pc-incr	0.7	1.3
pc gen: brtarg adder	2.9	5.9
pc gen: flip-flops/latches	1.3	2.5
pc gen: mux	1.4	2.7
pc gen: other	0.6	1.2
epc chain	1.8	3.6
instruction	3.8	7.6
reg bypass/latch: regfile-read	5.2	10.3
reg bypass/latch: bypass-rs	1.9	3.9
reg bypass/latch: bypass-rt	1.3	2.7
reg bypass/latch: bypass-sd	0.7	1.4
reg bypass/latch: alu rs latch	1.3	2.6
reg bypass/latch: alu rt latch	1.3	2.5
reg bypass/latch: shmd rs latch	0.3	0.6
reg bypass/latch: shmd rt latch	0.3	0.6
reg bypass/latch: sd latch	0.2	0.4
reg bypass/latch: other	0.4	0.8
execute: alu	9.8	19.4
execute: shifter	0.9	1.9
load/store pipe: data	0.7	1.4
load/store pipe: address	0.6	1.2
writeback: flip-flops/latches	3.6	7.3
writeback: muxes	2.2	4.4
regfile write	4.3	8.6
coprocessor 0: counter	1.8	3.5
coprocessor 0: eqcheck	0.5	1.0
coprocessor 0: other	0.3	0.6
total	50.1	100.0

Table 5.1: Datapath average energy breakdown detail. In general, energy dissipated on inter-component nets is included with the component that drives it.

datapath energy consumption.

Looking at the program-counter generation energy, the segmentation discussed in Section 4.4 lowered the energy enough to make the branch-target adder one of the most significant components even though it is gated most of the time. Another reason for the relatively high energy consumption in the branch-target adder is that one of its inputs is the sign-extended immediate field of the instruction which has much higher bit activity than the PC values.

This more detailed breakdown also shows how the energy usage is concentrated around the frequently used paths. Particularly, the ALU path consumes much more energy than the shifter or memory access portions of the pipeline.

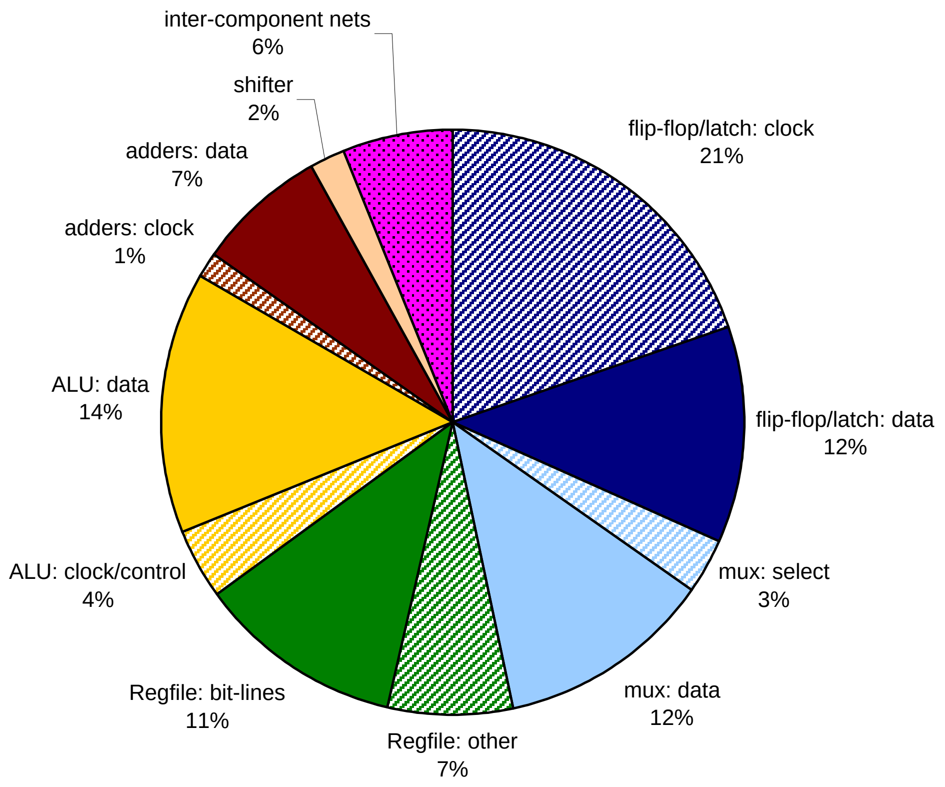


Figure 5-3: Datapath energy breakdown by component type. Each component type is further divided into data energy and clock/control energy.

5.1.3 Datapath Component Types

Figure 5-3 shows the average datapath energy consumption broken down into the various component types. It also shows how the energy is divided between data and overhead due to clocking and control.

The ALU and Regfile can be considered the work-horses of the datapath, and a large effort goes into their design. They use fast dynamic logic circuits and have carefully optimized structures. They each account for 18% of the datapath energy consumption. Including the other adders and the shifter, these specialized circuit designs account for almost half of the total energy.

Combined, the energy of the flip-flops, latches, muxes, and inter-component nets accounts for over half of the datapath energy. This is interesting since these components perform no computation; they are the glue that holds the datapath together. Furthermore, these components have relatively simple structures and circuit techniques are unlikely to lower their energy considerably; achieving lower energy will most likely require a reduction in their data and clock/select activation patterns. The breakdown demonstrates that energy-aware datapath designs must be very concerned with every flip-flop, latch, and mux on frequently active paths. This is in contrast to performance-oriented designs which can pipeline and insert muxes in non-critical timing paths at will.

The flip-flop and latch clock energy was lowered considerably with the various applications of clock gating and activity sensitive selection in Chapter 4. However, clocking these components still consumes 21% of the total datapath energy. Overall, the overhead for clocking, control, and select lines accounts for around 36% of the datapath energy. This energy is mostly fixed for a given mix of instructions since it does not depend on the bit transitions caused by the actual data values. One way to further reduce the flip-flop and latch clock energy is to use pulse-clocked latches in place of master-slave flip-flops and latches; by only requiring a single latch per clock period, this technique can both save energy and improve performance, but it also complicates timing requirements. The XScale design reduces clock power by approximately 1/3 by using pulse-clocked latches [15].

5.1.4 Instructions

From the point of view of software, the energy consumption of individual instructions is more useful than average datapath and cache energy breakdowns [30, 42]. Unfortunately the energy cost of an instruction is not a well-defined quantity. As it goes down the pipeline, energy such as clocking pipeline registers can be considered intrinsic to an instruction; but, much of the energy consumption of an instruction depends on inter-instruction effects. This includes bit transition activity which depends on the previous node values, and effects such as stalls and register result bypassing which depend on the other instructions in the pipeline. Additionally, even before it is executed, an instruction must be brought into the cache.

Table 5.2 presents a rough idea of the energy consumption by instruction type. To construct this table, I started with the average energy consumption breakdown. Then I scaled the energy for each sub-unit by its activation rate to obtain the average per-use energy or the average per-instruction energy if the sub-unit is active for every instruction. Thus, the energy numbers are representative of the average transition activities and take into account effects such as stall cycles and instruction cache misses. The table also shows several instruction types and which sub-units they use or cause activity in. Summing the associated energy costs gives the average energy usage of each instruction type.

These results reiterate the behavior observed in Figure 5-1. RISC instructions are designed to have similar characteristics, and for the most part they have little variance in energy consumption. Almost all the instructions read values from the register file and produce a result which is written back. Another remarkable feature is that the fixed energy costs associated with the program-counter and instruction fetch units consume around 60% of the total energy for most instruction types. This cost even neglects the instruction decode and other control logic overhead which will also be relatively independent of instruction type. Thus, most of the energy for executing an instruction does not depend at all on what the instruction does.

The instructions which stand out as being different in Table 5.2 are the load and store instructions which access the data cache. Even assuming that the accesses hit in the cache, these instructions consume almost twice the energy of the other instructions. A major part of this energy goes into the data cache tag check; potentially this can be eliminated as discussed in Section 4.5. Additionally, many loads and stores access temporary data with high temporal locality (e.g. the program stack); a more optimal implementation might provide alternative instructions with lower energy cost for such accesses.

		add	add (bypass)	addi	and	sll	lw	sw	beq	lui	slt	nop
per-instruction (fixed)												
sub-unit	energy (pJ)											
pc-gen (no brtarg)	7.9											
epc chain	2.5											
i-cache	65.2											
instruction reg	5.3											
regfile decoder	1.8											
coprocessor-0	3.7											
total	86.4	√	√	√	√	√	√	√	√	√	√	√
per-use (variable)												
sub-unit	energy (pJ)											
brtarg adder (clock/carry)	11.9								√			
regfile read rs	8.0	√		√	√	√	√	√	√		√	
regfile read rt	4.9	√	√		√	√	√	√	√		√	
bypass muxes	6.3	√	√	√	√	√	√	√	√	√	√	
bypass latch inputs	0.7	√	√	√	√	√	√	√	√	√	√	
alu rs latch (clock/output)	1.9	√	√	√	√		√	√	√		√	
alu rt latch (clock/output)	1.9	√	√	√	√		√	√	√		√	
sh/md rs latch (clock/output)	1.6					√				√		
sh/md rt latch (clock/output)	1.8					√				√		
alu: adder clock/carry	6.3	√	√	√			√	√				
alu: comparator	0.7								√		√	
alu: other	11.4	√	√	√	√		√	√	√		√	
shifter	9.5					√				√		
writeback: x muxes, x latch inputs	2.5	√	√	√	√	√	√	√	√	√	√	
d-cache/pipe: load hit	109.4						√					
d-cache/pipe: store hit	145.2							√				
writeback: x/m	5.8	√	√	√	√	√				√	√	
writeback: m/w	4.0	√	√	√	√	√	√			√	√	
regfile write	7.2	√	√	√	√	√	√			√	√	
total		147	139	142	141	139	251	276	137	126	142	86

Table 5.2: Energy breakdown by sub-unit and estimated average energy for various instruction types.

5.2 Bit Correlation

In this section I analyze the bit level correlation for nets in the design. Recalling the energy consumption equation introduced in Section 2.2, the energy dissipated on a given bit is directly proportional to its switching frequency. Bits toggle as values propagate through the circuit, and here bit correlation refers to the frequency with which *new values* cause a given bit to transition. The purpose of this definition is to eliminate the effects of clock gating so that the actual correlation between values can be analyzed. Figure 5-4 shows the bit correlation for a selection of nets in the processor; the results show the harmonic mean for the six benchmarks used in Chapter 4. In Figure 5-5, I choose a few nets and show their activities for the various benchmarks.

Perhaps the first thing to notice is that there tends to be a high degree of correlation between values. For static nodes, the maximum frequency is 1.0 and white-noise assumptions would yield transition frequencies of 0.5; however, most of the observed frequencies lie well below these values. The ALU output (x_alu) and the register file outputs (d_rx, d_ry) exhibit increased activities because these dynamic components cause up to two transitions in a cycle when a precharged bit discharges during the evaluation phase. The correlations also show significant variations across the different benchmarks. For example, in Figure 5-5 the load data bus (m_ld) has very high switching activities for adpcm_dec since this benchmark accesses compressed data with high entropy.

For the program counter nodes and the system counter, the switching activity decreases exponentially from low-order to high-order bits. Although to a lesser extent, most other nets in the datapath exhibit a similar decrease in activity from low to high order bits. This is caused by the tendency of programs to use small data values, and the prevalence of counters and sequential memory accesses. A good example is the data cache address bus (x_addr) for m88ksim_test, and another is the store data bus (x_sdalign) for adpcm_enc which exhibits the effect twice due to half-word stores. One exception to this trend is that since data cache addresses are usually word-aligned x_addr and several other nodes show reduced activity in the lowest two bits. The activity decrease from low-order to high-order bits is even visible in the register specifier fields of the instruction word (bits 16-20 and 21-25 of d_inst). Here, the compiler's policy of allocating registers sequentially from a list of free registers creates a usage pattern which causes more switching in the low-order specifier bits. [48] discusses a technique which relabels the register specifiers to reduce this activity and purportedly reduces the I-Cache data bus energy by 12%.

An effect exhibited by many of the nets is a high switching activity for bit 31. One might think that the switching is caused by alternating between positive and negative values, but this cannot be true since the other high-order bits have very low switching activities. The reason for this activity actually stems from the Vanilla Pekoe virtual memory system shown in Figure 5-6. The processor does not implement any form of memory translation, but a simple protection scheme segments the 32-bit virtual address space into two halves based on bit 31; user addresses always have a one in this bit, while addresses with a zero in bit 31 are allowed only in kernel mode. The high switching activities for bit 31 are a result of nets in the design alternating between user address values with a one in bit 31 and data values which more often than not have zeros in the high order bits. The effect is most prevalent in the rx/RS register path since RS is used as the base address for register-plus-offset memory

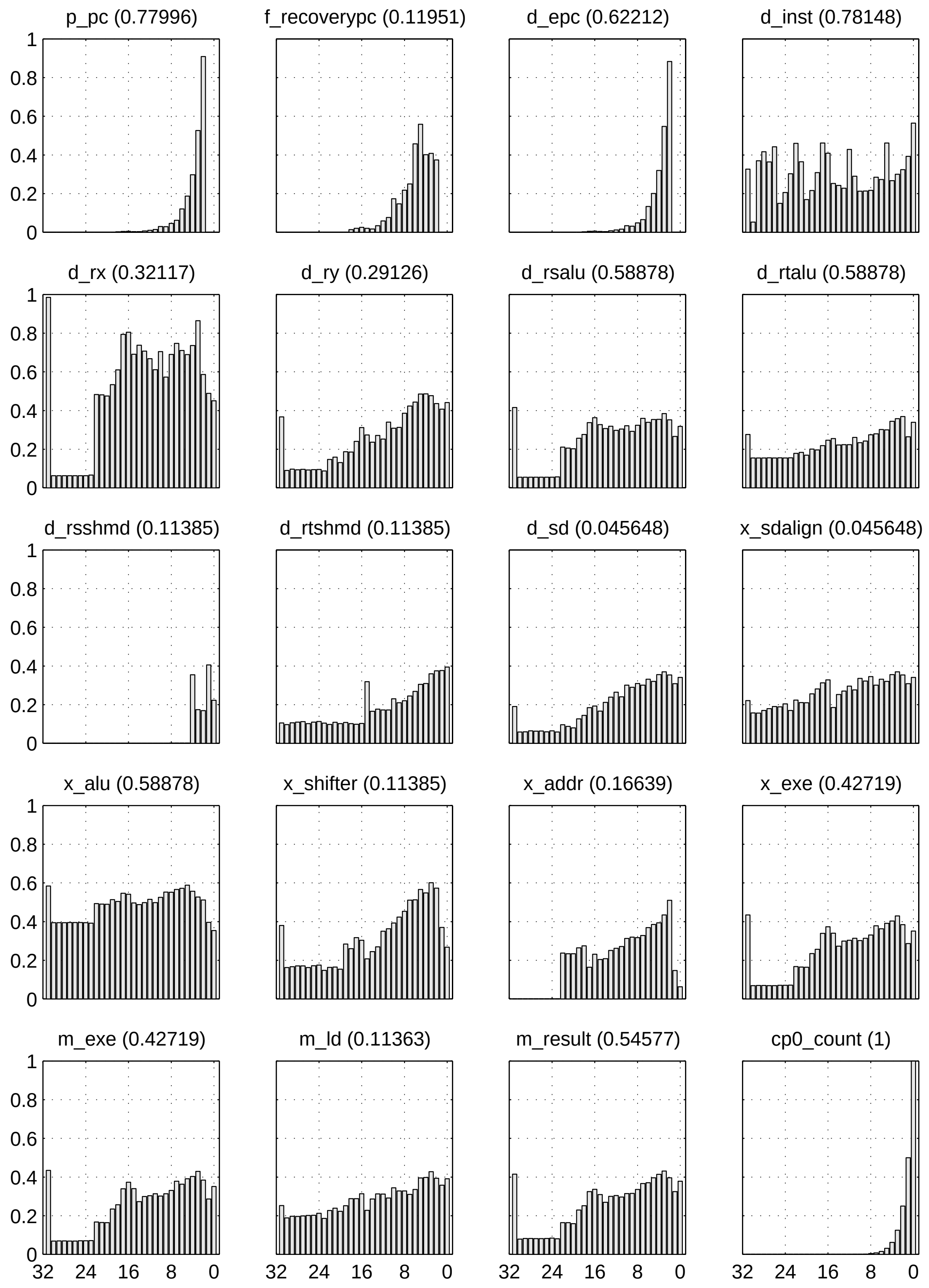


Figure 5-4: Bit correlation for various nets. Each plot shows a bar for each bit in the net (with low-order bits on the right), and is labeled with the name of the net as well as the average rate at which new values are introduced (e.g. the clock activity for the preceding flip-flop or latch). The vertical axis shows the new value switching frequency where 1.0 indicates one transition (positive or negative edge) for every new data value; the actual switching frequency is this value multiplied by the new value rate.

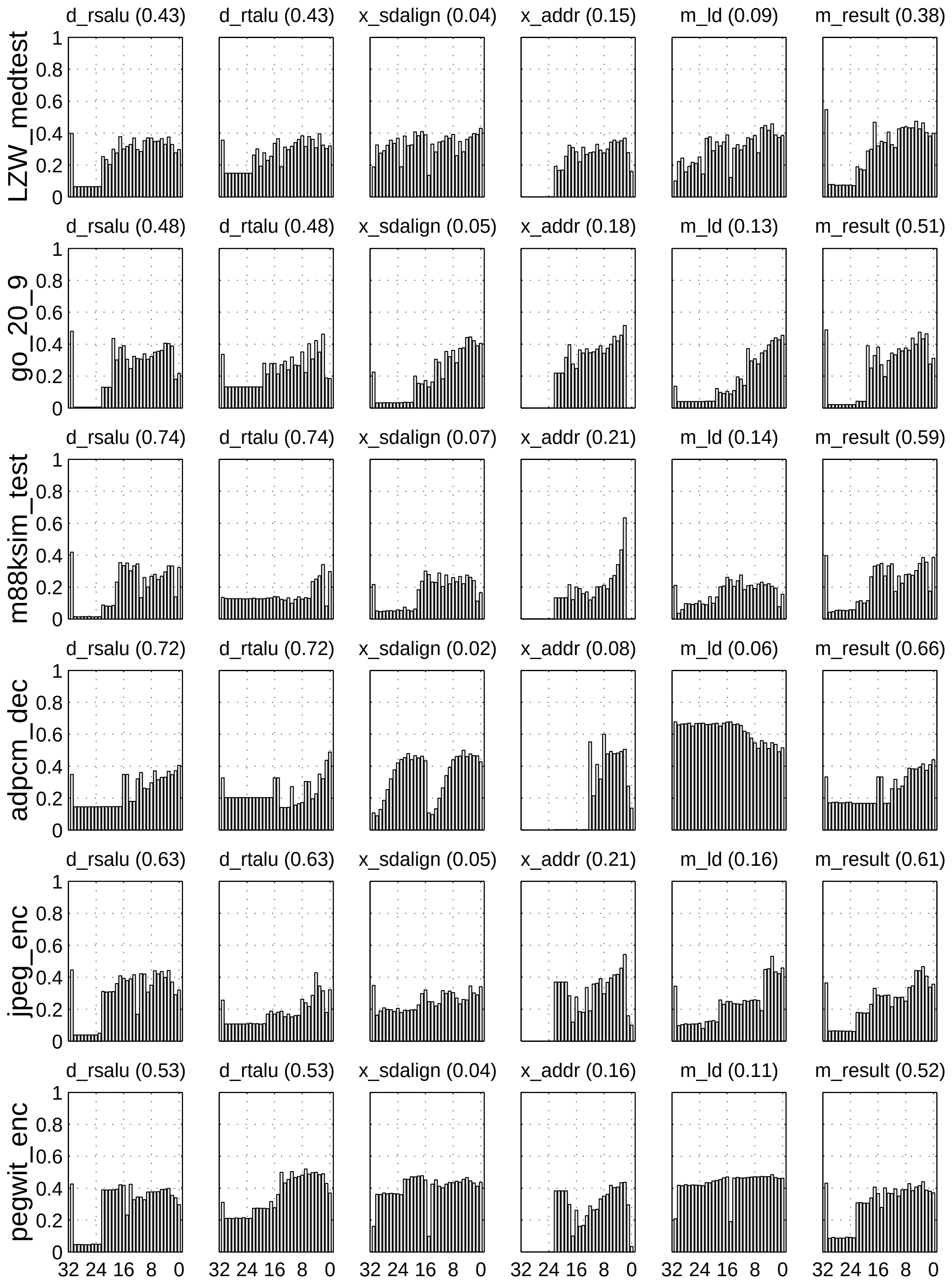


Figure 5-5: Bit correlation for various nets in different benchmarks. See Figure 5-4 for a description of the plots.

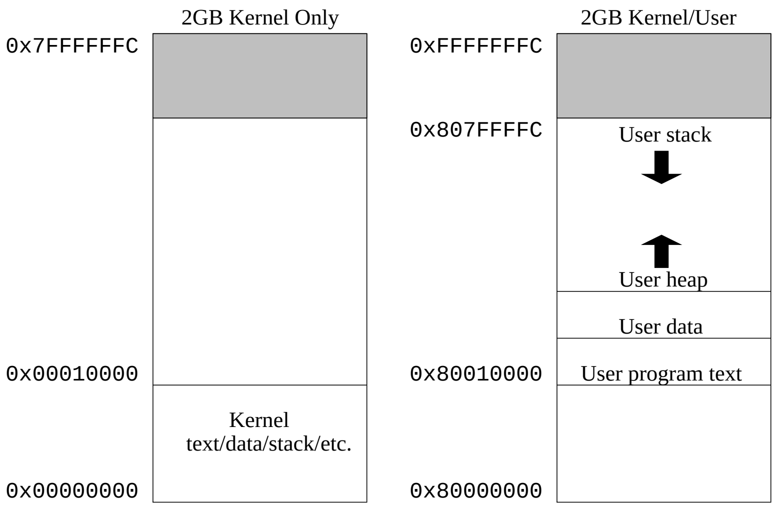


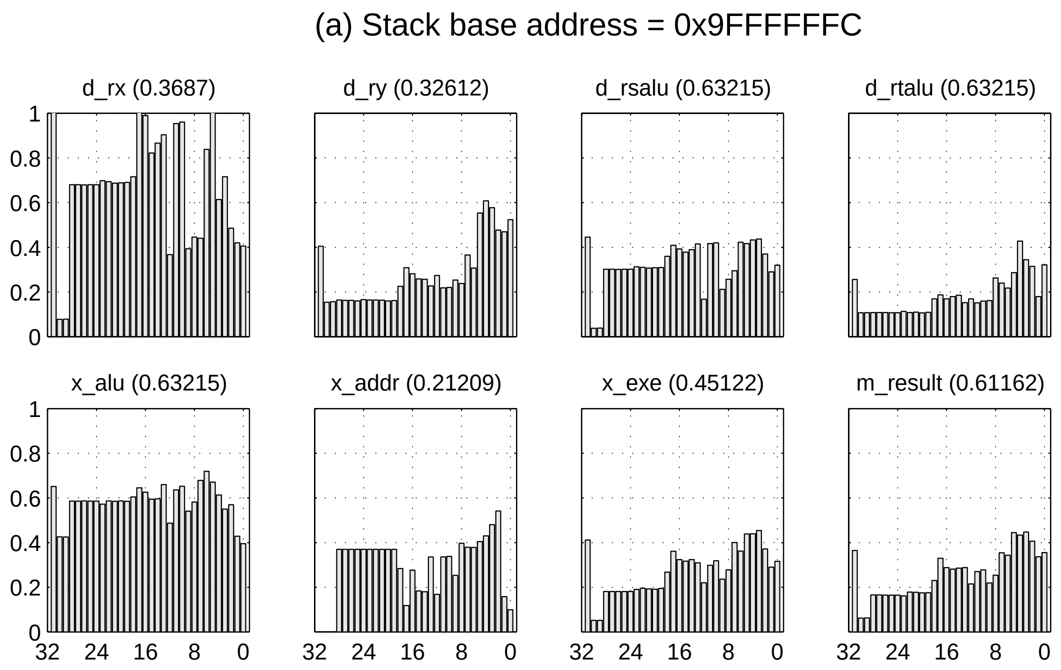
Figure 5-6: Vanilla Pekoe memory map (8 MB physical memory size).

addressing. Another place where it appears is bit 15 of the shifter input (d_rtshmd); this is caused by the compiler's use of load-upper-immediate instructions to access static memory locations with 32-bit immediate addresses. Based on the data in Figure 5-4, for many nets around 35% of new values cause a transition between an address value and a data value. This effect was also observed in a similar study which tracked bit transitions [5].

Another effect exhibited by many of the nets in Figure 5-4 is a low switching activity for bits 23 through 30 but a relatively sharp increase in switching activity for bits 22 and lower. Again, this effect can be traced to a subtlety of the Vanilla Pekoe memory system (Figure 5-6). The kernel manages the available physical address space such that the program stack starts at the top and grows downwards. So, with a physical memory size of 8 MB (23 bits, 0x00800000) the top of user memory is `0x807FFFFC`, and a typical stack address could be `0x807FFF00`. Since the stack does not typically grow far from the base, stack addresses have a block of bits which are all ones up until bit 22. Thus, the observed activities are caused by net values alternating between stack addresses and other values. This effect is even apparent in the data cache address bus (x_addr) because the stack addresses do not correlate well with heap and static memory addresses.

To analyze the impact of the switching activity caused by stack addresses, I chose one benchmark (jpeg_enc) and ran it with the stack base address set to two different values. In each case, the program executed exactly the same instructions, and the data cache operation was unaffected. Figure 5-7(a) shows the bit correlations for selected nets when the stack base address is `0x9FFFFFFC` and 5-7(b) shows the correlations when the base address is `0x8007FFFC`. Figure 5-8 shows the impact on energy consumption when the stack base address is set to the two different values.

For several nets, the difference in switching activity for bits 19-28 is dramatic. The activity for these 10 bits in the rx/RS register path is reduced by around 85%. This is remarkable since it indicates that the other instructions in the program rarely toggle these bits, and almost all of the activity is caused by switching between stack addresses and other values. Based on the jpeg_enc data for d_rsalu in Figure 5-7, about one in every



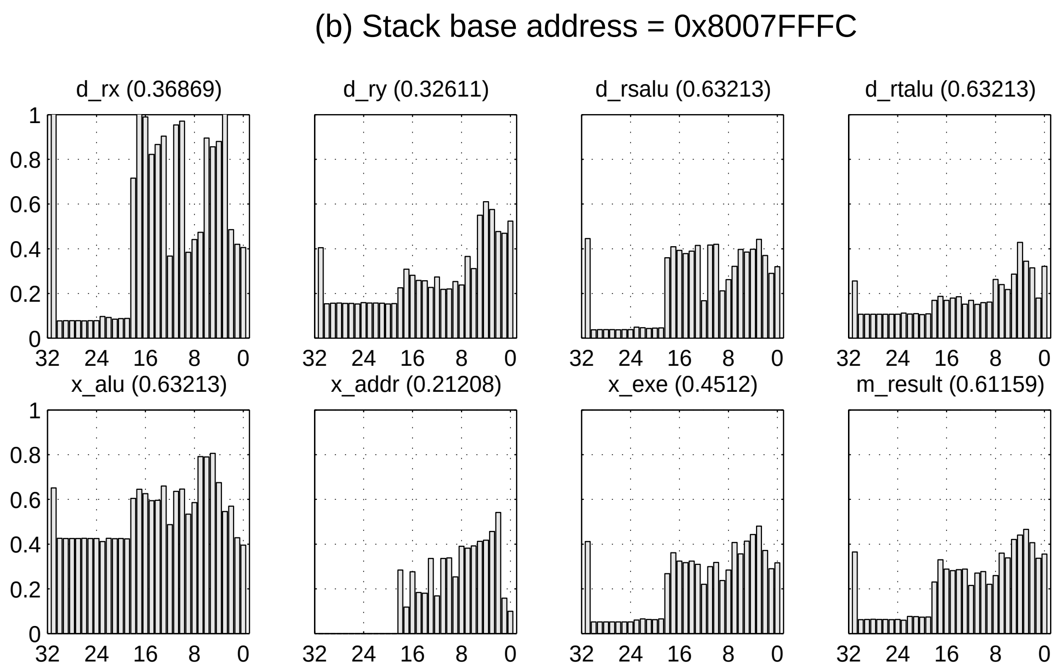


Figure 5-7: Bit correlations for selected nets using two different stack base addresses (jpeg_enc).

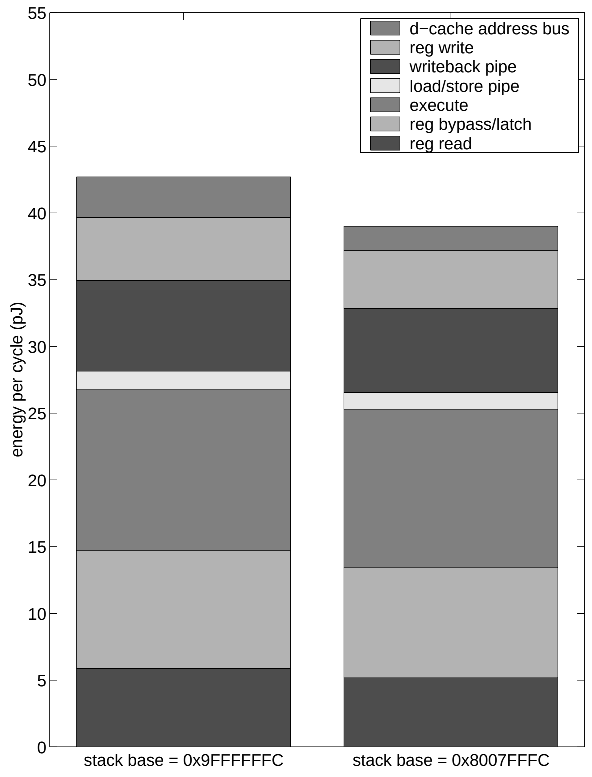


Figure 5-8: Energy using two different stack base addresses (jpeg_enc). The energy breakdown includes components in the memory access and execution segments of the pipeline.

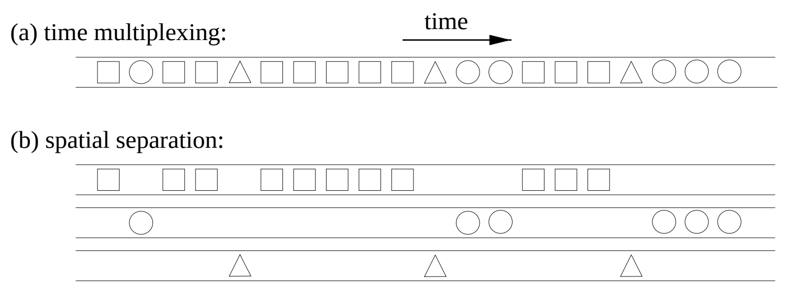


Figure 5-9: Time-multiplexing versus spatial separation.

three values (33% of new values) is a transition between a stack address and another value; Figure 5-4 indicates that the average across the benchmarks is around 15%. The stack addresses also cause excessive activity on the data cache address bus (x_addr) since they do not correlate with heap and static memory addresses. Figure 5-7(a) shows that more than one in every three memory accesses (37%) incurs a transition between a stack address and another address causing the upper bits to toggle; averaged across the benchmarks (Figure 5-4) the rate is about one in four (25%).

5.3 Spatial Separation

The energy impact of the high activity rates for bit 31 is not in itself alarming, but it points out a more important problem. Address values tend to have a high degree of bit level correlation, as do data values; but time-multiplexing address and data values over the same datapath causes excessive switching activity. Even intermixing stack and heap addresses[1] can lead to unwanted bit transitions. The previous section showed that setting the base stack address to correlate better with heap addresses can reduce this activity. However, a more ideal solution is to use separate pipelines to operate on different types of data values rather than time-multiplexing them over the same hardware. Figure 5-9 illustrates this point where the different shapes might represent values of different data types occupying a net or passing through a pipeline stage over time. By duplicating the shared resource spatially, the overall correlation between values improves.

A full implementation of such an architecture requires support from the compiler, but this thesis presents a theoretical study which analyzes the potential of separating these data values. In addition to the absolute bit transition statistics, for various nets in the design I tracked the bit transitions between stack addresses, heap addresses, and data values assuming that the net was duplicated spatially as in Figure 5-9(b).

Figure 5-10 shows the same bit correlation statistics as Figure 5-4; except, each bar is broken down into segments which show the transition activities intrinsic to data values, heap addresses, and stack addresses. The remaining portion of each bar reveals the overhead of time-multiplexing these data types over the same physical wires. Figure 5-11 shows

[1]For simplicity, the term *heap addresses* is used in this section to include both heap addresses and addresses in the adjacent static memory section.

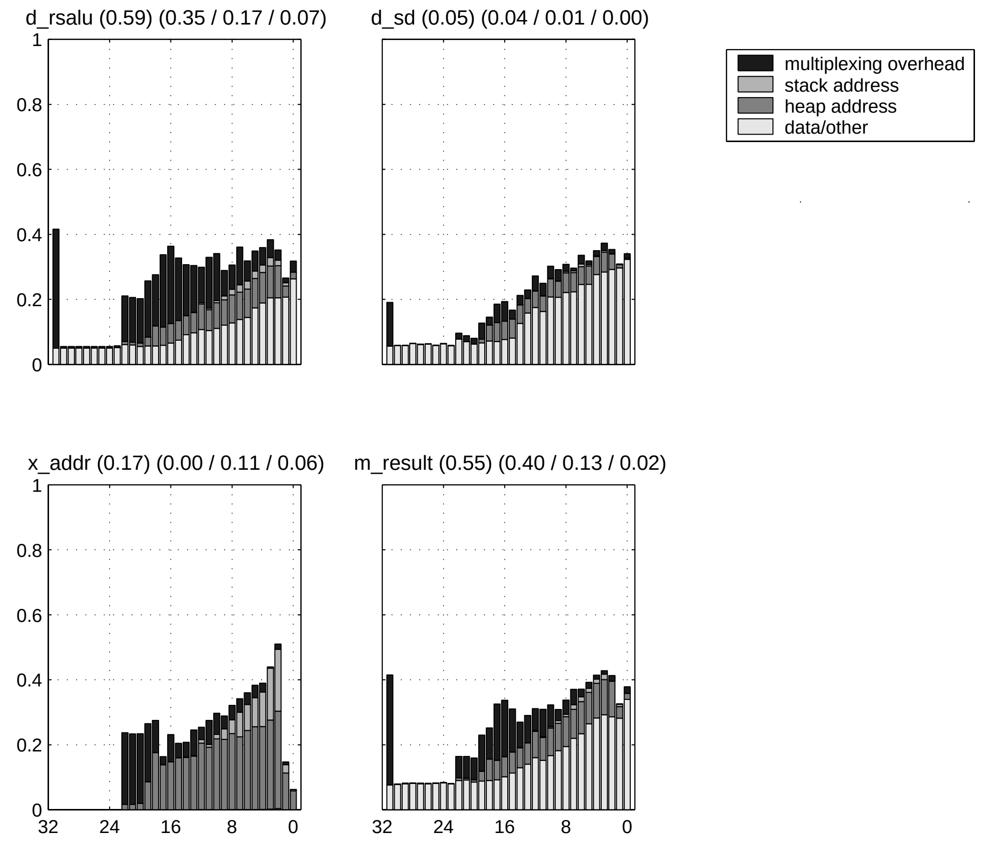


Figure 5-10: Bit transitions for selected nets broken down into: intrinsic transitions between data values, intrinsic transitions between stack addresses, intrinsic transitions between heap addresses, and time-multiplexing overhead. Each plot is labeled with the net name, the overall new value rate, and this rate broken down into the frequency for each value type (data/heap/stack).

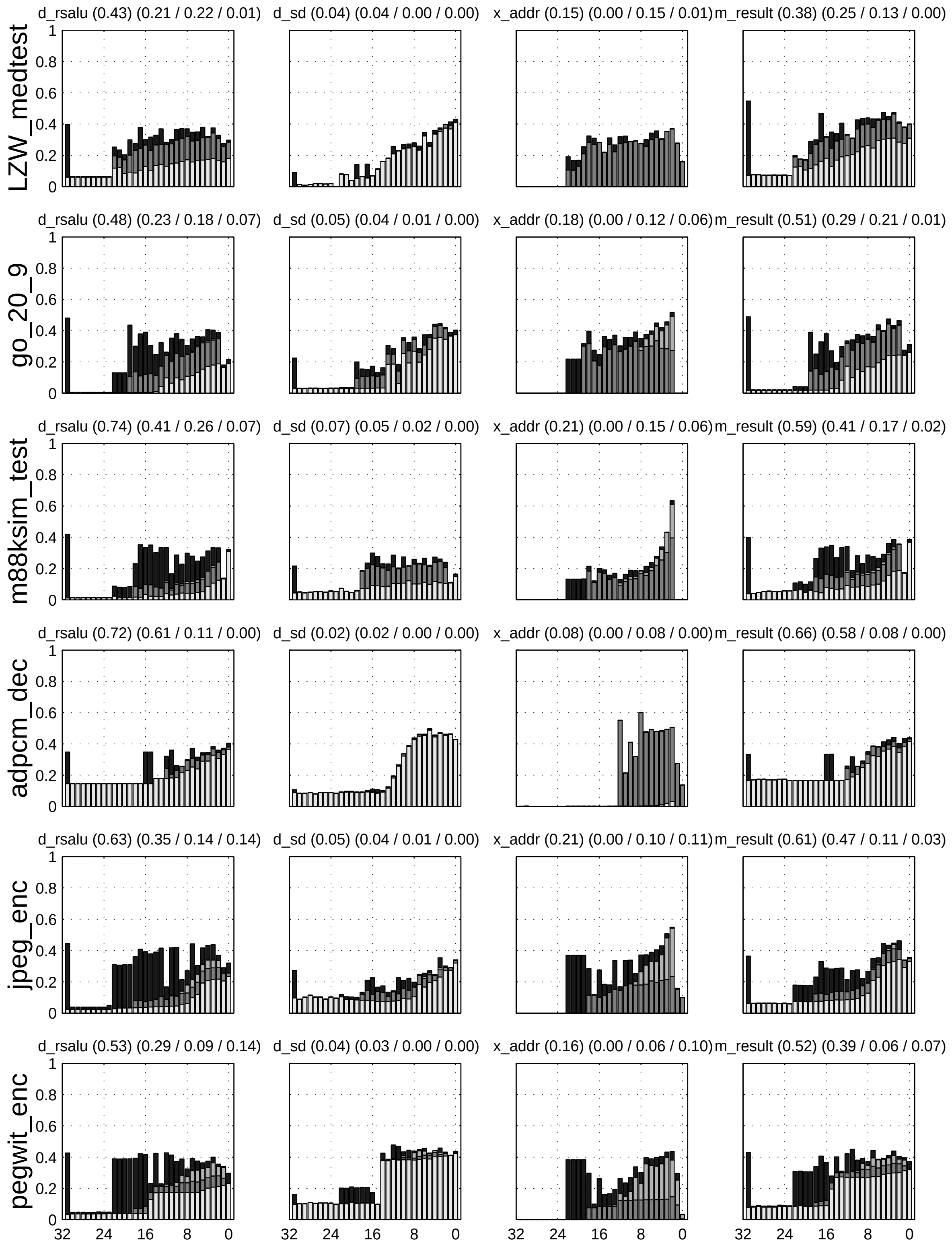


Figure 5-11: Bit transitions for various benchmarks split as in Figure 5-10.

	d_rsalu	d_sd	x_addr	m_result
LZW_medtest	18.5	8.9	9.7	12.9
go_20_9	41.9	14.4	20.4	25.1
m88ksim_test	57.2	14.0	20.1	33.8
adpcm_dec	12.1	1.6	0.5	5.5
jpeg_enc	57.0	16.1	39.6	28.9
pegwit_enc	44.7	11.6	42.5	27.8
average	38.6	11.1	22.1	22.3

Table 5.3: Percentage of bit transitions attributed to time-multiplexing overhead.

this breakdown for each benchmark, and the time-multiplexing overhead for these nets in each benchmark is summarized in Table 5.3.

As expected, the overhead for switching between the data types is quite large. For the RS ALU input, almost 40% of the total transition activity is due to switching between data values, heap addresses, and stack addresses; as is around 22% for the address bus. In some benchmarks these numbers are significantly higher; for example, in jpeg_enc the overhead is 57% for the RS ALU input and 40% for the address bus. Although only a few nets are shown, this activity propagates through the entire execution datapath, including the ALU and write-back path. Additionally, the activity is very sensitive to the memory allocation policies; for example, the activity would increase significantly if the stack base address was set to a value with more one bits.

An interesting characteristic is that most of the activity in the low-order bits tends to be intrinsic to the different data types, while the time-multiplexing overhead causes more activity in higher-order bits (up until bit 22). It is clear that the high-order bits will almost always have transitions when switching between the different data types due to their characteristic bit patterns. The low-order bits, in contrast, are expected to be more random and thus have about half the time-multiplexing overhead activity as the high-order bits; but, in many cases there is significantly more correlation between the low-order bits across the different data types. One possible explanation is that offsets are computed as data values but then combined with high-order bits to become address values; in this case the low-order bits will correlate between the different types.

Spatial separation allows for specialization in addition to improved value correlation. For example, if the address and data pipelines are partitioned, they can use separate smaller register files. In the case of the frequently used stack-pointer, there is no need to read it out of a register file at all; it can reside in a special-purpose register in the datapath to be accessible with very low energy. The address pipeline could also be specialized to support auto-incrementing and auto-decrementing loads and stores since memory accesses are often sequential. Once the memory access pipeline reaches this level of specialization, it becomes similar to the program counter pipeline. As in Section 4.4, the upper portions of the pipeline can be gated during sequential operation; and, as in Section 4.5 cache tag-checks can be eliminated.

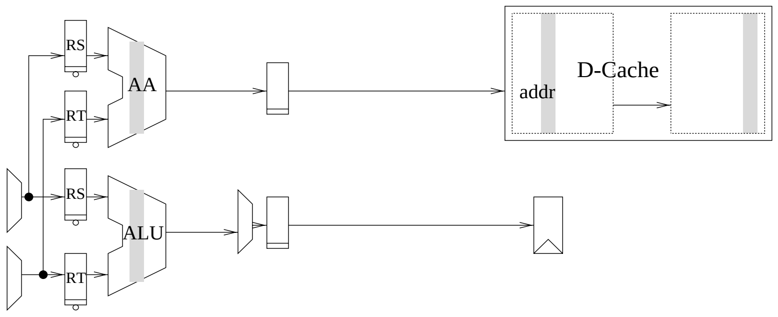


Figure 5-12: Split ALU and AA.

PC	IF	Dec/ Reg	AA	Mem ALU	WB

Figure 5-13: Advanced ALU pipeline stages.

5.4 Sidecar Address Pipeline

The previous section demonstrated the potential benefits of partitioning a unified pipeline into address and data segments. Implementing such an architecture will require a compiler to recognize the partition between address and data registers. In this section I experiment with an alternative pipeline organization that attempts to realize some of the benefits of spatial separation without requiring any ISA changes.

One source of address and data value intermixing is that the ALU is used both for data computation and to produce addresses for memory access instructions. The simple modification shown in Figure 5-12 uses an extra address adder (AA) so that memory access instructions do not use the ALU.

Once the ALU/AA separation has been made, a performance-oriented optimization is to advance the ALU into the same pipeline stage as the data cache as diagrammed in Figure 5-13. This is a relatively well known alternative pipeline implemented in industry processors and analyzed in [18]. The advanced ALU allows load values from the data cache to be bypassed to immediately following ALU instructions, thus eliminating the load-use interlock (LUI). However, if a memory access instruction follows an ALU instruction which computes its base address, the pipeline must stall for one cycle to allow this value to be bypassed to the AA; this is an address-generation interlock (AGI). Furthermore, the branch resolution latency increases by one cycle assuming that branches are still handled by the ALU; this doubles the branch mispredict penalty from one cycle to two.

5.4.1 Design

I modified the Vanilla Pekoe design to use a split AA and advanced ALU as diagramed in Figure 5-14; I term this the *sidecar address pipeline*. The sidecar address pipeline allows for some specialization in the AA. Since the RT input is always the sign-extended immediate value, this input latch can be reduced to 16 bits with the sign extension performed after the latch for memory access instructions only. Additionally, in contrast to the ALU, this

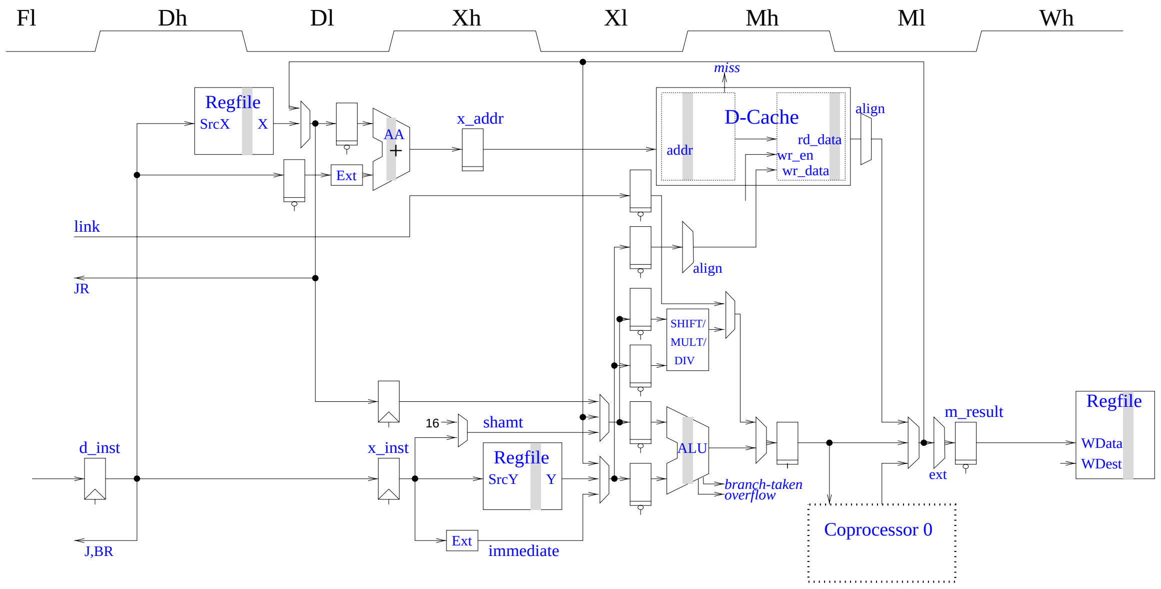


Figure 5-14: Sidecar address pipeline diagram. This design is modified from the Vanilla Pekoe pipeline in Figure 3-2; for clarity I retain the same pipeline stage abbreviations (P/F/D/X/M/W).

adder does not require any control signals. A drawback of the advanced ALU is that the register operands must be available for the AA in the X stage and the ALU in the M stage. The obvious solution is to read both values during the decode stage as before and use two 32-bit flip-flops to retain the values. However, the AA only requires the RS register, so my solution is for all instructions to read RS from the register file during stage D and RT during stage X. This eliminates the D stage flip-flop and bypassing logic for RT. An alternative could be for memory instructions to read RS during stage D and ALU instructions to read both RS and RT during stage X. This could be implemented with a three read-port register file, or with a two read-port register file that interlocked to resolve the potential structural hazards.

In comparison to the Vanilla Pekoe pipeline in Figure 3-2, the sidecar pipeline requires the extra 32-bit flip-flop to retain RS, and some extra flip-flops to retain the necessary instruction fields for the ALU. However, it reduces the RT latch for the AA to 16 bits, and eliminates a flip-flop on the write-back path and in the store-data path. Additionally the bypass network and control is simplified since there is only one bypassing source. An extra bypassing mux is required for RS in the D stage, but the store-data bypassing mux can be eliminated since the RT path is no longer used to input the immediate address offset to the ALU. In total, the number of bypass comparisons is reduced from 4 to 3, and the number of bypass mux inputs is reduced from 11 to 8. To handle jump-and-link instructions, an extra latch is required to hold the link value from X to M.

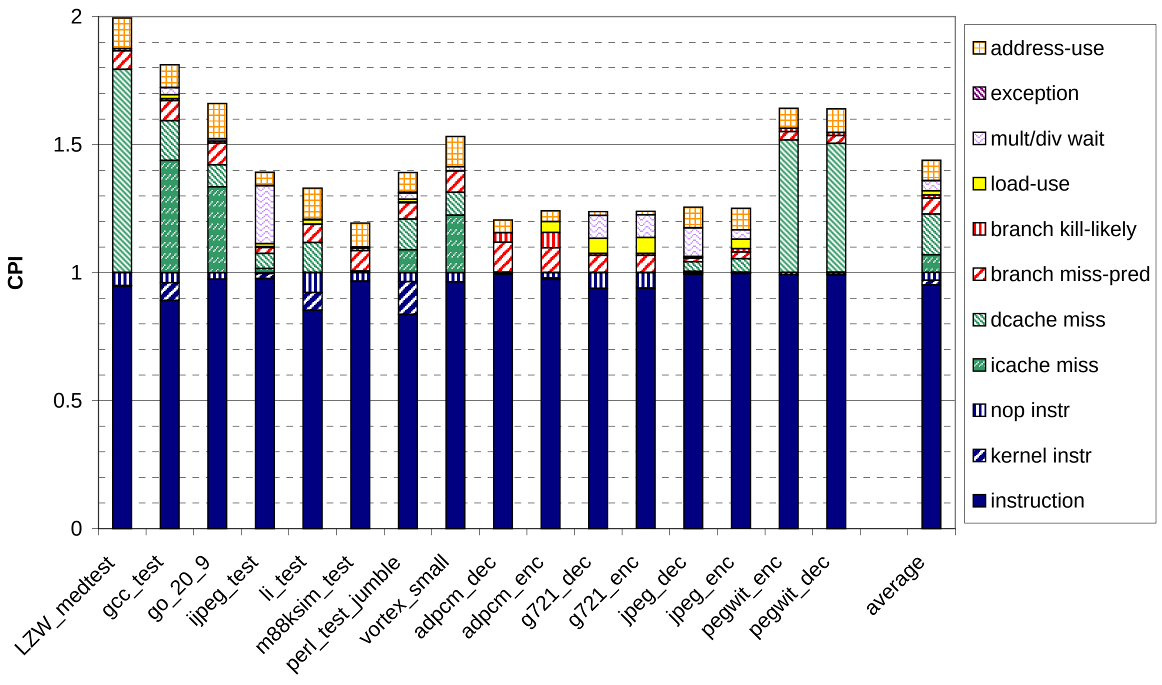


Figure 5-15: Sidecar pipeline CPI

5.4.2 Performance

Figure 5-15 shows the CPI for the various benchmarks running on the sidecar processor. Compared to the Vanilla Pekoe performance in Figure 3-7, most of the load-use stall cycles have been eliminated; a few remain because signed byte and half-word loads can not be bypassed in either design. However, the additional address-use interlock introduces about as many new stall cycles as load-use stalls eliminated. Additionally, doubling the branch misprediction penalty has a significant impact. Overall, the performance degradation is around 3%. Although disappointing, this negative result is consistent with [18] which found that the alternative pipeline was only beneficial with longer cache latencies and better branch prediction.

In this experiment, I ran the benchmark binaries unmodified on the sidecar processor. However, some of the AGI stall cycles could potentially be eliminated if the compiler (scheduler) were more aware of the machine configuration. [18] found that the scheduler could provide little additional performance improvement. Nevertheless, preliminary studies on the sidecar simulator with a modified scheduler have shown some performance gains which seem able to recover the 3% CPI degradation.

One particular problem is the use of load-upper-immediate (lui) instructions to generate a base-address for 32-bit immediate addressing. On the sidecar pipeline, the lui instructions go through the ALU (shifter) and thus cause an AGI if used by a subsequent memory access instruction. Ultimately, this interlock should be unnecessary since the immediate value is available in the pipeline; an ISA change could easily send the immediate value to the AA and eliminate the stall.

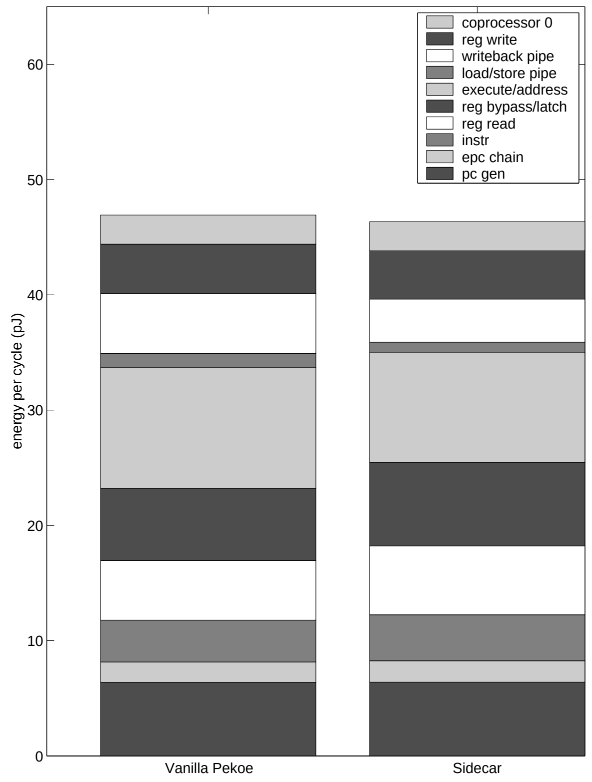


Figure 5-16: Sidecar pipeline energy consumption compared to Vanilla Pekoe. The bars show the datapath energy per cycle excluding inter-component nets.

5.4.3 Analysis

Figure 5-16 shows the datapath energy consumption of the sidecar pipeline in comparison to Vanilla Pekoe. The data represents the harmonic mean for all the benchmarks. The component energy models are the same for each design, but since extracted capacitance values are not available for the sidecar pipeline I exclude inter-component nets from the energy breakdowns. This should not significantly affect the comparison; Figure 5-3 shows that the expected contribution of inter-component nets is around 6%, and this energy should track that of the components themselves.

A very slight savings in energy per cycle is obtained using the sidecar pipeline; but, since more cycles are executed the overall energy is greater. The combined ALU and AA energy in the sidecar design is somewhat less than the ALU energy in Vanilla Pekoe; as is the write-back path energy, mainly due to the elimination of a 32-bit flip-flop. However, as expected some portions of the energy consumption increase; for example, due to the D stage 32-bit flip-flop for RS which is part of the “reg bypass/latch” energy.

Instead of further analyzing the positive and negative energy deltas for the sidecar pipeline, it is more enlightening to examine the bit correlation at the inputs to the ALU and AA. Based on the pipeline configuration in Figure 5-12, Figure 5-17 shows the bit correlation for the ALU (and AA) inputs in the original unified design as well as the split ALU/AA design. It can be seen that bit 31 and most other bits have a small reduction in switching activity, but still many address values go through the ALU. This is because in addition to

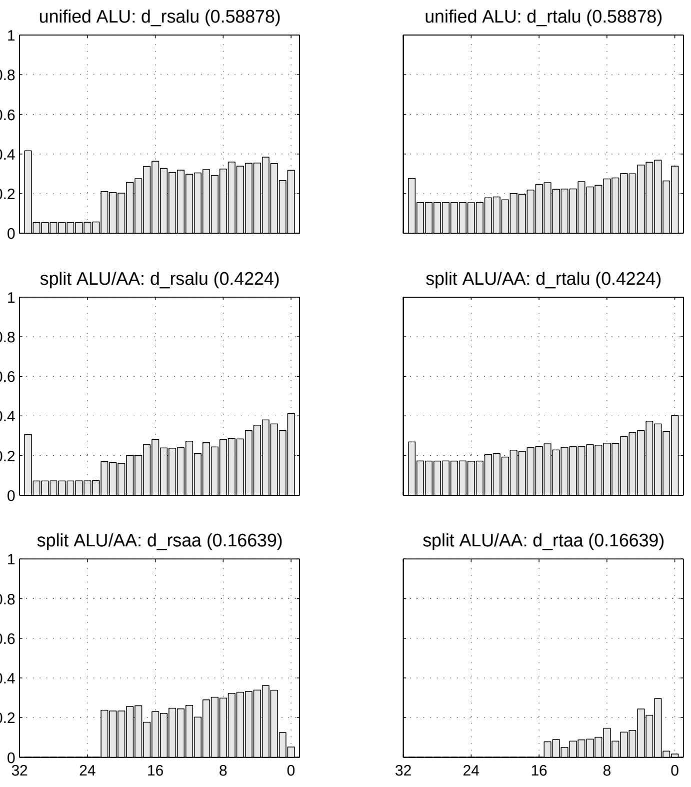


Figure 5-17: Bit correlations using a unified ALU or a split ALU and AA.

memory accesses, programs perform many computations on addresses (incrementing and decrementing for example). For this reason, the sidecar pipeline does not achieve its goal of increasing bit correlation through spatial separation. Separating address and data values requires a more dramatic change than simply targeting memory access instructions, and it will only be feasible through changes to the ISA.

5.5 Summary

In this chapter I have presented detailed energy breakdowns for the optimized processor datapath and cache based on 13 billion cycles of simulated program execution. The energy breakdowns reveal some interesting characteristics. RISC instruction sets are designed so that instructions have little variance in complexity, and likewise different instructions tend to have similar energy costs. Loads and stores are the exception to this rule since they consume much more energy in accessing the data cache. Potentially, instruction sets which expose the sources of energy consumption to software could eliminate some of the energy overheads hidden by typical RISC instruction sets [22]. Furthermore, a clear bottleneck is the overhead of maintaining a program counter and fetching instructions every cycle; even with optimizations and not including the control logic energy, the fixed cost for doing this is estimated to be around 60% of the total energy for most instructions. Thus, a majority of the energy for executing an instruction is spent even before values are read from the register file; this is a major limitation for optimization techniques which target the actual work performed by instructions to lower their energy consumption.

The total datapath and cache energy consumption is optimized to the extent that it is equivalent to clocking around 32 32-bit flip-flops every cycle. In such a design, every flip-flop, latch, mux, and wire on a frequently used path is an important part of the total energy. Additionally, static leakage currents become more important as the dynamic energy is minimized. And although it is not modeled in this thesis, the control logic is expected to become increasingly important as the datapath and cache energy is minimized.

I also presented detailed bit transition statistics in this chapter for many nets in the design and for various benchmarks. These reveal some interesting characteristics which demonstrate the impact of intermixing different types of address and data values. Significant switching activity is caused by time-multiplexing stack addresses, heap addresses, and data values over the same datapath. Spatial separation is a technique which can eliminate this overhead. I experimented with an alternative pipeline organization that uses a separate address adder so that load and store instructions do not use the ALU. However, truly separating these data types can only be achieved by re-designing the ISA.

Chapter 6

Summary and Conclusion

In this thesis, I have presented a complete progression starting from the development of a circuit simulation framework and resulting in detailed microprocessor energy breakdowns and analysis. The first challenge was the lack of tools for evaluating the energy impact of microarchitectural design choices and optimizations. Thus, I began by implementing SyCHOSys, a fast, accurate, and flexible simulation framework capable of modeling microprocessor energy usage for full benchmark executions. Additionally, energy models were developed for the major processor circuit structures including the caches and datapath components. Next, I described the microarchitectural design of the Vanilla Pekoe microprocessor which I implemented using SyCHOSys. The baseline design used energy-efficient circuit structures, but I found many opportunities for optimization in the orchestration and control of these components. I implemented and analyzed various individual optimizations which combined to target the entire datapath and cache energy consumption. After arriving at an optimized microprocessor design, I provided a detailed characterization of its energy usage. This included various energy breakdowns and extensive analysis of the signal transition activities in the design.

SyCHOSys simulates a circuit by compiling a structural netlist into scheduled component evaluations. I found that representing synchronous designs with structural netlists simplifies cycle scheduling and enables aggressive compiler optimizations. In addition to providing fast simulation speed, the SyCHOSys framework allows additional functionality to be automatically compiled in. I have added signal transition counting on external nets connecting blocks to augment block-internal statistics gathering code for energy estimation. Fast and accurate techniques for transition-sensitive datapath energy modeling have been developed that were found to be within 7% of SPICE energy estimates. Using SyCHOSys, I implemented a detailed pipelined MIPS processor model which runs at 45 kHz while tracking statistics sufficient to model almost all datapath and memory energy consumption.

In performance-oriented microarchitectures, all components are active every cycle. However, energy-conscious designs can eliminate much of the unnecessary activity without impacting performance. General clock gating based on operand types and the liveness of instructions in the pipeline reduces clock activity for flip-flops, latches, and dynamic adders, and also data activity in downstream functional units. More specialized clock gating can be applied to exploit predictably low data activity; for example, the high-order bits of the program-counter components can be gated most of the time. Compared to a design with minimal clock gating, these optimizations reduce the number of flip-flops and latches clocked per cycle by almost 70%. Ultimately, the average clock activity is 13% less than that required to move a typical arithmetic operation from instruction fetch through write-back (with the program-counter not included). Data activity is even lower, with only 43 flip-flop and latch output bits toggling per cycle on average. Additionally, 70% of all register file reads can be eliminated by avoiding those that are unnecessary. Likewise, instruction cache tag checks can be eliminated when the same line is accessed multiple times; this reduces the tag check rate by 70%. The rule-of-thumb seems to be that 70% of the activity for most of the components in the microprocessor can be eliminated using relatively simple optimizations.

Available timing slack can often be exploited to reduce energy, for example by minimizing the sizes of transistors off critical paths. For non-critical flip-flops and latches transistor-sizing provides little benefit, but more significant savings can be achieved by selecting different structures based on the local clock and data activity. Additionally, instead of optimizing individual components for energy-delay product, an energy-efficient design should minimize the energy for components off the critical paths and make timing-critical components faster even at the expense of higher energy usage.

Ultimately, the various microarchitectural optimizations combine to reduce the total energy for the datapath and caches by a factor of two. It is important to point out that the control logic energy is not accounted for, and this energy is expected to be increasingly significant as the rest of the microprocessor is optimized. Additionally, the low energy indicates that the chip utilization is very low; the total energy for the datapath and caches is equivalent to less than 32 32-bit flip-flops whereas the chip could accommodate over 3,500 of these devices. This hints at the degree to which static leakage currents will be important, and serves as a motivation for more specialized designs which can achieve higher utilization of the available resources.

RISC instruction sets have been designed to achieve one goal: performance. Since the execution of instructions can be pipelined, most of the microarchitectural work they perform is hidden; only data access and computation is on the critical path for performance. However, only around 12% of the energy for computational instructions is spent in the ALU. This provides motivation for energy-exposed instruction sets which allow software to eliminate unnecessary microarchitectural work. Additionally, in analyzing the energy breakdowns, I found that the overhead for maintaining a program-counter and fetching an instruction from the cache accounts for the majority of the energy that it takes to execute an instruction. Many optimization techniques seek to reduce the energy for the actual work performed by instructions, but this result indicates that the impact of these techniques will be limited. More energy-efficient microprocessors will most likely require alternative instruction sets which allow more work to be performed with less overhead.

Typical performance-oriented designs are based on generic resources and use time-multiplexing to utilize these resources to the greatest extent possible. Energy-efficient designs, in contrast, can benefit from more spatial organizations with specialized resources that have lower energy access costs than the general purpose structures. Through detailed analysis of the bit transition activity in the microprocessor, I found that time-multiplexing stack addresses, heap addresses, and data values over the same physical hardware causes significant overhead. Potentially, alternative instruction sets can enable spatial designs which are specialized for different data types.

In conclusion, I have found that relatively simple optimizations can eliminate many unnecessary microarchitectural operations and have a significant impact on microprocessor energy consumption. After these optimizations are implemented, the energy for generating a program-counter and fetching the instruction is more than that spent in the actual instruction execution. Future architectures will need to reduce this overhead as a first step in providing more efficient computation. Furthermore, in performance-oriented RISC architectures all instructions have similar complexity and microarchitectural operations are hidden from software. Energy-exposed architectures can allow software to eliminate sources of energy consumption which would be difficult or impossible to eliminate in hardware. Finally, traditional architectures utilize generic resources by time-multiplexing various operations over the same physical structures. Separating these resources spatially can eliminate the overhead of inter-mixing different data types, and enable more efficient energy usage through specialization. Thus, future microprocessor designs have ample opportunity to provide more energy-efficient computation.

Bibliography

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA-27*, pages 83–94, June 2000.

[2] T. D. Burd. *Energy Efficient Processor System Design*. PhD thesis, University of California at Berkeley, Spring 2001.

[3] T. D. Burd and R. W. Broderson. Energy efficient CMOS microprocessor design. In *Proceedings of the 28th Annual HICSS Conference*, volume I, pages 288–297, January 1995.

[4] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Broderson. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid State Circuits*, 35(11):1571–1580, November 2000.

[5] T. D. Burd and B. Peters. Power analysis of a microprocessor: A study of an implementation of the MIPS R3000. Technical report, ERL Technical Report, University of California, Berkeley, May 1994.

[6] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-1342, University of Wisconsin-Madison, June 1997.

[7] J. A. Butts and G. S. Sohi. A static power model for architects. In *Micro-33*, pages 191–201, December 2000.

[8] CAD Group, Stanford University. *THOR tutorial*, 1988.

[9] R. Canal, A. Gonzalez, and J. E. Smith. Very low power pipelines using significance compression. In *Micro-33*, pages 181–190, December 2000.

[10] A. P. Chandrakasan, S. Cheng, and R. W. Broderson. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[11] Standard Performance Evaluation Corporation. Spec95, 1995. `http://www.spec.org`

[12] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA-28*, June 2001.

[13] A. Dhodapkar, C. Lim, G. Cai, and W. Daasch. TEM2P2EST: A thermal enabled multi-model power/performance estimator. In *Workshop on Power-Aware Computer Systems, ASPLOS-IX*, November 2000.

[14] J. Montanaro *et al.* A 160-MHz, 32b, 0.5-W CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1712, November 1996.

[15] L. T. Clark *et al.* A scalable performance 32b microprocessor. In *Digest of Technical Papers, International Solid-State Circuits Conference*, volume 44, pages 230–231, February 2001.

[16] L. T. Clark *et al.* A scalable performance 32b microprocessor (presentation slides). In *Visuals Supplement to the Digest of Technical Papers, International Solid-State Circuits Conference*, volume 44, pages 186,187,451, February 2001.

[17] S. Ghiasi and D. Grunwald. A comparison of two architectural power models. In *Workshop on Power-Aware Computer Systems, ASPLOS-IX*, November 2000.

[18] M. Golden and T. Mudge. A comparison of two pipeline organizations. In *Micro-27*, pages 153–161, November 1994.

[19] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid State Circuits*, 31(9):1277–1284, September 1996.

[20] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. Technical report, Intel Technology Journal, February 2001.

[21] S. Gupta and F. N. Najm. Power macromodeling for high level power estimation. In *Proceedings DAC*, pages 365–370, Anaheim, CA, June 1997.

[22] M. Hampton. Exposing datapath elements to reduce microprocessor energy consumption. Master's thesis, Massachusetts Institute of Technology, May 2001.

[23] S. Heo. A low-power 32-bit datapath design. Master's thesis, Massachusetts Institute of Technology, August 2000.

[24] S. Heo, R. Krashinsky, and K. Asanović. Activity-sensitive flip-flop and latch selection for reduced energy. In *19th ARVLSI*, March 2001.

[25] `http://www.cag.lcs.mit.edu/scale`

[26] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski. The design and implementation of PowerMill. In *ISLPED*, pages 105–110, April 1995.

[27] IEEE. *Std 1364-1995*.

[28] G. Kane and J. Heinrich. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1992.

[29] R. Krashinsky, S. Heo, M. Zhang, and K. Asanović. SyCHOSys: Compiled energy-performance cycle simulation. In *Workshop on Complexity-Effective Design, ISCA-27*, June 2000.

[30] Peggy Laramie. Instruction level power analysis and low power design methodology of a microprocessor. Master's thesis, University of California at Berkeley, 1998.

[31] C. Lee, M. Potkanjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. In *Micro-30*, North Carolina, December 1997.

[32] A. Ma, M. Zhang, and K. Asanović. Way memoization to reduce fetch energy in instruction caches. In *Workshop on Complexity-Effective Design, ISCA-28*, June 2001.

[33] H. Mehta, R. M. Owens, and M. J. Irwin. Energy characterization based on clustering. In *DAC*, pages 702–707, Las Vegas, NV, June 1996.

[34] T. Mudge. Power: A first class design constraint for future architectures. In *7th Int. Conf. on High Performance Computing*, pages 215–224, Bangalore, India, December 2000.

[35] Mike Muller. Power efficiency & low cost: The ARM6 family. In *Hot Chips IV*, August 1992.

[36] L. Nagel. SPICE2. Technical Report ERL-M520, ERL Technical Memo, University of California, Berkeley, 1975.

[37] Landman P. High-level power estimation. In *ISLPED*, pages 29–35, Monterey, CA, USA, August 1996.

[38] R. Panwar and D. Rennels. Reducing the frequency of tag compares for low power I-cache design. In *ISLPED*, pages 57–62, October 1995.

[39] Silicon Graphics, Inc. *Standard Template Library Programmer's Guide*. `http://www.sgi.com/tech/stl/`

[40] J. E. Smith. A study of branch prediction strategies. In *ISCA-8*, pages 135–148, 1981.

[41] V. Stojanović and V. G. Oklobdžija. Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems. *IEEE Journal of Solid-State Circuits*, 34(4):536–548, April 1999.

[42] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2/3):1–18, August/September 1996.

[43] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In *DAC*, pages 732–737, June 1998.

[44] J. Tseng and K. Asanović. Energy-efficient register access. In *Proc. XIII Symposium on Integrated Circuits and Systems Design*, Manaus, Brazil, September 2000.

[45] N.P. van der Meijs and A.J. van Genderen. SPACE Tutorial. Technical Report ET-NT 92.22, Technical Report, Delft University of Technology, Netherlands, 1992.

[46] L. Villa, M. Zhang, and K. Asanović. Dynamic zero compression for cache energy reduction. In *Micro-33*, December 2000.

[47] E. Witchel and K. Asanović. The span cache: Software controlled tag checks and cache line size. In *Workshop on Complexity-Effective Design, ISCA-28*, June 2001.

[48] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: A cycle-accurate energy estimation tool. In *DAC*, June 2000.

[49] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Kool Chips Workshop, Micro-33*, December 2000.