Combining Instruction Coding and Scheduling to Optimize Energy in System-on-FPGA

Robert G. Dimond, Oskar Mencer and Wayne Luk
{rgd,oskar,wl}@doc.ic.ac.uk
Imperial College Department of Computing
London, SW7 2RH, England

Abstract

In this paper, we investigate a combination of two techniques — instruction coding and instruction re-ordering — for optimizing energy in embedded processor control. We present the first practical, hardware implementation incorporating both approaches as part of a novel flow for automatic power-optimization of an FPGA soft processor. Our infrastructure generates customized processors and associated software, to enable power optimizations to be evaluated on multiple architectures and FPGA platforms. We evaluate using both software estimates of power and actual measurements from both low-cost and high-performance FPGAs. We generate over 150 optimized processor designs for two FPGA platforms, two processor architectures and six different benchmarks at four different clock rates and achieve consistent measured dynamic power reduction of up to 74%, without performance cost. Our results are applicable beyond processor optimization, quantifying the benefits of practical switching reduction and highlighting non-obvious pitfalls and complexities in dynamic power optimization.

1 Introduction

Energy consumption is an increasingly critical concern for designs that incorporate FPGAs. In many cases, a given computation must be executed within an energy constraint, or a level of performance sustained within a practical power dissipation. Power consumption in CMOS devices — including FPGAs — has static and dynamic components. Static power is due mainly to leakage, a feature of the silicon that is outside the influence of the designer, save for reducing the size of the circuit so that it can fit onto a smaller device. Dynamic power is caused by switching, where switching activity on each net of the circuit requires energy to charge/discharge the load capacitances of the FPGA logic and routing. Hence, the designer can have a favorable impact on energy consumption by minimizing the switching frequency of the design.

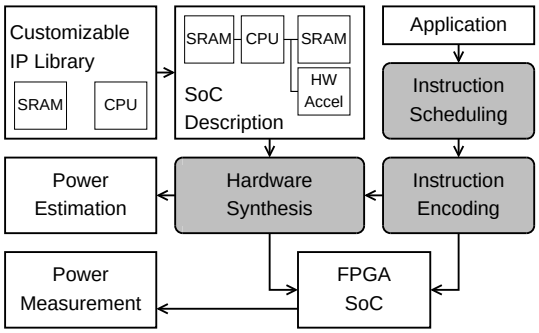


Figure 1. The power-aware compiler schedules instructions to minimize switching. Instruction encoding minimizes switching between adjacent instructions. Hardware synthesis generates the new custom processor.

We present four main contributions in this paper:

1. combined instruction recoding and instruction scheduling, to minimize power in embedded soft-processors.
2. automated generation of systems incorporating power-optimized instruction processors
3. infrastructure for evaluating instruction recoding and power-aware scheduling, independently and combined, using both measurements from the actual hardware and software power estimation.
4. quantitative results showing up to 74% reduction in total FPGA system dynamic power with minimal design overhead and no loss of performance.

Modern FPGAs provide substantial area resource and are used to implement entire systems, incorporating a number of instruction processors, custom accelerators and memory interfaces onto the reconfigurable fabric. The activity of the

circuit nets in these systems, and therefore energy consumption, is dictated by the complex interaction of software and hardware components in the design. FPGA vendors provide tools such as EDK (Xilinx) and SOPC Builder (Altera) for instantiating systems from IP cores, custom design units and software. This type of design flow — where system hardware is synthesized at the same time as the software is compiled — provides a unique opportunity to customize both hardware and software simultaneously. We propose to exploit this opportunity to optimize the energy consumption of an entire system.

In this paper we introduce and investigate a flow for generating both hardware and software for an FPGA based system-on-chip incorporating a soft processor. Our framework (Fig. 1) automatically optimizes aspects of hardware and software that are conventionally assigned arbitrarily for minimum energy consumption with no loss of performance. We optimize power by combining two transformations that minimize bit-switching in the processor control path and instruction memory bus: instruction recoding and instruction re-ordering (scheduling). Both transformations seek to minimize the 'hamming distance' — or number of bit-switches — between successive instructions in machine code.

The following section provides a review of related work while the remaining sections describe each contribution in turn. Section 3 details the power optimization aspect of our framework. Section 4 describes the implementation, specifically our method for processor description and synthesis. Section 5 presents the infrastructure used to evaluate power optimizations. Section 6 provides a quantitative evaluation of our approach for two different FPGA platforms and three different benchmarks. Finally, in Section 7 we conclude and give possible directions for future work.

2 Related Work

Tiwari et al. [12] present some of the earliest work on relating power to software instructions (on the Intel 486 in this case). In particular, Tiwari differentiates the power consumed by each instruction independently and 'circuit state overhead' arising from interactions between groups of instructions. Our focus is on the minimization of this circuit state overhead. A similar analysis is presented by Ou et al. [7] for the MicroBlaze soft-processor, implemented on the reconfigurable fabric of a Virtex-II Pro FPGA. Unlike Tiwari, the MicroBlaze study does not consider circuit state overhead and relies upon FPGA vendor tools to perform power estimation rather than actual measurements. Simunic et al. [8] extend Tiwari's work and consider power modeling and minimization in complete embedded systems, in particular battery powered designs.

A number of authors propose reducing energy in processor buses and decode logic by re-encoding opcodes to minimize hamming distance between adjacent instructions. Benini et al. [2] apply a min-cut algorithm — originally designed for state machine power optimization — to processor instruction encoding. Woo et al. [13] perform a similar optimization that extends to register fields and unused fields in instructions. Kim et al. [5] present additional results and introduce a simulated annealing solution to the opcode assignment problem. In all cases, the authors evaluate their work in terms of bit-switch reduction and do not quantify the impact on actual system power. Re-encoding complicates the decode logic of the processor, possibly offsetting any savings in bit-switching, so actual measurements are important. Our work enables further and more detailed evaluation by performing both power measurement from real hardware and software power estimation.

Su et al. [11] propose power aware 'cold' scheduling that re-orders instructions for minimal bit-switching. Cold scheduling reduces bit-switching by up-to 35% for their benchmark set but at the expense of some performance since the conventional scheduling — that orders instructions to minimize pipeline stalls — is disrupted. The authors do not perform measurements of actual power although Erdogan et al. [3] consider switched capacitence and Lee et al. [6] achieve measured power reduction for a DSP processor with a particularly high circuit state overhead. We differentiate our work by combining re-ordering with instruction re-coding and again by performing actual measurement. In addition, our scheduler guarantees that execution time does not increase as a result of power-aware scheduling.

Wilton et al. [9] quantify the effect of pipelining on energy per-operation in FPGAs and demonstrate considerable dynamic power reduction. We also investigate pipelining the decode unit of our instruction processor to show the combined impact and to generalize our results so that they are not specific to a particular processor instance. The results in Wilton et al. [9] indicate that power estimation tools fail to reproduce many trends that are observable in actual measurements, hence we evaluate our framework using actual measured results.

Our synthesis approach for processors has much in common with the SPREE [14] soft-processor design space exploration tool. The most important distinction is in how we abstract the pipelining from the connectivity of the processor. In SPREE, the control is automatically generated but the pipeline registers are manually located in the microarchitecture. In our framework, the control and pipelining are concisely specified independently from the non-pipelined connectivity of the functional units. We also specify complete systems rather than just the soft-processor. Our work builds upon a large volume of research on architecture description languages, in particular the MIMOLA system [1], that also describes instruction processors structurally. The

PD-XML language [10] similarly differentiates between the description of low and high level microarchitecture, although the distinction is different, where high level in PD-XML is the functional units and instruction set, as opposed to the pipeline structure in our work.

3 Power Optimization Framework

Our framework extends and automates two optimizations: (1) opcode recoding and (2) power-aware instruction scheduling. Both techniques aim to reduce the hamming distance — the number of bit-switches — between successive instructions fetched by the processor. (1) Recoding assigns opcodes to instructions so that frequently adjacent instructions have most bits in common. Implementing this technique requires simultaneous customization of the processor and software so that both are configured for the new encoding. (2) Power-aware instruction scheduling re-orders instructions, subject to dependence constraints, so that instructions with similar encodings are adjacent. Unlike optimal opcode assignment, low-power or 'cold' scheduling is a pure software approach where only the ordering of software instructions is modified. In particular we explore both optimizations together, rather than separately as in previous work [5, 11].

3.1 Instruction Recoding

Instruction recoding exploits the observation that the frequency that each pair of instructions are adjacent in code execution is highly skewed. The principle is to assign opcodes to instructions so that instructions adjacent with high frequency differ by only a few bits, while instructions that are rarely adjacent differ by many bits. Provided that the frequencies can be correctly estimated, we reduce overall bit-switching. Recoding offers the possibility of saving power simply by clever assignment of opcodes that are often assigned arbitrarily. For FPGA soft-processors, it is practical to perform recoding for each application and we perform application specific instruction coding.

$$FM_{xy} = \sum Follows(x, y) \tag{1}$$

We build a matrix FM of frequencies that each instruction is adjacent to every other instruction, then perform a simulated annealing optimization, similar to that proposed in [5], to assign an opcode to each instruction. The frequency matrix is of size $2^n \times 2^n$ where n is the bitwidth of the opcode. We populate each entry xy as per Eq. 1 with the frequency that opcode x is adjacent to opcode y in the execution profile. The dynamic profile is collected from an RTL simulation where a duplicate instruction memory contains the number of times that each location is accessed. We approximate each matrix entry from these instruction frequencies by visiting each instruction in turn and adding its frequency to the entry for that instruction opcode and the preceding opcode. The approximation introduces some small errors at the boundary of loops but considerably reduces the size of profile data required from n^2 to n. The profiling could equally take place in hardware using an actual duplicate instruction memory to hold the profile data.

$$Cost = \sum_{xy \in FM} HammDist(x, y) \times FM_{xy} \tag{2}$$

The optimization step swaps the encoding of each opcode to minimize the total switching cost. We calculate the total cost from the instruction frequency matrix FM using Eq. 2, where x and y are two opcodes. We stochastically swap the assignment of each opcode, using simulated annealing to prevent getting stuck in local minima. In simulated annealing, we always accept swaps that reduce the total cost but occasionally, with decreasing probability, accept swaps that increase the cost. The incremental change to total cost can be trivially calculated for each swap by subtracting the contribution of the opcodes swapped in the initial state and adding the contribution in the final state. We use the processor default instruction set as the initial state and terminate after a pre-determined number of successful swaps which we set by experimentation. In practice, the algorithm rapidly converges on an optimal solution.

We provide the *opcodeopt* tool for automating opcode assignment given an application (compiled from C) and an execution profile generated automatically from our synthesis framework (Section 4). The *opcodeopt* tool is integrated into our flow so that the optimization is completely transparent to the designer. When the system is synthesized and the application software compiled, *opcodeopt* customizes the processor decode unit and re-encodes the assembled software. Our experimental processors have MIPS instruction sets, hence *opcodeopt* performs two independent optimizations, one for each six bit opcode field used in the 32-bit MIPS instruction set, the opcode and the function code. Using two $2^6 \times 2^6$ matrices, each *opcodeopt* run takes around 2–3 seconds to generate an optimal encoding.

3.2 Low-power instruction scheduling

Our framework includes an optimizing compiler, built using the CoSy [4] framework, that we extend to support an enhanced version of power-aware or 'cold' scheduling [11]. The implementation introduces two significant enhancements over the previous work. Firstly, we perform power-aware scheduling simultaneously with performance aware scheduling to ensure that the execution time is not increased. Secondly, since the first enhancement significantly

limits the freedom for the scheduler to re-order instructions, we introduce a two-phase approach to make additional flexibility available to the power aware scheduler.

We extend a conventional heuristic list scheduler to support power aware scheduling. A conventional scheduler generates a graph of the data dependences between instructions, then explores the space of schedules from the partial order imposed by the dependences. Heuristic schedulers evaluate the cost of a schedule by summing a set of heuristics, each weighted to reflect their relative importance.

We add an additional power-aware term to the heuristic of a conventional list scheduler to benefit schedules with a low total hamming distance. To guarantee that the performance schedule is not disrupted, we assign the lowest weight to the power-aware term. Essentially, we use the power-aware heuristic as a 'tie-breaker' to choose the lowest power from possibly many schedules that are equal in terms of performance. We find that, in practice, using a lower priority than the performance driven terms causes the power-aware term to influence the schedule only very rarely. In fact, for the SHA-256 benchmark that has unusually large basic blocks that allow the scheduler considerable freedom, the scheduler was not influenced at all by the power-aware term. We identify a classical phasing problem: power-aware scheduling requires the bit-encoding of each instruction, available after register allocation. However, after register allocation, there is insufficient flexibility to perform power-aware scheduling without disrupting the performance motivated schedule.

We propose to perform two independent scheduling passes that both optimize performance and are power-aware at the same time. The first pass occurs prior to register allocation, which allows for more flexibility, since temporal reuse of registers constrains the schedule. Before register allocation, the registers in the instruction encoding are of course undefined and so we do not consider bit-switching of register fields. Instead, the scheduler optimizes for all other parts of the instruction word. The compiler performs a second scheduling stage last, after register allocation. The second stage has reduced freedom but can make fine adjustments that consider the entire instruction word.

Our implementation introduces one further enhancement to improve the flexibility available to the power-aware scheduling. We add a loop-unrolling stage prior to code generation to increase the size of loop bodies and thus the range over which loop instructions can be moved. This step has an added advantage that the unrolled loop bodies contain repeated copies of the same instructions. Thus, instructions with identical opcodes can be packed together, subject to dependence constraints. We retain the compiler default heuristics for loop unrolling that compromise between performance and code size. To ensure that loop unrolling does not influence the results, we unroll loops equally irrespective of whether power-aware scheduling is also performed.

We anticipate that recoding and scheduling are complementary for three main reasons. Firstly, opcode encoding only provides improvement in sections of code where the instruction adjacency frequencies are similar to those in the profile of the entire program. In contrast, power-aware scheduling is performed independently for each code segment. Secondly, the techniques influence different portions of the instruction word: scheduling considers the entire word whereas opcode assignment is only applicable to certain fields. Thirdly, the techniques influence different aspects of the activity: scheduling is able to place identical instructions together for zero-bit switching, while encoding minimizes the bit changes when they must occur between non-identical instructions.

4 Implementation

We generate FPGA Systems-on-chip from an abstract specification using our new synthesis framework. Our synthesis framework includes all the functionality of the Xilinx EDK tool with two major enhancements. (1) We support a significant amount of customization of the soft-processor including flexible pipeline organization and customizable instruction encoding. (2) Our framework has flexible scripting capabilities allowing many hundreds of bitstreams to be generated. Thus, we explore a wide design space in an automated fashion.

We describe instances of systems-on-chip concisely via an object-oriented Python scripting interface. Our description interface provides a meta-language for connecting and instantiating hardware blocks which are directly translated into structural Verilog. The system includes a novel abstraction of microarchitecture and connectivity for embedded processors, allowing various organizations of the functional units to be tested by modifying only a few lines of code.

We support specification at four levels of abstraction:

(1) System level: structural connection of auto-generated or pre-designed modules at the bus level.
(2) Microarchitecture level: pipeline organization of instruction processor components, where functional units are allocated to pipeline stages and pipeline control is specified.
(3) Connectivity level: connectivity of instruction processor components with no pipeline information, essentially an unpipelined implementation of a processor.
(4) Register Transfer Level: conventional hardware description language to specify complete IP cores — such as I/O or memory interfaces — or small state machines that can be combined into larger modules, for example a fetch unit for an instruction processor.

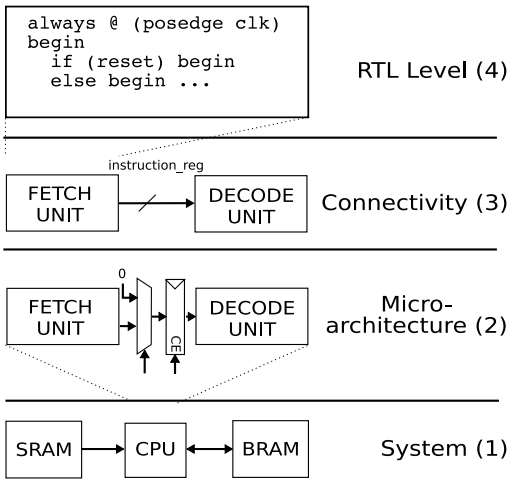


Figure 2. The four levels of abstraction we use to specify systems for synthesis. The connectivity and microarchitecture levels are exclusive to soft-processor components.

Our synthesis tool combines all four levels to generate a set of top-level, structural HDL designs. We connect high-level blocks using only the first and fourth levels, in common with contemporary design flows such as Xilinx EDK where cores are implemented in RTL (4) and connected using a system level netlist (1). The cores are hand optimized, while the top level interconnections are generated automatically. We build instruction processors using all four levels of abstraction. The motivation for the intermediate levels (2 and 3) is to abstract the connectivity of functional units from the pipeline architecture. We generate multiple pipeline organizations of the same processor by modifying only the microarchitecture description.

The system (1) and connectivity (3) levels are essentially structural descriptions for connecting blocks. There are two basic units at these levels: blocks, which are equivalent to modules in Verilog or entities in VHDL and nets, used to interconnect blocks. Blocks are constructed hierarchically so that a single block can be constructed from multiple internal blocks. At the lowest level of the hierarchy are *opaque* blocks that are implemented and hand optimized at the register transfer level (4). Blocks have ports, either input or output with specified bitwidth. Nets can be connected to any number of block ports and are declared as global or local. Two special rules apply to nets that deviate from conventional structural language semantics but allow for concise description of our designs: Firstly, bitwidth is not declared but inferred from the connected ports, an error results if ports of different size are connected. Secondly, if multiple outputs are connected we generate an OR function to provide a shorthand for wired-OR buses common in FPGA SoC designs. The nets at the connectivity level are able to span multiple pipeline stages for processor construction, where pipeline registers and control are added automatically. Block output ports at the connectivity level have a latency parameter to facilitate the automated insertion of pipeline registers into these nets. A non-zero latency n indicates a pipelined or multi-cycle output where a result is available after n cycles. Our tool uses latency information to construct the pipeline so that the correct number of pipeline registers are added for each signal.

The below excerpt describes a section of processor datapath at the connectivity level. The example instantiates two functional units, a fetch unit and a decode unit, then connects a port on each to a common net, `instruction_reg`. The `instruction_reg` net translates to a signal or a register in RTL depending on the nature of the microarchitecture description that the connectivity description is composed with.

Python description for processor datapath segment

```
clock = NetGlobal('clock')
instruction_reg = Net('instruction_reg')
fe0 = fetchunit('fe0')
fe0.connect('clock', clock)
fe0.connect('instruction_reg', instruction_reg)
ilmb.busattach(fe0)
de0 = decodeunit('de0')
de0.connect('instruction_reg', instruction_reg)
```

The microarchitecture level (2) assigns pipeline stages to sub-blocks within a custom processor designed at the connectivity (3) level. The pipelined data paths are automatically generated by adding the appropriate number of pipeline registers to each net. We specify pipeline control manually at the microarchitecture level using two primitives: stalling and squashing. The stall primitive controls a clock enable input for the pipeline registers after a specified stage. Similarly, squashing controls a multiplexer at each stage that replaces an instruction with a no-op. The designer specifies a boolean equation for the stall and squash conditions at each stage.

The code below is the entire microarchitecture description for the four-stage pipelined processor used in our experiments. The first function call `init` instantiates a basic pipeline with functional units from the connectivity description (`de0`, `alu0` etc.). We present the functional units as a list of lists, where each sub-list contains the units in a single pipeline stage. The squash functions describe the pipeline control required, stalling every pipeline stage when the processor is waiting for data from the memory system.

Python 4 stage pipeline description

```
init(self, name, [ [fe0],
[de0, alu0, be0, rf0, ru_a, ru_b, rm_a, rm_b, rmux0],
[ls0, rf1],
[rf2] ] )
self.stall(0, "stall_mem") self.stall(1, "stall_mem")
self.stall(2, "stall_mem")
```

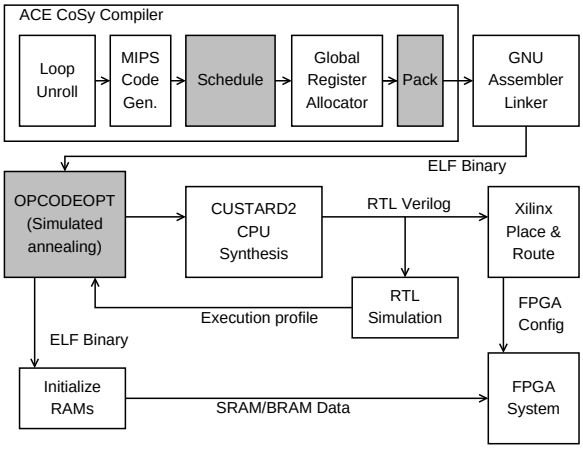


Figure 4. Implementation of our compilation framework. The compiler unrolls loops to increase the number of identical instructions in loop bodies and schedules instructions twice: pre and post register allocation. The opcodeopt tool uses profile information to assign opcodes to instructions.

We make minor modifications to generate the five stage pipeline (below), moving the decode unit (`de0`) and register read unit (`rf0`) to new stage. We add an additional `stall` function for the new stage and also a `squash`. The `squash` serves to nullify an instruction incorrectly fetched in case of a taken branch.

Python 5 stage pipeline description.

```
init(self, name, [ [fe0],
[de0, rf0],
[rm_a, rm_b, ru_a, ru_b, alu0, be0, rmux0],
[ls0, rf1],
[rf2]])
self.stall(0, "stall_mem") self.stall(1, "stall_mem")
self.stall(2, "stall_mem") self.stall(3, "stall_mem")
self.squash(1, "target_en")
```

5 Experimental Framework

Fig. 4 shows an overview of our complete framework with the power optimization stages highlighted. The route from application software to final FPGA configuration incorporates:

1. the compiler, which contains the two power-aware scheduling passes, pre and post register allocation.
2. the standard GNU assembler/linker for the 32-bit MIPS instruction set.
3. *opcodeopt*, our tool for optimizing the instruction set encoding for power.
4. our synthesis framework, for instantiating a system and processor with the required parameters and instruction set encoding.

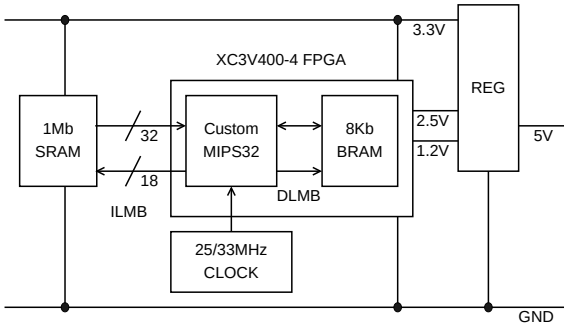
Our compiler is built using the CoSy [4] development system. Not shown in the figure are the standard C front-end and source level optimizations including common subexpression elimination, scalar replacement, strength reduction and dead-code removal. We have highlighted the two power and performance aware scheduling phases that occur before and after global register allocation (GRA).

The *opcodeopt* tool reads the Executable and Linking Format (ELF) binary output from the GNU tools. Using profile information from simulation of the real system, *Opcodeopt* outputs an optimal coding to the processor synthesis tool and substitutes opcodes in the original ELF file to generate code that runs on the customized processor. We also generate an opcode substitution file, used to make incremental changes to the power-optimized software without repeating the *opcodeopt* and synthesis steps.

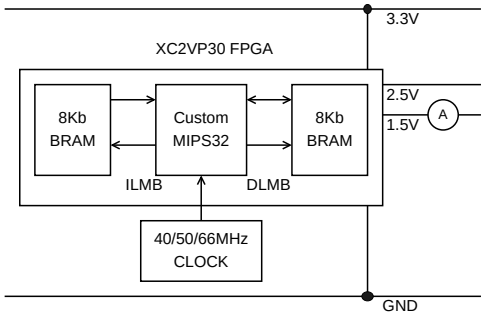
We generate over 150 bitstreams for experiments using a single Python script in our synthesis framework at the system level of abstraction. This script takes a number of parameters that define the system:

- the FPGA board to target (XUP or Spartan3).
- instruction memory in external SRAM/Block RAM.
- instruction encoding generated by *opcodeopt*.
- ELF file, to initialize instruction and data memories.
- clock division factor for the system clock.
- number of pipeline stages (4 or 5).
- optionally, a guide placed and routed design.

We enumerate over ranges of these parameters using a single top-level script, which automates the entire synthesis and optimization process, configures the FPGA and runs a power estimation step. We measure and estimate power consumption from designs placed and routed using Xilinx ISE 7.1. We minimize the impact of the place and route tools on the results using a guide file for each platform, to encourage the tools to use the same placement and routing for common design elements. Figures 3(a) and 3(b) show the two platforms that we use for power estimation and measurement. The first platform uses the Spartan-3 Starter Board, with 90nm XC3S400-4 FPGA and 1Mb of SRAM. The second platform uses the Xilinx University Program (XUP) board, with 130nm XC2VP30-7 FPGA. We measure total system power on the Spartan board, placing an ammeter before the board power regulators. We use jumpers provided on the XUP board to measure the 1.5V VCCINT to isolate the FPGA internal power. Both platforms contain a processor with separate instruction and data buses, labeled ILMB and DLMB respectively. For all results in this paper, we configure external instruction memories on the Spartan3 board and internal BRAM on the XUP board. We measure static power independently by gating the entire clock tree and subtract static power from our measurements.



(a) Spartan 3 board system (XC3S400-4) with on-chip data memory and off-chip SRAM instruction memory. We measure 5V board supply current before regulators.



(b) XUP board system (XC2VP30-7) with on-chip instruction and data memories. We measure VCCINT current directly, after regulation.

Figure 3. FPGA platforms we use for power estimation and measurement

6 Results

We evaluate our framework using six application benchmarks: (1) SHA-256 (secure hashing) (2) Matrix/Matrix Multiplication (3) CRC-32 (checksum) (4) SUSAN (edge detection) (5) Quicksort and (6) ADPCM (audio codec). We present detailed results for benchmarks 1–3 and summary results of final power reduction for 4–6. Fig. 5 shows the reduction in bit-switches in the instruction stream across benchmarks 1–3. Four histograms for each benchmark show results with recoding alone, with scheduling alone, with both and with no optimization. Each histogram shows number of bit-switches against number of instructions, broken down by the two MIPS opcode fields, *op* and *func*. Reduced bit-switching results in more and higher bars towards the left of the graphs.

The opcode assignment optimization alone causes a mean improvement in switching of 45% across the benchmarks. We note that the number of zero bit-switch transitions cannot increase, since all opcodes must differ by at least one bit. The power-aware scheduling reduces switching by a mean of 13%, but causes no improvement in the CRC-32 case. The lack of impact in the CRC-32 case is due to a dependence between each step of the CRC-32 calculation that tightly constrains the ordering of instructions in the code. Unlike recoding, scheduling increases the number of zero bit-switch transitions (by 58% in Matrix Multiply) by scheduling identical instructions together. Composing the two techniques gives the best result in all cases and looks to combine the improvements from the independently applied optimizations. In particular, the number of zero bit-switch transitions is increased by the same amount as with scheduling alone. These results support our hypothesis that the two optimization techniques are orthogonal, although the impact that these abstract bit-switching metrics have on actual system power is still unclear. Hence, the focus of our analysis is the actual power measurement that our framework provides.

Energy saving % Benchmark	Scheduling Only	Recoding Only	Both
SHA-256	59.26	7.41	74.07
Matrix Mult.	34.78	34.78	52.17
CRC-32	-4.55	9.09	4.55
SUSAN	11.48	7.99	13.32
Quicksort	0.85	2.61	8.89
ADPCM codec	5.88	4.07	7.49

Table 1. Energy savings for five stage pipeline, XUP platform at 40MHz

Table 3 shows detailed software estimated and measured results for the XUP platform. We obtain software estimated results from the XPower tool, using signal activities from the post place and route netlist. The percent power saving column shows the reduction in measured dynamic power for each optimization, relative to the measured dynamic power with no optimization. We show the impact of power optimization for benchmarks 1–3, running on 4 stage and 5 stage pipelined processors, each at three different clock rates, 66, 50 and 40MHz. We omit performance measurements from our results since our optimizations do not impact execution speed, hence, the power measurements for the same pipeline depth are equivalent to normalized energy consumption. Table 1 shows a summary of final power savings for all benchmarks running on the XUP Platform with 5 stage pipelined processor at 40MHz. Table 2 shows results of identical experiments for the Spartan3 platform, with the same processors and benchmarks running at 33 and 25MHz. We have condensed the results to show only a mean and maximum (in parenthesis) measured dynamic power saving for each optimization. Recoding complicates

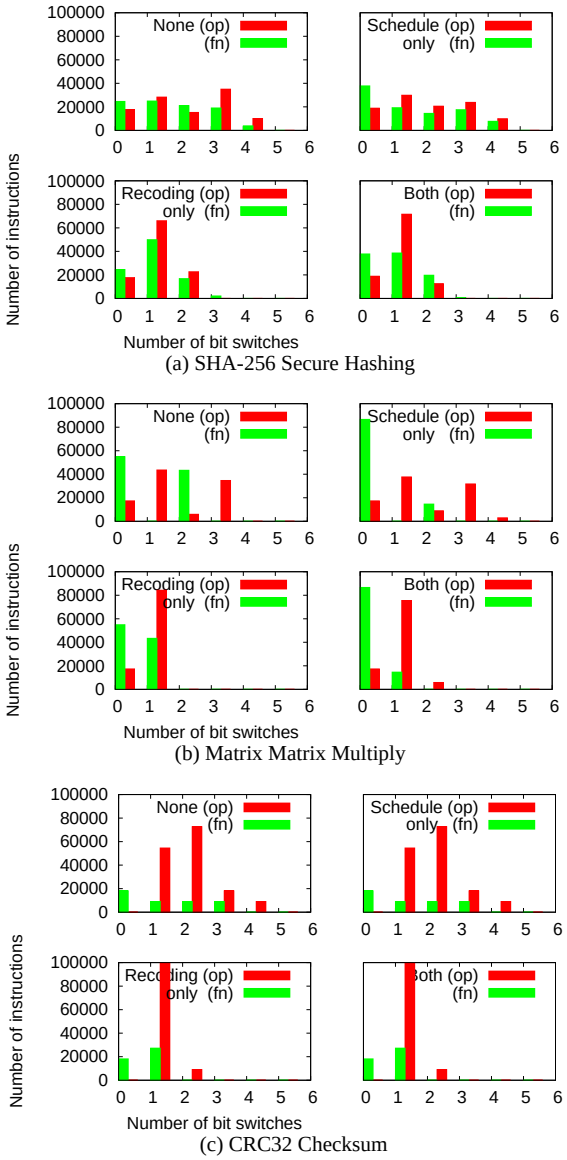


Figure 5. Number of bit-switches against number of dynamic instructions for both optimizations independently and combined.

the decode logic that accounts for around 7% of the processor, Table 6 summarises the increase in decoder area above the default MIPS encoding. We are unable to detect any significant impact on timing from increased decoder complexity. The estimated critical path is increased on average by 1.69% although there is no consistent trend: for 29% of designs recoding decreases the critical path length.

We highlight the following observations:

- the impact of our optimizations varies across the benchmarks, architectures and clock rates, illustrating the value of the automation and flexibility our framework provides. Experiments for a single architecture at a single clock-rate would be misleading in either direction.
- optimization performs best on the five stage pipeline and at 25MHz on the Spartan3 and 50MHz on the Virtex2Pro.
- the percentage power savings are high, as much as 74% (SHA-256, 5 stages at 40MHz) on the Virtex2Pro and 22% (SHA-256, 5 stages at 33MHz) on the Spartan3.
- scheduling reduces power consumption by a mean of 7.11% across all results on the Spartan3 and 13.02% on the VirtexIIPro.
- scheduling reduces power in all cases on the Spartan3 and for all except the CRC benchmark on the VirtexIIPro where a small (<6%) power increase occurs.
- the relative improvement of scheduling compared to recoding is larger than suggested in Fig 5, we anticipate an additional benefit arising from identical instructions executed in sequence.
- recoding reduces power by a mean of 7.11% on the Spartan3 and 14.26% on the VirtexIIPro.
- recoding is effective in all cases, reducing power by up to 37% (CRC-32, 4 stages at 50MHz) on the VirtexIIPro and 21% (Matrix Multiply, 4 stages at 25 MHz) on the Spartan3.
- recoding incurs a negligable area cost, an average of 2.5% above the default processor core. We were unable to detect any significant timing cost, even for the four stage pipeline where the decoder is on the critical path.
- combined optimization yields the highest mean improvement: 9.31% for the Spartan3 and 28.37% for the VirtexIIPro.
- for the XUP platform, combined optimization reduces power in all cases although in 5/18 experiments recoding alone performs slightly better.
- for the Spartan3 platform, combined optimization causes two power increases of 11% and 4% respectively, both for the four stage pipeline.
- our results show greater power for the five stage pipeline in contrast to previous work [9]. Interestingly,

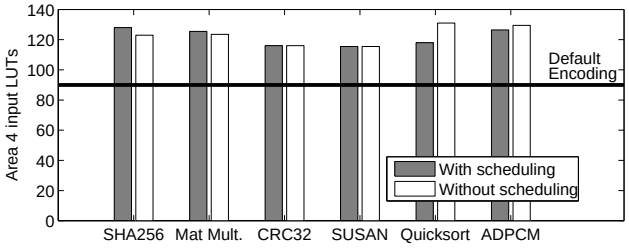


Figure 6. Area increase of decode logic to support recoding

Mean (Max) % power saving	4 Stage		5 Stage	
	33MHz	25MHz	33MHz	25MHz
Scheduling only	3.14 (7.69)	6.15 (12.5)	8.81 (15.63)	10.34 (17.39)
Re-coding only	4.67 (6.9)	9.72 (20.83)	8.21 (13.51)	9.12 (13.64)
Both	0.63 (4.55)	5.93 (20.83)	15.37 (21.88)	14.92 (21.74)

Table 2. Summary of percent energy savings, Spartan-3 platform with external SRAM.

the software power estimation tool predicts the opposite trend.

7 Conclusion

In this paper, we quantify the independent and combined impact of two power optimization techniques: instruction recoding and power-aware scheduling. We present a novel infrastructure for simultaneously optimizing aspects of hardware and software and automating a large number of experiments. We measure significant dynamic power savings, a mean of 28.37% for a VirtexIIPro system and 9.31% for a Spartan3 system . In particular, we find that the two optimization techniques are largely complementary; in some cases, the combined optimization result is larger than the sum of the independent results. We show that the efficacy of the optimizations varies considerably with the architecture of the processor and with the system clock rate. In a general sense, our results show that design techniques for reducing signal activity can have a considerable impact on measured system power, well beyond that predicted by current software estimation tools.

Comparison with prior work is difficult since we are the first to evaluate using actual power measurements and the results are specific to the exact benchmarks. Comparing switching activity, our recoding reduces power by an average of 46%, similar to the 49% achieved in [13]. With scheduling, our average reduction of 14% is lower than the 27% achieved in [11], although our method does not incur a performance overhead and so optimizes energy in addition to power. Our combined average reduction of 52% is higher than for either individual optimization.

In our future work, we intend to leverage our infrastructure to investigate more extensive customization of the processor control and datapath and the impact on power.

Acknowledgement We are grateful to ACE Associated Compiler Experts, Celoxica, the EPSRC (grant numbers C544692 and C509625) and Xilinx who provided tools and/or funding to support our work.

References

[1] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA language - version 4.1, 1994.

[2] Luca Benini, Giovanni De Micheli, Alberto Macii, Enrico Macii, and Massimo Poncino. Reducing power consumption of dedicated processors through instruction set encoding. In *Proc. Great Lakes Symposium on VLSI*, page 8, 1998.

[3] Ahmet Teyfik Erdogan and Tughrul Arslan. On the low-power implementation of FIR filtering structures on single multiplier DSPs. *IEEE Transactions on Circuits and Systems*, 49(3):223–229, March 2002.

[4] ACE Associated Compiler Experts. CoSy compiler development system. `http://www.ace.nl/compiler`.

[5] Sunghwan Kim and Jihong Kim. Opcode encoding for low-power instruction fetch. *IEEE Electronics Letters*, 35(13):1064–1065, June 1999.

[6] Mike Tien-Chien Lee, Vivek Tiwari, Sharad Malik, and Masahiro Fujita. Power analysis and low-power scheduling techniques for embedded DSP software. In *International Symposium on System Synthesis*, 1995.

[7] Jingzhao Ou and Viktor K. Prasanna. Rapid energy estimation of computations on FPGA based soft processors. In *Proc. IEEE International SoC Conference (SOCC)*, September 2004.

[8] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Energy-efficient design of battery powered embedded systems. *Special Issue of IEEE Transactions on VLSI*, May 2001.

[9] S.J.E.Wilton, S-S. Ang, and W. Luk. The impact of pipelining on energy per operation in field programmable gate arrays. In *Proc. Field Programmable Logic and Applications (FPL)*, pages 719–728, 2004.

[10] S.P.Seng, K.V.Palem, R.M.Rabbah, W.Luk, and P.Y.K.Cheung. PD-XML: Extensible markup language for processor description. In *Proc. Field-Programmable Technology (FPT)*, 2002.

[11] Ching-Long Su, Chi-Ying Tsui, and Alvin M. Despain. Saving power in the control path of embedded processors. *IEEE Design and Test*, 11(4):24–30, October 1994.

[12] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration Systems (VLSI)*, 2(4):437–445, December 1994.

[13] Seungdo Woo, Jungmin Yoon, and Jihong Kim. Low-power instruction encoding techniques. In *Proc. SOC Design Conference*, 2001.

[14] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. The microarchitecture of fpga-based soft processors. In *Proc. Compilers Architecture and Synthesis for Embedded Systems (CASES)*, pages 202–212, September 2005.

Benchmark	Optimization	Power/mW		
		Est.	Actual	Saving %
SHA-256	None	102.5	78	-
SHA-256	Scheduling only	103.5	78	0
SHA-256	Re-coding only	98.5	73.5	5.77
SHA-256	Both	93.5	67.5	13.46
MatMult	None	82.5	77.25	-
MatMult	Scheduling only	79.5	71.25	7.77
MatMult	Re-coding only	75.5	66.75	13.59
MatMult	Both	74.4	60.75	21.36
CRC-32	None	131.5	89.25	-
CRC-32	Scheduling only	131.5	89.25	0
CRC-32	Re-coding only	120.5	78	12.61
CRC-32	Both	120.5	78	12.61

(a) 4 Stage Pipeline @ 66MHz

Benchmark	Optimization	Power/mW		
		Est.	Actual	Saving %
SHA-256	None	93.5	114	-
SHA-256	Scheduling only	89.5	72	36.84
SHA-256	Re-coding only	92.5	108.75	4.61
SHA-256	Both	88.5	69	39.47
MatMult	None	74.5	72.75	-
MatMult	Scheduling only	71.5	76.75	8.25
MatMult	Re-coding only	70.5	69.75	4.12
MatMult	Both	68.5	60.75	16.49
CRC-32	None	132.5	99.75	-
CRC-32	Scheduling only	132.5	102	-2.26
CRC-32	Re-coding only	128.5	97.5	2.26
CRC-32	Both	128.5	96	3.76

(b) 5 Stage Pipeline @ 66MHz

Benchmark	Optimization	Power/mW		
		Est.	Actual	Saving %
SHA-256	None	68.5	42	-
SHA-256	Scheduling only	71.5	42	0
SHA-256	Re-coding only	68.5	40.5	3.57
SHA-256	Both	66.5	31.5	25
MatMult	None	54.5	42.75	-
MatMult	Scheduling only	51.5	35.25	17.54
MatMult	Re-coding only	47.5	30.75	28.07
MatMult	Both	51.5	31.5	26.32
CRC-32	None	88.5	48	-
CRC-32	Scheduling only	88.5	49.5	-3.13
CRC-32	Re-coding only	78.5	37.5	21.88
CRC-32	Both	78.5	39	18.75

(c) 4 Stage Pipeline @ 50MHz

Benchmark	Optimization	Power/mW		
		Est.	Actual	Saving %
SHA-256	None	60.5	69.75	-
SHA-256	Scheduling only	64.4	40.5	41.94
SHA-256	Re-coding only	60.5	67.5	3.23
SHA-256	Both	61.5	32.25	53.76
MatMult	None	49.5	42	-
MatMult	Scheduling only	47.5	35.25	16.07
MatMult	Re-coding only	44.5	33.75	19.64
MatMult	Both	43.5	26.25	37.5
CRC-32	None	93.5	58.5	-
CRC-32	Scheduling only	93.5	60	-2.56
CRC-32	Re-coding only	89.5	55.5	5.13
CRC-32	Both	89.5	57	2.56

(d) 5 Stage Pipeline @ 50MHz

Benchmark	Optimization	Power/mW		
		Est.	Actual	Saving %
SHA-256	None	53.5	21	-
SHA-256	Scheduling only	56.5	21	0
SHA-256	Re-coding only	54.5	19.5	7.14
SHA-256	Both	49.5	12	42.86
MatMult	None	38.5	20.25	-
MatMult	Scheduling only	35.5	14.25	29.63
MatMult	Re-coding only	32.5	12.75	37.04
MatMult	Both	33.5	12.75	37.04
CRC-32	None	63.5	28.5	-
CRC-32	Scheduling only	63.5	30	-5.26
CRC-32	Re-coding only	57.5	18	36.84
CRC-32	Both	57.5	20.25	28.95

(e) 4 Stage Pipeline @ 40MHz

Benchmark	Optimization	Power/mW		
		Est.	Actual	Saving %
SHA-256	None	46.5	40.5	-
SHA-256	Scheduling only	48.5	16.5	59.26
SHA-256	Re-coding only	43.5	37.5	7.41
SHA-256	Both	46.5	10.5	74.07
MatMult	None	30.5	17.25	-
MatMult	Scheduling only	28.5	11.25	34.78
MatMult	Re-coding only	27.5	11.25	34.78
MatMult	Both	26.5	8.25	52.17
CRC-32	None	66.5	33	-
CRC-32	Scheduling only	66.5	34.5	-4.55
CRC-32	Re-coding only	64.5	30	9.09
CRC-32	Both	64.5	31.5	4.55

(f) 5 Stage Pipeline @ 40MHz

Table 3. Measured and estimated power consumption for auto-generated SoC designs running on the XUP platform. Power estimation from XPower with signal activities from simulation of the placed and routed netlist.