Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance

David Brooks and Margaret Martonosi
Dept. of Electrical Engineering
Princeton University
{dbrooks, martonosi}@ee.princeton.edu

Abstract

In general-purpose microprocessors, recent trends have pushed towards 64-bit word widths, primarily to accommodate the large addressing needs of some programs. Many integer problems, however, rarely need the full 64-bit dynamic range these CPUs provide. In fact, another recent instruction set trend has been increased support for sub-word operations (that is, manipulating data in quantities less than the full word size). In particular, most major processor families have introduced "multimedia" instruction set extensions that operate in parallel on several sub-word quantities in the same ALU.

This paper notes that across the SPECint95 benchmarks, over half of the integer operation executions require 16 bits or less. With this as motivation, our work proposes hardware mechanisms that dynamically recognize and capitalize on these "narrow-bitwidth" instances. Both optimizations require little additional hardware, and neither requires compiler support.

The first, power-oriented, optimization reduces processor power consumption by using aggressive clock gating to turn off portions of integer arithmetic units that will be unnecessary for narrow bitwidth operations. This optimization results in an over 50% reduction in the integer unit's power consumption for the SPECint95 and MediaBench benchmark suites. The second optimization improves performance by merging together narrow integer operations and allowing them to share a single functional unit. Conceptually akin to a dynamic form of MMX, this optimization offers speedups of 4.3%-6.2% for SPECint95 and 8.0%-10.4% for MediaBench.

1. Introduction

As high-end processor word widths have made the shift from 32 to 64 bits, there has been an accompanying trend towards efficiently supporting subword operations. Subword parallelism, in which multiple 8- or 16-bit operations are performed in parallel by a 64-bit ALU, is supported in current processors via instruction set and organizational extensions. These include the Intel MMX [1], HP MAX-2 [2], and Sun VIS [3] multimedia instruction sets, as well as vector microprocessor proposals such as the T0 project [4].

All of these ideas provide a form of SIMD (single instruction-multiple data) parallel processing at the word level. These instruction set extensions are focused primarily on enhancing performance for multimedia applications. Such applications perform large amounts of arithmetic processing on audio, speech, or image samples which typically only require 16-bits or less per datum. The caveat to this type of processing is that thus far these new instructions are mainly used only when programmers hand-code kernels of their applications in assembler. Little compiler support exists to generate them automatically, and the compiler analysis is limited to cases where programmers have explicitly defined operands of smaller (i.e., char or short) sizes.

This paper proposes hardware mechanisms for dynamically exploiting narrow width operations and sub-word parallelism without programmer intervention or compiler support. By detecting "narrow bitwidth" operations dynamically, we can exploit them more often than with a purely-static approach. Thus, our approach will remain useful even as compiler support improves.

In this paper we provide two optimizations that take advantage of the core "narrow width operand" detection that we propose. The first idea watches for small operand values and exploits them to reduce the amount of power consumed by the integer unit. This is accomplished by an aggressive form of clock gating. Clock gating has previously been shown to significantly reduce power consumption by disabling certain functional units if instruction decode indicates that they will not be used [5]. The key difference of our work is to apply clock gating based on operand values. When the full width of a functional unit is not required, we can save power by disabling the upper bits. With this method we show that the amount of power consumed by the integer execution unit can be reduced by 54.1% for the SPECint95 suite with little additional hardware.

The second proposed optimization improves performance by dynamically recognizing, at issue time, opportunities for packing multiple narrow operations into a single ALU. With this method the SPECint95 benchmark suite shows an average speedup of 4.3%-6.2% depending on the processor configuration. The MediaBench suite showed an average speedup of 8.0%-10.4%.

The primary contributions of this work are three-fold: a detailed study of the bitwidth requirements for a wide-range of benchmarks, and two proposals for methods to exploit narrow width data to improve processor power consumption and performance. In Section 2 we further discuss the motivations for our work and place it in the context of prior work in multimedia instruction sets, power savings, and other methods of using dynamic data.

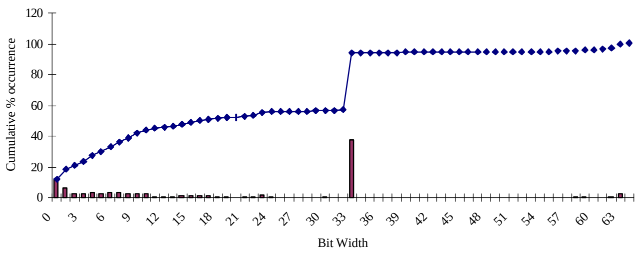


Figure 1 – Bitwidths for SPECint95 on 64-bit Alpha.

Section 3 describes the experimental methodology used to investigate our optimizations. Section 4 details the power optimization technique based on clock gating for operand size and presents results on its promise. In Section 5, we describe the method for dynamically packing narrow instructions at issue-time. Finally, Section 6 concludes and discusses other opportunities to utilize dynamic operand size data in processors.

2. Motivations and Past Work

2.1 Application Bitwidths

In this study we show that a wide range of applications have small operand sizes. Figure 1 illustrates this by showing the cumulative percentage of integer instructions in SPECint95 in which both operands are less than or equal to the specified bitwidth. (Section 3 will discuss the Alpha compiler and SimpleScalar simulator used to collect these results.) Roughly 50% of the instructions had both operands less than or equal to 16-bits. We will refer to these operands as narrow-width; an instruction execution in which both operands are narrow width is said to be a "narrow-width operation". Since this chart includes address calculations, there is a large jump at 33 bits. This corresponds to heap and stack references. (Larger programs than SPEC might have this peak at a larger bitwidth.) The data demonstrate the potential for a wide range of applications, not just multimedia applications, to be optimized based on narrow-width operands. While other such work, e.g. protein-matching codes [6], required algorithm or compiler changes, we focus here on hardware-only approaches.

2.2 Observing Narrow Bitwidth Operands

The basic tenet behind both of the optimizations proposed here is that when operations are performed with narrow-width operands, the upper bits of the operation are unneeded. For example, when adding 17, a 5-bit number, to 2, a 2-bit number, the result is 19, a 5-bit number. Only the lower five bits are needed to perform the computation. To decrease power dissipation, clock gating can disable the latch for the unneeded upper bits. Alternatively, to improve performance, we propose "operation packing", in which we issue and execute several of these narrow operations in parallel within the same ALU. In either case, the crux in exploiting narrow-width operands, however, lies in recognizing them and modifying execution. Sections 4 and 5 will discuss hardware approaches for tagging result operands as "narrow-width" as they are produced, and for storing these tags along with source operands as we stage subsequent instructions waiting for issue.

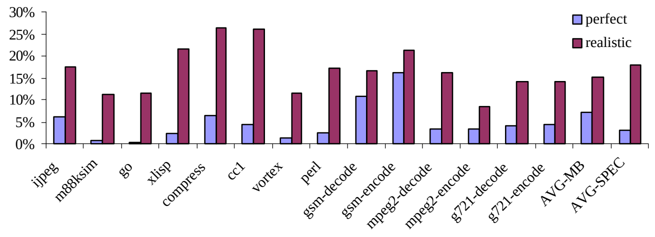


Figure 2 – Percentage of instructions whose operand precision changes from less than 16-bit to greater than 16-bit over a single program run.

2.3 Disadvantages of Static Compiler Analysis

Part of the motivation for this work was the fact that *static* analysis of input operand sizes has several disadvantages. Most importantly there are many cases where it is impossible to know what the true operand bitwidths (as opposed to the declared operand sizes) will be until run-time. Actual operand sizes depend very much on the input data presented. Operand sizes for particular instructions can also vary over the program run even with the same input data, which makes the task of the compiler even more difficult.

Figure 2 shows the percentage of PC values where operand width changes as the instruction is executed repeatedly within a single run. In particular, the figure shows how often an instruction fluctuates from having less than 16-bit operands to greater than 16-bit operands as it executes repeatedly within a single program run.

With perfect branch prediction, the instruction operand sizes are far more predictable than with realistic branch prediction. This is because with perfect branch prediction only the true execution path is seen. With imperfect branch prediction, uncommon paths, like error conditions, may be executed (but not committed) if the branch predictor points that way. Along these paths, operand statistics may be markedly different. Compile time analysis must conservatively analyze all potential paths to ensure that operations can truly be packed. This may include uncommon error conditions and other extreme cases. As a result, the compiler runs into much of the same diverse operand values as seen by imperfect branch prediction.

Overall, compiler dataflow analysis for operand sizes must be conservative about possible operand values.

Table 1 – Baseline configuration of simulated processor.

Parameter	Value
Processor Core	
RUU size	80 instructions
LSQ (ld/store queue) size	40
Fetch Queue Size	8 instructions
Fetch width	4 instructions/cycle
Decode width	4 instructions/cycle
Issue width	4 instructions/cycle (out-of-order)
Commit width	4 instructions/cycle (in-order)
Functional units	4 Integer ALUs (performing arithmetic, logical, shift, memory, branch ops), 1 integer multiply/divide
Branch Prediction	
Branch Predictor	Combining: 4K 2-bit selector, 12-bit history; 1K 3-bit local predictor, 10-bit history; 4K 2-bit global predictor, 12-bit history
BTB	2048-entry, 2-way
Return-address stack	32-entry
Mispredict penalty	2 cycles
Memory hierarchy	
L1 data-cache	64K, 2-way (LRU), 32B blocks, 1 cycle latency
L1 instruction-cache	64K, 2-way (LRU), 32B blocks, 1 cycle latency
L2	Unified, 8M, 4-way (LRU), 32B blocks, 12-cycle latency
Memory	100 cycles
TLBs	128 entry, fully associative, 30-cycle miss latency

Programmer hints about operand sizes can aid the compiler. It is unrealistic, however, to assume that programmers will provide these hints on codes other than small multimedia kernels.

From Figure 1 it is clear that many opportunities exist to exploit narrow width data for subword parallelism and aggressive clock gating. Searching for subword parallelism in applications is somewhat analogous to the search for instruction-level parallelism (ILP) in applications. In the late 80s and early 90s, most general purpose superscalar microprocessors were statically scheduled, and the compiler was responsible for uncovering ILP in programs. Current microprocessors implement aggressive dynamic scheduling techniques to uncover more ILP. This evolution was necessary to feed the wider-issue capabilities of these processors. In a similar manner, more subword parallelism can be uncovered with the dynamic approaches we propose than if one relies solely on compiler techniques.

There has been other work in specializing for particular operand values at runtime. The PowerPC 603 includes hardware to count the number of leading zeros of input operands to provide an "early out" for multicycle integer multiply operations. This can reduce the number of cycles required for a multiply from five for 32-bit multiplication to two for an 8-bit multiplication [7]. At a higher level, value prediction seeks to predict result values for certain operations and speculatively execute additional instructions based on these predicted operand values [8].

Finally, there has also been other work in exploiting narrow bitwidth operations. Razdan and Smith propose a hardware-programmable functional unit which augments the base processor's instruction set with additional instructions that are synthesized in configurable hardware at compile time [9]. Since all synthesized instructions must complete in a single cycle, bitwidth analysis is performed at compile time to highlight sequences of narrow-width operations that are the best candidates for implementation.

3. Methodology

3.1 Simulator

We used a modified version of SimpleScalar's [10] *sim-outorder* to collect our results. SimpleScalar provides a simulation environment for modern out-of-order processors with speculative execution. The simulated processor contains a unified active instruction list, issue queue, and rename register file in one unit called the reservation update unit (RUU). The RUU is similar to the Metaflow DRIS (deferred-scheduling, register-renaming instruction shelf) [11] and the HP PA-8000 IRB (instruction reorder buffer) [12]. Separate banks of 32 integer and floating point registers make up the architected register file and are only written on commit. Table 1 summarizes the important features of the simulated processor. The baseline configuration parameters are roughly those of a modern out-of-order processor.

The changes made to the simulator for this study are localized to the issue and decode stages. In decode, bitwidths are calculated for dynamic data and stored in the reservation station entry to be used during the issue stage. In the issue stage, this data is used to decide if instructions can be issued and executed in parallel based on the data from the decode stage. These changes reflect the simulator implementation; Section 4 discusses how they would be implemented in an actual processor.

3.2 Benchmarks

A goal of this study is to demonstrate and exploit the prevalence of narrow-width operations even in applications outside the multimedia domain. For this reason we evaluate the SPECint95 suite of benchmarks as well as several benchmarks from the MediaBench suite [13]. We have compiled the benchmarks using the DEC *cc* compiler with the following SPEC optimization options: `-migrate -std1 -O5 –ifo –non_shared`. In particular, the `-O5` setting, along with numerous other optimizations, provides vectorization of some loops on 8-bit and 16-bit data (char and short).

Table 2 – SPECint95 Benchmarks

Benchmark	Inputs	Warm Up Instructions
ijpeg	vigo.ppm	824M
m88ksim	dhrystone	26M
go	9stone21	926M
xlisp	All xlisp inputs	271M
compress	bigtest.in	2576M
gcc	cccp.i	221M
vortex	persons.1k	2451M
perl	scrabble game	601M

Table 3 – MediaBench Benchmarks

Benchmark	Description
gsm-encode	Audio and speech encoding with the GSM standard
gsm-decode	Audio and speech decoding with the GSM standard
mpeg2encode	MPEG digital compressed format encoding
mpeg2decode	MPEG digital compressed format decoding
g721encode	Voice compression using the G.721 standard
g721decode	Voice decompression using the G.721 standard

For this study we wanted to use the reference inputs for the SPECint95 suite. The test or training inputs are unsuitable because our data-specific optimizations might be unfairly helped by smaller data sets. Using reference inputs, the SPECint95 benchmarks run for billions of instructions, which, if simulated fully, would lead to excessively long execution times. Thus we use a methodology similar to that described by Skadron et al. [14]. We warm up architectural state using a fast-mode cycle-level simulation that updates only the caches and branch predictors during each cycle. The warmup period also avoids the effects of smaller operand sizes that are prevalent within program initialization. Using Skadron et al.'s results identifying representative program sections based on cache and branch prediction statistics, we then simulate a 100 million instruction window using the detailed simulator. Table 2 lists the reference input that we have chosen for the SPECint95 benchmarks, and the number of instructions for which we warm up the caches and branch predictor. Table 3 describes the applications chosen from the MediaBench suite. For the MediaBench suite, *gsm*, *g721*, and *mpeg2-decode* were run to completion while *mpeg2-encode* was simulated for 100 million instructions after a 500M instruction warmup period.

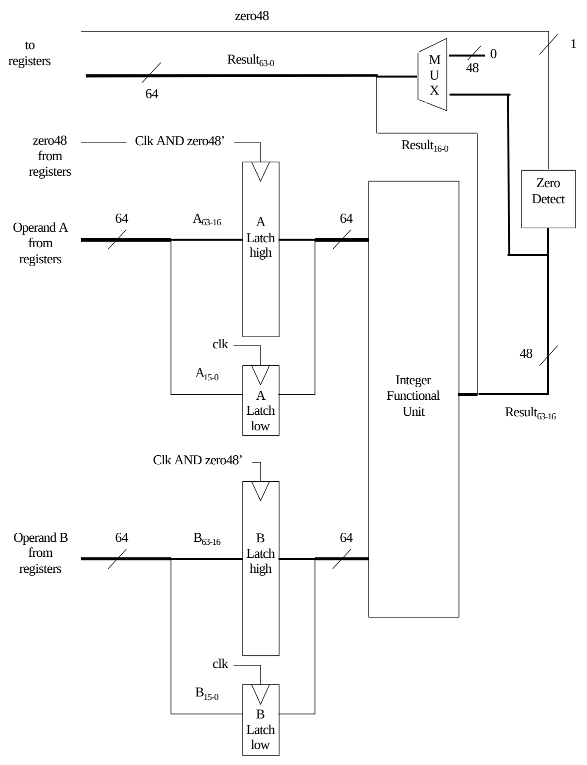


Figure 3 – Clock gating architecture.

4. Power Optimizations

4.1 Clock Gating

Dynamic power dissipation is the primary source of power consumption in CMOS circuits. In CMOS circuits, dynamic power dissipation occurs when changing input values cause their corresponding output values to change. Only small leakage currents exist as long as inputs are held constant. Clock gating has been used to reduce power by disabling the clock and thereby disabling value changes on unneeded functional units. In static CMOS circuits, disabling the clock on the latch that feeds the input operands to functional units essentially eliminates dynamic power dissipation. Power consumption on the critical clock lines is also saved because the latch itself is disabled. In dynamic or domino CMOS circuits, the same effect can be obtained by disabling the clocks that control the pre-charge and evaluate phases of the circuit.

Currently most work on clock gating has used the decoded opcode to decide which units can be disabled for a particular instruction. For example, *nop*'s allow most of the units to be disabled since no result is being computed. As another example of opcode-based clock gating, consider an "add byte" instruction. Since the opcode *guarantees* that only the lower part of the adder is needed, the top part of the functional unit is disabled.

4.2 Proposed Architecture

We propose and quantify a more aggressive clock gating approach. At run-time, the hardware determines instances when, based on the input operands, the upper bits of an operation are not needed; in those cases, it disables the upper portion of the functional unit. Key differences from prior approaches are that (1) our approach is operand-based, not opcode-based, and (2) our approach is dynamic, not static. (One could, of course, use our method *in addition to* prior opcode-based approaches.) Different program runs, or even different

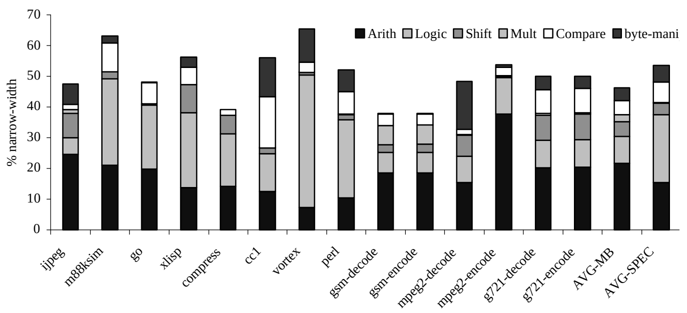


Figure 4 – Operations with both operands 16-bits or less.

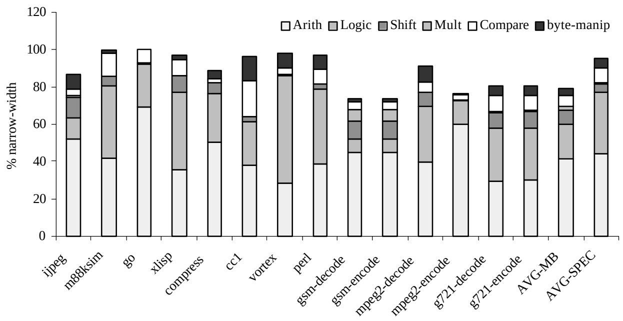


Figure 5 – Operations with both operands 33-bits or less.

executions of the same instruction, dissipate different amounts of power depending on the operands seen.

Figure 3 is a diagram of our proposed implementation. This unit recognizes that the upper bits of both input operands are zeros. For example, in an addition operation, if both input operands have all zeros in their top 48 bits, these bits do not have to be latched and sent to the functional units. We already know that the result of this part of the addition will be zero, and thus zeros can be multiplexed onto the top 48 bits of the result bus, rather than computed via the adder. In this architecture the low 16 bits are always latched normally. The high 48 bits are selectively latched based on a signal that accompanies the input operand from the reservation station. This signal, called *zero48* in Figure 3, denotes that the upper 48-bits are all zeros and is created by zero detection logic when the result was computed. Since some operands come directly from the cache, there must also be a zero-check during load instructions. We believe such zero-detects are already performed in some processors; for example, to recognize divide-by-zero exceptions early. However, in some processors it may not be possible to perform zero-detects on incoming loads, and in these cases the hardware will not recognize an opportunity to gate the clock. For the SPECint95 suite, 13.1% of power saving instructions have one or more operands that come directly from a load instruction; these are the instructions that would be missed if zero-detect were omitted on loads. The percentages for the media benchmarks are much lower at 1.5%.

In order for any power saving technique to be useful, it must save more power than it consumes. In our technique the new power dissipated is mainly in the zero-detection logic and in widening the mux onto the result bus. The primary power savings stems from selectively clock-gating the functional units based on the results of the zero-detection logic. In the following subsections we evaluate these costs and benefits in more detail.

4.3 Bitwidth Analysis of Benchmarks

The success of our approach relies on the frequent occurrence of narrow bitwidth operands. Figure 4 shows, for each benchmark, the percentage and type of operations whose input operands are both less than or equal to 16-bits. (Both operands must be small in order for the clock gating to be allowed.) The breakdown by operation type is another important metric. Intuitively, disabling the upper bits on an adder or multiplier will save more power than turning off the upper bits on the less power-hungry logical functions. Figure 4 shows that for most benchmarks arithmetic and logical operations dominate the number of narrow-width operations. In most of the benchmarks multiplies are rather infrequent although they do account for 6% of the narrow-width operations in *gsm*.

Recall that Figure 1 illustrated how address calculations result in many operations with bitwidths of 33. Figure 5 emphasizes this point. From this data it makes sense to include a second control signal for clock gating of operands that are 33-bits or less. The zero detect logic can be shared so that the extra hardware requirements are minimal. This modification is also useful for optimizing the multiplication of two 16-bit numbers. In these cases a 32-bit result can occur, so the 33-bit mux onto the result bus would be used.

In the Alpha architecture that we considered in this study, the fundamental datum is the 64-bit quadword. Quadword integers are represented with a sign bit occupying the most significant bit [15]. Numbers are expressed in two's complement form, which simplifies arithmetic operations. The techniques presented in this paper rely on determining when data requires less than the full word width of the machine. For positive numbers, this can be accomplished by performing a zero detect on the high order bits. For negative numbers in the two's complement representation, leading 1's signify the same thing that leading 0's do for positive number – essentially unneeded data. Thus a ones detect must be performed in parallel with the zero detect computation to detect narrow bitwidth negative numbers.

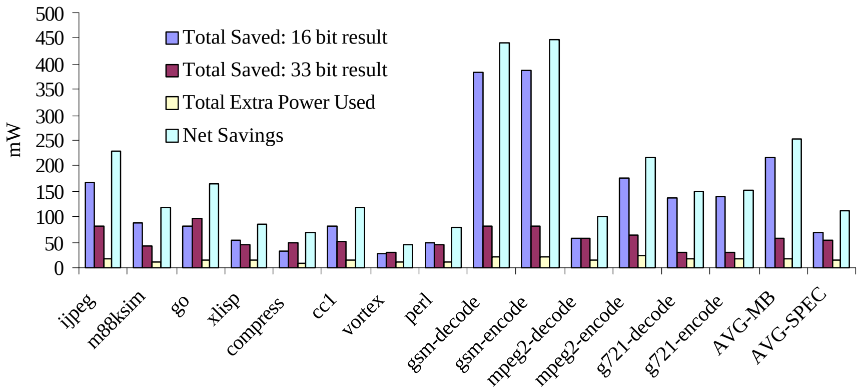


Figure 6 – Net power saved by clock gating at 16 and 33 bits. Total extra used is the amount used by zero detection and muxing. Net savings denotes the amount saved at 16 bits plus the amount saved at 33 bits minus the amount used. Numbers are per cycle.

4.4 Power Results

The amount of power that is saved by our approach depends on both the type and frequency of narrow width operations. In order to quantify the amount of power saved, we use previously-reported research to estimate the amount of power that various functional units use [16, 17, 18]. From these sources we obtain power estimates assuming dynamic logic and relatively fast carry look-ahead adders. We assume that the multiplier is pipelined with its power usage scaling linearly with the operand size. Table 4 summarizes the values that we have assumed for different size devices. The functional units in current high-end microprocessors are likely to use even more power, but detailed numbers are not yet available in the literature. For this analysis though, the important factor is the ratio of the respective functional units to each other.

Figure 6 summarizes the amount of power saved and expended per cycle. We arrived at these numbers by determining the amount of power saved and expended per instruction executed and multiplying by the average issue rate. These results include all loads, stores, branches, and other integer execution unit instructions that are not part of the set of instructions that our optimization applies to. Among the SPECint95 benchmarks, our technique saves the most power for *ijpeg* and *go*. *Ijpeg* has a large number of narrow-width arithmetic operations. *Go* includes a large number of address calculations and is helped the most by adding the extra signal to detect 33-bit operations. The media benchmarks tend to save even more power than the SPECint95 benchmarks. This is primarily because of the larger number of arithmetic operations. *GSM*, in particular, has a relatively large number of narrow bitwidth multiply operations. The amount of power used by the zero detection circuitry is small and nearly constant for all benchmarks. In no case does the amount of power used for zero detection exceed the amount of power saved.

Figure 7 shows the total amount of power that is saved by the integer unit with our optimization. For the baseline system, we assume that all operations use the amount of power that a 64-bit device would use. (We assume basic clock gating in which, for example, multipliers are turned off for add instructions and vice versa.) For the SPECint95 benchmark suite, the average power consumption of the integer unit was reduced by 54.1%. For the media benchmarks, the reduction was 57.9%.

While a 50-60% power reduction seems exceptional, it is important to note that the integer unit's contribution to total power varies depending on the CPU. In some high-end CPUs much of the power is spent on clock distribution and control logic, and thus the integer unit represents only about 10% of the power dissipation [19]. In such a processor, our optimizations will lead to 5-6% power reductions on average. As control is streamlined, either in DSPs or via explicitly-parallel instruction computing

Device	32-bit	48-bit	64-bit
Adder (CLA)	105	158	210
Booth Multiplier	1050	1580	2100
Bit-Wise Logic	5.8	8.7	11.7
Shifter	4.4	6.6	8.8
Zero-Detect	--	4.2	--
Additional Muxes	--	3.2	--

Table 4 – Estimated power consumption of functional units at 3.3V and 500Mhz (mW).

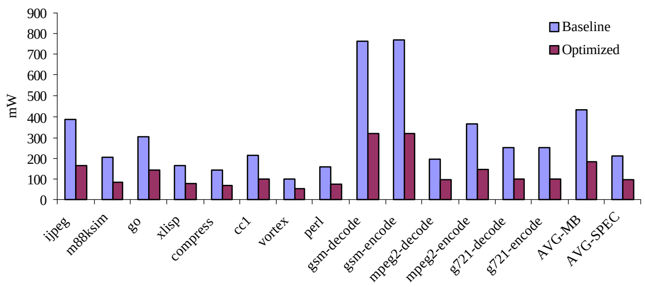
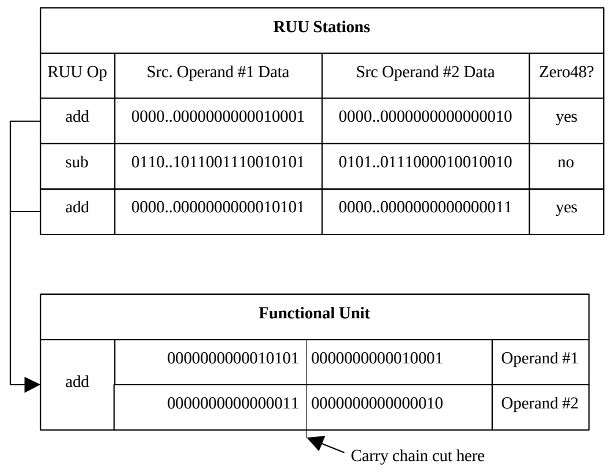


Figure 7 – Power usage of integer unit (per cycle).



RUU Stations			
RUU Op	Src. Operand #1 Data	Src Operand #2 Data	Zero48?
add	0000..0000000000010001	0000..0000000000000010	yes
sub	0110..1011001110010101	0101..0111000010010010	no
add	0000..0000000000010101	0000..0000000000000011	yes

Functional Unit			
add	0000000000010101	0000000000010001	Operand #1
	0000000000000011	0000000000000010	Operand #2

Figure 8 – Packing two add instructions with narrow operand widths.

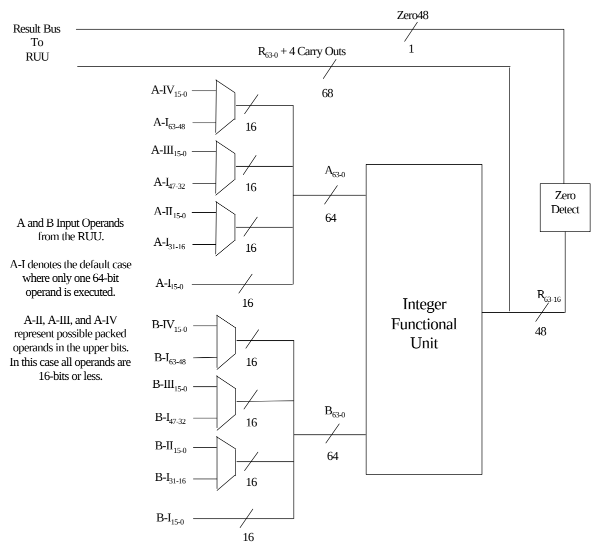


Figure 9 – Datapath Modifications.

(EPIC) as in future Intel processors [20], the integer unit is a larger factor in the processor's total power dissipation, as much as 20-40%. In these cases, the total power savings from our technique will approach 20%. In all processors, our approach promises a relatively easy way to prune power from the integer unit where this is important. We also note that our power savings estimates are somewhat conservative. The clock gating technique also reduces the switch capacitance seen by the clock distribution network, and this can lead to a further power reduction. Although this effect can be significant, it cannot be quantified without a chip floorplan.

5. Operation Packing

In this section, we present a technique to increase performance by exploiting dynamic data values. As with the prior technique, this relies on dynamically recognizing zeros in the upper bits of the input operands to take advantage of the unused upper bits in the functional units. Since the power optimization involves clock gating functional units and the performance optimization involves executing instructions in parallel, only one technique can be used at a time. However, because the techniques share a common hardware base, one could implement both and choose between them. For example, one could use thermal sensory data to have the processor switch between the two techniques, depending on current thermal or performance concerns. Related but simpler approaches are already found in commercial processors; for example, the IBM/Motorola PPC750 is equipped with an on-chip thermal assist unit and temperature sensor which responds to thermal emergencies by controlling the instruction fetch rate through I-cache throttling [21].

5.1 Background

Multimedia instruction sets define new instructions to perform a common operation on several subwords in parallel. For example, the Parallel Add instruction in HP-MAX performs four parallel additions on the 16-bit subwords that reside in the two specified 64-bit source registers. Few hardware changes are necessary to support these additional instructions; only the carry chain between the 16-bit chunks must be handled differently. Figure 8 demonstrates how two `add` instructions in the RUU, both with narrow operand widths, can be packed together at issue time into one functional unit. In this example, there are three instructions in the RUU: an `add` with source operand values of 17 and 2, a `sub` with source operands that are larger than 16-bits, and another `add` with source operands of 21 and 3. In this case, the two `add` instructions both have narrow width operands, so a single 64-bit adder can perform the two additions in parallel. The hardware built into ALUs for multimedia instruction sets will automatically stop the carry at 16-bit boundaries.

In machines with multimedia extensions, programmers or (less frequently) compilers statically generate code using multimedia instructions. As previously discussed, there are several shortcomings to this method. For those reasons, this section introduces an approach that is akin to dynamically generating multimedia instructions. In this study, we focus on merging narrow integer operations into parallel sub-word operations as currently supported by multimedia instruction set extensions. This is a subset of the operations that we explore in Section 4 and consists of the arithmetic, logical, and shift operations. For example, we do not attempt to pack multiply operations, although in some implementations this would be possible.

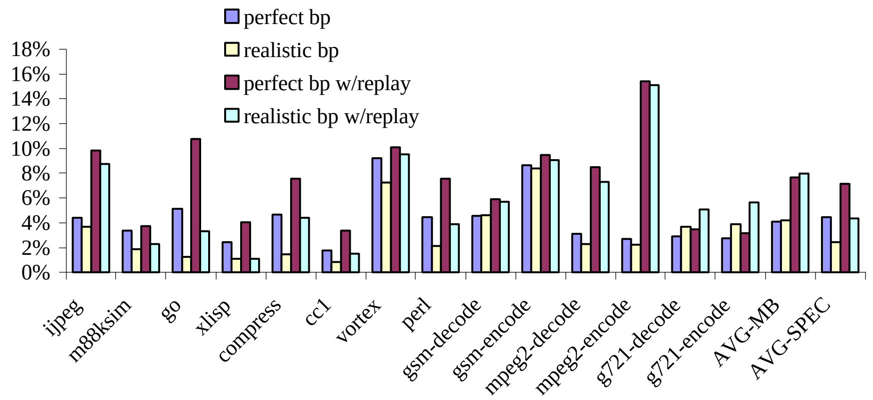


Figure 10 – Speedup due to operation packing.

5.2 Proposed Architecture

Figure 9 is a diagram of the proposed changes to the datapath. The most notable changes to the datapath are the additional muxes which move data from the low 16 bits of the source RUU stations onto the higher 16-bit paths of the source operand bus. In addition, 4 extra lines are needed on the result bus for the carry-out that could result when two 16-bit operands are added. These additional carry-out lines are needed because most multimedia instruction sets provide a form of saturating arithmetic which, upon overflow of two 16-bit values, sets the result to the maximum 16-bit value, namely 0xFFFF. In addition, the reservation stations must be modified. The additional hardware needed here includes muxes which reverse the effect of the above; data in the higher 16-bit subwords of the result bus are muxed into the low 16-bit boundaries to be written back to the result reservation station. It should be noted that much of this "additional" muxing hardware most likely already exists in processors with multimedia instruction sets. For example, the HP MAX-2 instruction set includes instructions to select any field in a source register and right-align it in the target register. Instructions also exist to select a right-aligned field from the source register and place it anywhere in the target register.

Our core idea is similar to the power optimization discussed in the last section. Each entry in the reservation update unit (RUU) stores an extra bit for each operand indicating that the size of the operand is 16-bits or less. These fields are updated when operands are computed and stored in the RUU buffers. Using these fields, the issue logic can recognize opportunities to pack narrow width operations together to share one integer ALU in the same way that the multimedia instructions do. In order for two operations to be packed, three things must occur. First both instructions must have satisfied their data dependencies and be ready to issue. Second, both instructions must have narrow width operands. Finally, they must perform the same operation.

The issue logic issues ready instructions from the RUU using its normal algorithm. In most processors, this algorithm issues the oldest instructions in the RUU which are ready to issue. However, when operations are issued in which both operands are 16-bits or less, an opportunity for packing exists. The issue logic must keep track of which issuing instructions are available for packing. If other instructions that perform the same operations are available to issue and have narrow-width operands, these instructions can be packed. The issue logic will set the appropriate muxes to issue the packed instructions in parallel.

After the instructions issue, they execute in the same fashion as packed instructions do in the multimedia instruction sets. When execution completes, the result operands share the result bus and are sent back to their respective RUU station as well as RUU stations awaiting the results as input operands. This optimization opens up machine issue bandwidth and integer ALUs available for certain integer executions. Much of the required multiplexing hardware already exists within processors designed with multimedia instruction sets. These processors also have functional units that are designed to disable the carry chain at 16-bit intervals. The primary hardware cost for this optimization is in the increased complexity of the logic that decides when packed instructions can issue. Handling negative numbers adds additional complexity to the issue logic.

5.3 Replay Packing: Speculating on Operand Size

The architecture in Section 5.2 is designed to pack operations together to be executed in parallel when both input operands are less than 16-bits. The requirement that both input operands be less than 16-bit excludes a large number of arithmetic operations used for memory addressing, loop incrementing, etc. In these cases, one of the input operands may be very large, while the other is

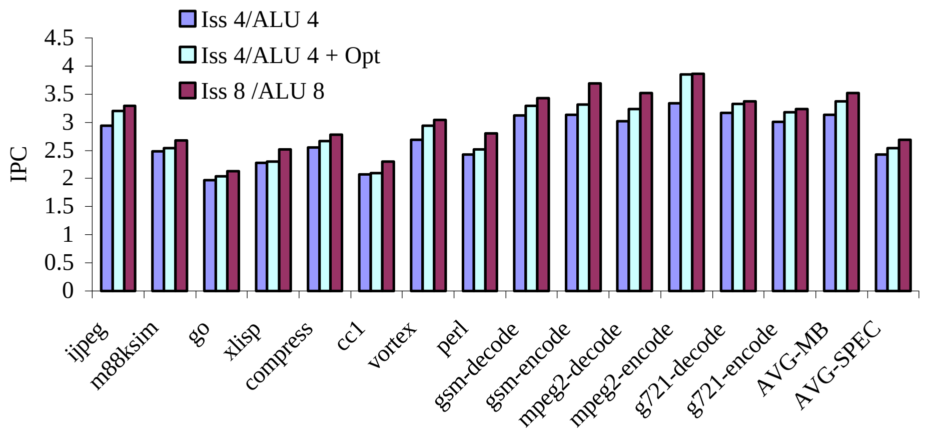


Figure 11 – IPC for the baseline system, the optimized system, and an 8-issue system.

quite small. If one operand is large and the other is small, in most arithmetic operations only the lower bits of the result will change. However, in some cases, the carry will ripple all the way down to the higher bits. In practice, this happens relatively infrequently. Based on this observation, we propose an extension to the architecture in Section 5.2 that allows operands to be packed if only one of the two input operands is less than 16-bits. In most cases there will be no overflow from the 16-bit addition, and the high 48-bits of the larger source operand can be muxed into the destination RUU station. However, in the rare cases that there is overflow from the 16-bit addition, the instruction can be squashed and subsequently re-issued as a full-width instruction. Such a situation could be handled by “replay traps”, which are already available for other reasons in the Alpha 21264 and other CPUs.

5.4 Operation Packing Results

In this section we present the results for the speedup with and without replay packing. We have considered two configurations: The first configuration is exactly the same as the baseline configuration discussed in Section 3.1. The second configuration increases the decode bandwidth from four instructions per cycle to eight instructions per cycle. The increased decode bandwidth causes the RUU to fill up faster giving more opportunities for packing.

Figure 10 shows the percent speedup over the baseline system in the configuration with the decode width of four. In this chart, we include results for both perfect and the combining predictor. In most cases moving from perfect to realistic branch prediction shows a performance degradation, because it reduces the number of useful instructions that are ready to issue each cycle. We can see that *go*, notorious for its poor branch prediction, is affected the most. *Ijpeg* and *vortex*, on the other hand, see little difference in the speedup between perfect and the realistic predictor. The average speedup across SPECint95 was 7.1% for perfect branch prediction and 4.3% with the realistic predictor. As one might expect, the multimedia benchmarks performed better than SPECint95. Here better branch prediction led to only a small difference in speedup between perfect and realistic predictors. In fact, *g721* had higher speedup with realistic branch prediction, due to speculative execution. Speculative instructions that will eventually be squashed still get executed until the branch is resolved; packing them with other instructions can increase effective issue bandwidth. The average speedup for the media benchmarks was 7.6% with perfect branch prediction and 8.0% with the combining predictor.

We also studied the packing optimization with 8-wide decode. As expected, the optimization performs better with increased decode bandwidth, because the RUU is filled with more useful instructions which have the potential to be packed, issued, and executed in parallel. Most of the benchmarks show a 2-3% increase in speedup with the increased decode bandwidth. The average speedup for SPECint95 was 9.9% for perfect branch prediction and 6.2% with the combining predictor. The multimedia benchmarks performed better as well, but not as significantly as SPECint95. This is because multimedia applications have many loop-oriented arithmetic operations with few data dependencies. This gives them a larger pool of usable instructions even with the smaller decode bandwidth. The average speedup for the media benchmarks was 10.3% with perfect branch prediction and 10.4% with the realistic predictor.

As previously mentioned, the proposed optimization increases the effective issue bandwidth and number of integer ALUs by packing several instructions and issuing and executing them in parallel. Thus, it is useful to compare our optimization to a machine that simply has more issue and execution bandwidth. Figure 11 compares instructions per cycle (IPC) for three different configurations, all with combining branch prediction and decode and commit width of four. The first is the baseline machine with issue width of 4 and 4 integer ALUs. The second is the baseline machine augmented with our operation packing optimizations. The third machine is the baseline machine with an issue width of 8 and 8 integer ALUs. *Ijpeg* and *vortex*, as well as many of the media benchmarks, come very close to achieving the same IPC as the more costly 8-issue/8-ALU implementation.

6 Conclusions and Future Work

Increased interest in support for sub-word parallelism motivated this work on value-specific power and performance optimizations in current microprocessors. Prior use of multimedia-style operation packing has required significant programmer intervention. Compile-time analysis is constrained by the fact that the operand range may vary over the course of a program run depending on the input data. In addition, the compiler must conservatively analyze all potential paths taken. Our work notes that certain uncommon paths may have different operand size characteristics than the typical path through programs.

Thus, in order to augment compile-time analysis, we present two techniques to *dynamically* exploit low bitwidth data. The first reduces power in integer execution units with aggressive clock gating, after determining that the upper portion of functional unit is not needed. The second increases performance by dynamically recognizing opportunities to issue multiple narrow width instructions to the same functional unit to be executed in parallel. The mechanisms we discuss could be extended to other optimizations as well, such as reducing power in the floating point units or in the cache memories.

A key characteristic of our current proposals is that they require only a small amount of hardware and no compiler intervention. More broadly, they represent a step towards implementing operand-value-based optimization strategies throughout processors.

Acknowledgments

We thank Saman Amarasinghe, Douglas W. Clark, and the reviewers for their helpful comments on early paper drafts. Research support includes funds from DARPA DABT63-97-C-1001, NSF MIP-97-08624, and a donation from Intel Corp. Brooks is supported by an NSF Graduate Research Fellowship, and Martonosi by an NSF CAREER Award.

References

[1] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, vol.16, no.4, p. 42-50.

[2] R. Lee. Subword parallelism with MAX-2. *IEEE Micro*, vol.16, no.4, p. 51-9.

[3] M. Tremblay et al. The Visual Instruction Set (VIS) in UltraSPARC. *Proc. COMPCON* 1995, p. 462-469.

[4] Asanovic K., Kingsbury B., Irissou B., Beck J., and Wawrzynek J. T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines. *Proc. 22nd European Solid-State Circuits Conference*, Sep., 1996.

[5] R. Gonzalez and M. Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, vol.31, no.9, p. 1277-84.

[6] Alpern, B., L. Carter, and K. S. Gatlin. Microparallelism and High-Performance Protein Matching. *SuperComp.95*

[7] G. Gerosa, et al. A 2.2W, 80 MHz Superscalar RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, vol. 29, no. 12, p. 1440-54.

[8] M. Lipasti, J.P. Shen. The performance potential of value and dependence prediction. *Proc. 3rd International Euro-Par Conference*. p. 1043-52

[9] R. Razdan and M. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. *Proc. of Micro 27*. p. 1-9. Nov 1994.

[10] D. Burger, T.M. Austin, and S. Bennett. Evaluating future microprocessors: the SimpleScalar tool set. TR-1308, Univ. of Wisconsin-Madison CS Dept., July 1996.

[11] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman. The Metaflow architecture. *IEEE Micro*, p. 10-13, 63-73, June 1991.

[12] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *CompCon '95*, p. 123-128, Mar. 1995.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, MediaBench: A Tool for Evaluating Multimedia and Communications Systems. *Proc. of Micro 30*, 1997.

[14] K. Skadron, et al. A Quantitative Evaluation of Branch Prediction's Impact on Instruction-Window Size and Cache Size. Princeton Univ. CS Dept. TR-578-98.

[15] D. Bhandarkar. Alpha Implementations and Architecture – Complete Reference and Guide. Digital Press, 1996.

[16] R. Zimmermann and W. Fichtner. Low-Power Logic Styles: CMOS Versus Pass-Transistor Logic. In *IEEE Journal of Solid State Circuits*, vol. 32, no. 7, p. 1079-90.

[17] M. Borah, R. Owens, M. Irwin. Transistor Sizing for Low Power CMOS Circuits. *IEEE Trans. on CAD for Integrated Circuits and Systems*, vol. 15, no. 6, p. 665-71.

[18] P. Ng, P. Balsara, D. Steiss. Performance of CMOS Differential Circuits. *IEEE Journal of Solid State Circuits*, vol. 31, no. 6, p. 841-846.

[19] M. Gowan, L. Biro, D. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. *Proc. 35th Design Automation Conference*. p. 726-731. June, 1998.

[20] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, vol. 31, no. 7, p. 24-32.

[21] H. Sanchez *et al.* Thermal management system for high performance PowerPC microprocessors. *Proc. COMPCON 1997*, Feb. 1997.