Variable Word Width Computation for Low Power

Sayf Alalusi and Bret Victor

Abstract— *Today's mobile processors must often times tackle a multitude of computing applications, many of which do not require the full 32-bit data word for their computations. In light of this, it is possible to restrict the computations for these applications to a reduced width data word to minimize power consumption. A modification to a standard RISC architecture is presented to reduce the overall power consumption of these reduced-width applications. The processor is modified to allow computations to be performed on either 32-bit or 16-bit wide data. Instructions remain 32 bits, and 32-bit memory calculations can be interspersed with 16-bit data operations. An analysis is presented of the approximate power savings of a processor employing these modifications over the same processor without these modifications.*

I. INTRODUCTION

VIRTUALLY all general-purpose processors today use 32-bit data words. This is for many reasons, not the least of which is the efficient and quick access to much larger memory areas than are available with 16-bit data words. Another trend today is toward mobile computing solutions that must often tackle general purpose computing applications, as well as multi-media processing applications.

A major constraint on mobile computing platforms is their reliance on an internal power supply, i.e. a battery. This means that energy is at a premium, and should not be wasted. Energy consumption can be reduced through circuit and architecture techniques, such as lowering the supply voltage and removing higher performance functionality that is not necessary.

The goal of this project was to explore whether further energy savings are possible by considering the word width requirements of the application. Even though a 32-bit data word is required in general, certain applications may not require that computation be performed on the full 32-bit data word. For example, image processing applications will never require more than 24 bits (and commonly only 16 bits), audio processing typically only requires 16 bits, text processing only requires 8 bits, and Boolean logic only requires 1 bit. We sought to discover the energy savings possible if the processor can gain and then exploit this knowledge.

Energy is consumed in a processor when a node that has an associated capacitance makes a transition from one voltage level to another, or equivalently in this case, from one logic level to another. The energy consumed by a node is directly proportional to the capacitance at the node in question. So to minimize power consumption, nodes should be prevented from switching unnecessarily. The energy consumption can be calculated by summing all the capacitance that is switching.

II. DESIGN DESCRIPTION

The goals were to develop a fairly simple and general set of modifications that could be applied to an architecture in order to save power in applications where the data words are not 32-bit, and to understand how much savings is possible and where it comes from. Our design was based around the standard MIPS instruction set and five-stage pipeline, which is a typical 32-bit RISC architecture. We were faced with many possibilities at each step of the design phase. Our design decisions will be explained in this section, and the energy analysis of the design in the next section.

The design consists of two classes of modifications to the standard architecture. The first class is the modifications needed simply to communicate the current data word width to the processor. This can be done dynamically, with the processor somehow sensing how many data bits are "active", or statically, through the compiler or programming language. A dynamic scheme would require no modification to the instruction set or decode and could be "smarter" than a static scheme. However, it would incur a large overhead in hardware area, complexity, and power, and since we sought a solution that was simple and consumed little extra energy itself, we chose static.

The possibilities at this point were supplying additional instructions to set the processor word width in a modal fashion, or adding a field to the instruction encoding so that the width would be set on an instruction-by-instruction basis. The second possibility entails a much more drastic modification to the ISA, either expanding the instruction width or recoding the instructions. However, all memory accesses and pointer arithmetic require the full 32 bits, so a typical application needs to switch frequently between word widths. Having to issue extra instructions to do so would incur a large overhead. Also, specifying width on an instruction-by-instruction basis leads to a much cleaner solution than using a global mode, because the word width information can easily travel down the pipeline with the instruction itself. Therefore, we decided to specify data width directly in the instruction word.

Finally, we had to choose which word widths to support. We decided on supporting only 16-bit and 32-bit instructions, primarily to simplify the control and decode logic, and for ease of analysis. (However, supporting more than two word widths, while more complicated, would not require a major change to our design.) Differentiation between word widths can be accomplished with a single flag bit in the instruction word. We now have two versions of many instructions – `add16` and `add32`, for example. The compiler can easily choose which version to use through a programming language mechanism, such as "`short int`" versus "`int`" in C.

The second class of modifications is what actually saves energy, now that the data word width is known. The underlying philosophy behind this class is that we only drive the lines than we need to drive, we only clock the registers that need to be clocked, and we keep the control logic simple. This minimizes the amount of capacitance transitioning in the system, and thus the energy consumption. The downside of this approach is that it introduces internal indeterminacy – unused data lines and register bits will have "garbage" values, left over from previous operations. In most cases, however, this is not a problem.

In the following discussion, we step through the pipeline as shown in Figure 1 as we discuss the modifications made to each stage. The low-level details and energy analysis of the modifications will be presented in Section III.

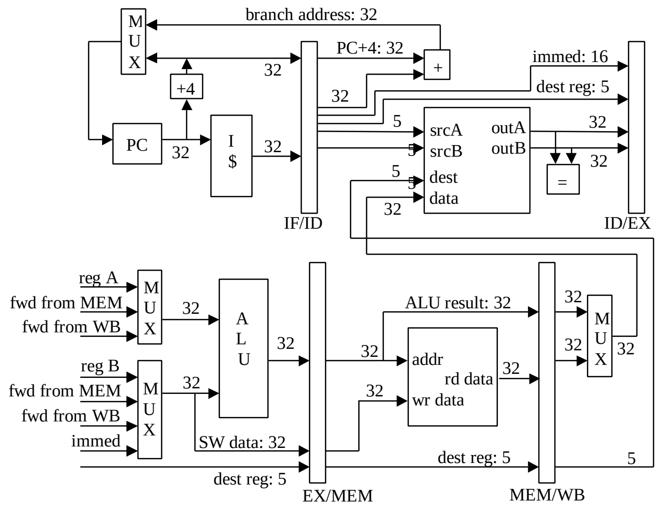

FIGURE 1: PIPELINE OVERVIEW

The first stage is Instruction Fetch. Since this stage deals only with instruction words and address words, no energy-saving modifications can be applied.

The second stage is Instruction Decode, which includes register file reads. Several modifications can be applied. One modification, which we will see in subsequent pipeline stages as well, is clock gating of the following pipeline register. This not only saves on energy dissipated in switching the clock line and changing the contents of the register, but also prevents the output of the register from switching, which in turn minimizes the switching activity in later stages. However, one must apply clock gating with care, because it can lead to skew. Also, we are attempting to gate the clock of the following pipeline register using a result stored in the previous pipeline register, which brings up some tricky timing issues. The circuit in Figure 2 attempts to address both of these concerns. The buffering stages illustrate how the gated clock can be fit into the clock distribution network. The buffers can be sized appropriately in order to match the arrival times of the clocks at the registers. The flip-flop is used to sample the width flag on the falling edge of the clock, and sidestep the second concern above. We partition the pipeline register so the upper 16 bits ("high word") of each register file output is only latched into the pipeline register if the processor is executing a 32-bit operation, using the WidthGatedClock Additionally, we use another gated clock, ImmedGatedClock, to store the immediate field of the instruction word only if the instruction requires an immediate operand. However, we are not assuming that this information is encoded in a single bit like the width, so it may require some decode logic to determine whether the immediate operand is required. We could further use this information to not latch any of register B if the immediate operand is used, since in such a case, register B is discarded in the next stage. However, instructions without immediate operands are issued more frequently than instructions with immediates, so the savings from such a modification would be smaller, and not worth the control and complexity overhead.

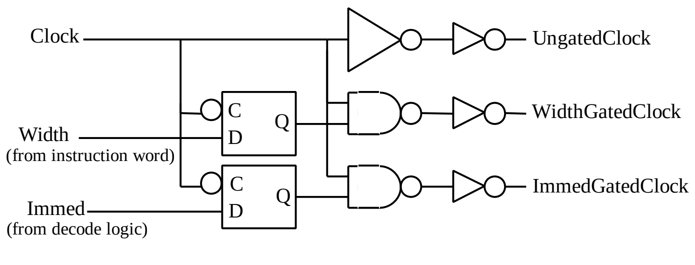

FIGURE 2: GATED CLOCK GENERATION

Other modifications to the Instruction Decode stage are preventing the register file from driving the high bit lines of its outputs if the word width is 16-bit, and ensuring that the comparitor, which is used for branch decisions, only does a 16-bit compare in such a case. The first modification provides an energy savings, whereas the second one is simply required so the processor behaves correctly when branching on 16-bit results.

The third pipeline stage is Execute, where the ALU is the big player. We first modify the multiplexors to ensure that on 16-bit operations, the high words of their outputs cannot change, which prevents the ALU inputs from transitioning unnecessarily. We also modify the ALU to internally perform 16-bit operations and only drive the low word of the output on short width instructions. The following pipeline register is gated as well. The registers storing the high word output of the ALU are not clocked if the word width is short, and the registers storing the direct output of register B are only clocked if the instruction is a "store word" operation, which is the only case in which this data is needed.

The fourth pipeline stage is Memory Access, where the data cache is read and written. Our cache design "packs" 16-bit words into the memory, allowing them to be accessed on 16-bit boundaries as well as 32-bit boundaries. This is unlike our policy everywhere else in the system, where we transport 16-bit data along the existing 32-bit architecture with no packing or realignment. There are numerous reasons for packing in the memory: the control overhead is small, it avoids exposing internal indeterminacies to the external system, it allows arrays of C "`short int`"s to be used without modification, and it increases cache block size, better exploiting spatial locality with no relative increase in miss penalty. The data cache is segmented such that a 16-bit access only drives the word line across the necessary bits, which results in significant energy savings. In addition, control is added so that if the instruction is not a memory operation, no word line is driven at all.

The final stage is Write Back, where the computation results are written into the register file. Modifications here include ensuring that the multiplexer does not drive the high word on a 16-bit instruction, and only writing the low word into the destination register in such a case. However, as we will see, the second modification is optional because it does not result in a significant energy savings.

III. ANALYSIS OF DESIGN

Models and Assumptions

To determine the energy savings from our design, the capacitance at each node must be determined. Due to the large number of nodes and the complex connections at each one, a simple device parasitic model needs to be utilized. We model a transistor as having equal input and output capacitances (as calculated in [3]), equal to a basic unit, "C." We can then easily approximate the amount of capacitance at a given node, and perform our calculations in terms of C. Energy consumption at that node is then the product of its capacitance and switching frequency. The numerical value of C is unimportant because our energy calculations are inherently relative.

Another assumption that is needed in order to perform our analysis is the relative frequencies of different classes of instructions. These are calculated from the instruction mix breakdown given in [1]

Instruction Type	Percent of Total Instructions
Load + Store	35%
Addition	14%
Compare	14%
Branch (any)	20%
Shift	4%
Logical	9%

TABLE 1: INSTRUCTION MIX MODEL

for the SPEC benchmarks and are summarized in Table 1. We apply these numbers and Amdahl's law to our design in order to calculate the energy savings of the system as a whole.

HWTE

When comparing our design to an unmodified processor, we must be careful to assess our design's energy savings relative to a processor running the same type of application, specifically one in which the data being operated upon is 16-bit. It is important to note that if the processors were operating only on 16-bit values, our design would show control savings relative to the conventional processor, but in general, *no savings along the datapath.* The reason is that once the high word along the datapath were set to zero, as it would be for 16-bit data, it would remain that way, and no transitioning along the high half of the datapath would occur in either processor.

However, a 16-bit application does not deal solely with 16-bit data values, because the address space is still 32 bits. Thus, all memory accesses and pointer arithmetic, which make up a significant fraction of the instruction stream, are full 32-bit operations. We will model a 16-bit application as having two types of data. "Computational data" is 16-bit; the high word is zero. "Address data" is 32-bit, and the high word is approximately constant. This is equivalent to saying that most memory accesses occur within a 64 Kbyte block of the address space, which, while not entirely correct, is a reasonable enough model.

When dealing with one of these data types, a conventional processor does not consume more energy along the datapath than our design, because the high half of the data path does not transition. However, when the program switches from one type of data to another (a computational data instruction followed by an address instruction, for example), the processor must drive the high word of the data bus to the high word of the data type, and the consequences ripple all the way down the datapath. Our design, on the other hand, *always* has the high word of the address data "sitting" on the high word of the data bus, and simply chooses to use it or not depending on what type of instruction is requested. Thus, in this model, our design *never* expends the energy required for a "32-bit operation", whereas a conventional processor must expend the full 32-bit operation energy when switching between data types. We therefore define a concept called the High Word Transition Energy (HWTE). It is the amount of energy that the conventional processor consumes switching the high word of the datapath, and is also the amount of datapath energy that our design saves over that processor.

$$HWTE = \frac{(E_{32} - E_{16})}{N}$$

where E_{32} is the energy required for a 32-bit operation, E_{16} is the energy required for a 16-bit operation, and N is the average number of consecutive instruction that use the same data type. You can see that the HWTE is proportional to the frequency at which the program switches between data types.

The datapath blocks where the HWTE effect must be taken into account are the register file, the adder, and logic units. The multiplier and shifter are not affected because they would never be used on address data. Energy consumption in the data cache is dominated by the control. Clock energy, determined by the gating of the pipeline registers, is dependent on the ratio of 32-bit instructions to 16-bit instructions, but not the switching frequency.

Register file

The modified register file read port is shown in Figure 3. It is implemented similar to standard register file read port, where the register name enters a decoder, which tells one of the registers to drive the output bus. The control logic necessary to prevent the high word from being driven unnecessarily consists of one AND gate per register. On a given read cycle, one AND gate transitions, which can prevent up to 16 output lines from transitioning. Since each output line is wired to 32 register outputs, they carry a very large capacitance which dominates that of the AND gate. The energy loss from the control overhead is negligible in comparison to the datapath savings, and we can say that a 16-bit register read takes 50% less energy than a 32-bit read. Taking the HWTE effect into account, our design thus provides an energy savings of 33%.

The modified register file write port is shown in Figure 4. The register name enters a decoder, and the select signal is ANDed with the write control signal to clock the register. The logic necessary to prevent the high word from being written on 16-bit operations consists of one AND gate per register, plus one to generate the HiWrite signal. It turns out that the energy required for this extra logic is about equal to the energy saved by not transitioning the high 16 clock lines, so there are no control line energy savings for 16-bit operations (and indeed, a relative energy loss on 32-bit operations). Furthermore, there are little datapath savings resulting from not writing the high word into the register, because the high words of the register file are unlikely to change in a 16-bit application. Unlike a common data bus, which must be driven to every data word that comes down the pipe, registers are special purpose, and the type of data stored in an individual register will not change for relatively long periods of time. Thus, for the register file write port, control overhead cancels out clock line savings, and datapath savings are negligible.

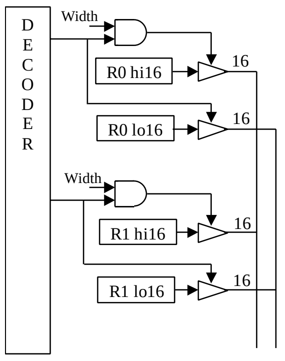

FIGURE 3: READ PORT

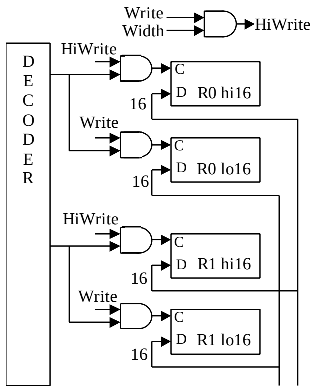

FIGURE 4: WRITE PORT

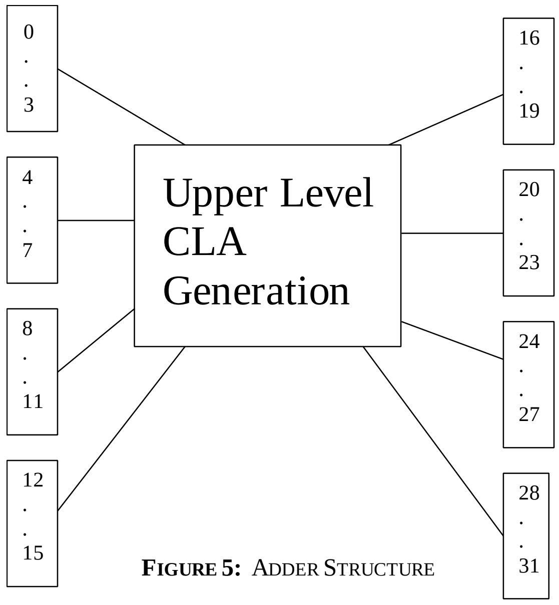

FIGURE 5: ADDER STRUCTURE

ALU

The blocks in the ALU are mostly combinational logic, and energy is dissipated when the inputs to the ALU change, switching the nodes of the ALU. So, the main change to the structure of the ALU is to allow it to restrict its computations to only the lower 16 bits of the data word. This will keep signals from propagating to the higher bits in the ALU and causing unneeded transitions.

The most basic operations of the ALU are the logic blocks. To perform, for example, an AND operation, each pair of bits of the 2 input data words must simply pass through an AND gate. If this is a 16-bit data word, then the upper 16 bits of each input should not be changed. This is accomplished by not allowing the outputs of the pipeline registers preceding driving the ALU inputs to change. Under these circumstances, the ALU will take 50% less energy to perform a logical operation on a 16-bit data word than on a 32-bit data word. Taking the HWTE effect into account, our design thus provides an energy savings of 33%.

The major energy consumer in the ALU is the adder. A typical adder structure is pictured in Figure 5. Each nibble of 4 bits enters a full 4-bit carry-lookahead (CLA) adder structure, which is the largest practical size. The other part of the adder is an upper-level carry generation structure. It takes inputs from the 4-bit CLAs and also provides the 4-bits CLAs with their carry inputs. By again not allowing the inputs to transition, the upper four 4-bit CLAs will not make any transitions, giving a linear energy savings. It is much more complex to generate the higher-bit carries than the lower-bit carries, and by not generating these, there will be a greater-than-linear savings in energy per add. For a typical adder design, a 16-bit adds takes 58% less energy than a 32-bit add, and this provides us with an energy savings of 39%.

A multiplier structure is depicted in Figure 6. It performs a 32-bit multiply in 32 cycles, or with modified control logic, a 16-bit multiply in 16 cycles. The complexity of a multiply grows with the number of bits squared, and this implies that the energy required for each multiply will grow at the same rate. For an N-bit multiply, the multiplier must perform N adds, N shifts, and N register writes, each of N bits. So, the energy required is, in fact proportional to N^2, and for a typical multiplier implementation such as in Figure 6, a 16-bit multiply takes 77% less energy than a 32-bit multiply.

A block diagram of a barrel shifter is shown in Figure 7. The barrel shifter is modified to be able to operate as a 32-bit shifter, or as a strictly 16-bit shifter. In 16-bit mode, the data inputs and outputs are not driven to the upper 32 bits of the shift structure. Whereas in 32-bit mode, even if the input data is 16-bit (i.e. its upper 16 bits are zero), the 16 active input bits must be driven across all 32 shift lines. In calculating the energy savings, one must recognize that an N-bit shifter has N^2 transistors in its structure (one pass transistor from every input to every output). So, to perform a 16–bit shift on a 32-bit shifter, half of the 32x32 structure is used, shown as blocks C and D in Figure 7. However, to perform a 16-bit shift on a 16-bit shifter, the entire 16x16 structure is used, i.e. only block C in Figure 7 is used. Therefore, a 16-bit shift, regardless of the input data, takes 50% less energy than the same shift on a structure that is capable of a 32-bit shift.

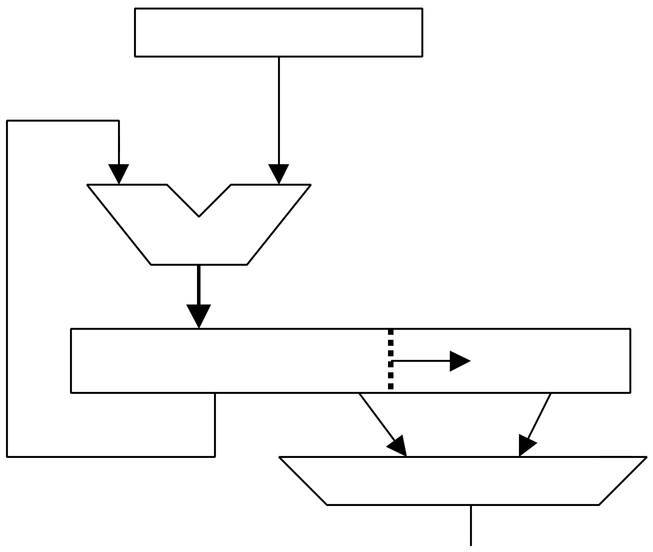

FIGURE 6: MULTIPLIER STRUCTURE

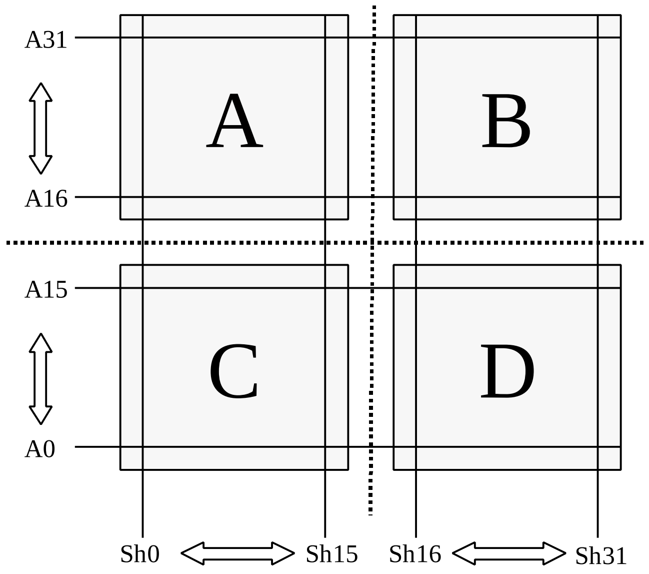

FIGURE 7: BARREL SHIFTER

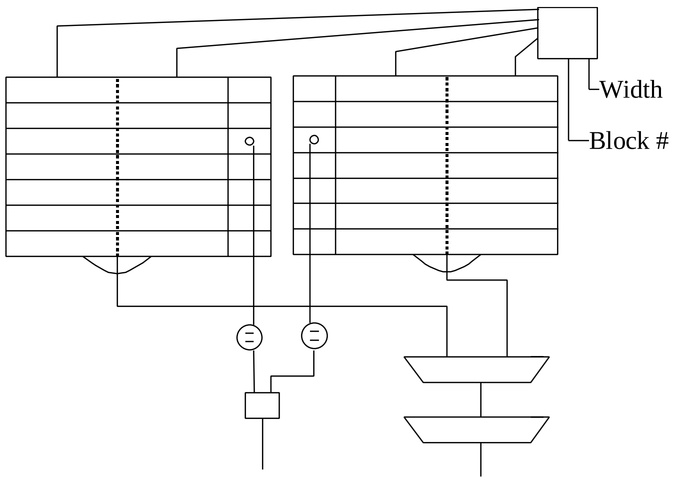

FIGURE 8: DATA CACHE STRUCTURE

Data Cache

The data cache is a large, highly regular memory structure that can easily be segmented into blocks and sub-blocks. It is this fact that will allow us to save a significant amount of energy in the cache, if we are aware that we are only working with a 16-bit data word.

A block diagram of the cache is shown in Figure 8. The memory itself is assumed to be made up of standard 6-T SRAM cells, as described in [2]. The word line is now segmented into 8 lines, 4 to each block, so that each 16-bit word in the cache line has its own independent word line. In the event of a 32-bit access, there are two word lines that must be driven to select the correct column in the cache. However, if there is a 16-bit access, then only one word line needs to be driven. In order to determine which word lines to drive, some logic is added in front of the cache. This logic only needs to examine the width flag bit, and if it is a short data word, then the least significant bit of the cache index is used to select the high or low word of the block, in each set. Finally, an additional multiplexer is added at the output of cache to select the high or low 16-bit data word out of a 32-bit block, based again on the least significant bit of the cache index. However, this multiplexer is only enabled if the cache access is to a short data word, and is set to simply pass the whole 32-bit block if it is a long data word access.

The energy savings comes from segmenting the word line, and only having to drive half the total word line capacitance per block. Each of the 8 word lines controls an entire column of the cache, and therefore runs the entire length of the cache block, giving it a huge capacitance. These energy savings greatly outweigh the extra energy that needed for the extra control structures. Therefore, the total energy savings for the cache is approximately 50%.

Clock Energy

Energy savings along the clock line can be calculated by considering which pipeline registers are gated and when. This approach ignores other fanouts of the clock, such as dynamic logic blocks, but it is assumed that in low power design, the use of dynamic logic would be kept to a minimum. In our design, we clock:

- 138 registers every cycle,
- 64 registers only on 32-bit operations,
- 16 registers only on instructions with an immediate operand
- 32 registers only on load instructions
- 32 registers only on store instructions

A conventional processor would clock all 282 of these registers on every cycle. Using the assumed instruction mix for a 16-bit application, we calculate that our design would average 173 clocked registers per cycle. Taking into account the overhead of generating the gated clocks, our design achieves a clock line energy savings of 35%.

Results Summary

The results are summarized in Table 2.

IV. CONCLUSION

The primary drawback to our design is the change to the standard MIPS ISA to signal which instructions can be dealt with as 16-bit operations. The changes to the datapath elements and the control logic of various blocks present minimal design time and effort costs, and nearly negligible increases in silicon area. However, for these very modest costs, there are substantial power savings under certain operating conditions. Furthermore, the changes that are proposed can simply be added to virtually any existing architecture to achieve a 20%-25% savings in energy, depending on the relative power consumption of the major components of the processor.

APPENDIX A: RELATED RESEARCH

There is little other research that deals with the question of utilizing smaller data words to reduce power consumption on multi-media applications. The most closely related research is on adding

Core Block	Percent of Total Power	Power Reduction
ALU	34%	33%
I-Decode	23%	0%
Reg. File	13%	33%
Clock	10%	35%
Shifter	11%	50%
Pipeline Reg.	9%	26%
Savings in Core:		27%

Cache	Percent of Total Power	Power Reduction
I-Cache	60%	0%
D-Cache	40%	50%
Savings in Cache:		20%

Sub-System	Percent of System Power	Power Reduction
Cache	66%	20%
Core	33%	27%
Total Power Savings:		22%

TABLE 2: RESULTS SUMMARY

SIMD instructions, not unlike the MMX extensions on the Pentium, to speed up certain audio and visual applications. These applications fall more in the realm of DSP-like computations, with small, high-speed kernels that do most of the computations. These types of machines have special instructions that can turn the N-bit data path into m datapaths, each N/m bits wide. This is in the interest of higher-speed media processing.

There are no power savings with these schemes, and in fact, it may take more energy to perform these types of operations than the full N-bit operation. This is because every bit of the data word is driven to a new value on every cycle. The emphasis in these machines is on performance, not on power consumption. So, instructions are executed wastefully in the hopes of obtaining higher performance. Also, since the speedup may be gained from an off-chip co-processor, there is a substantial power penalty that is incurred for communicating off-chip.

Appendix B: Project History

Our project, as originally proposed, was "dynamic code optimization based on run-time environment". It may amuse the reader to learn the long and strange chain of events through which the proposed project morphed into the current one.

The original project involved dynamically modifying program code through periodic software-based recompilation in order to optimize for speed. Optimization would be possible because various parameters which were not known at the original compile time would be known and constant at this dynamic recompilation phase. Unfortunately, we could not think of a compelling situation where this technique would be especially useful. We moved on to looking at more low level optimizations that would be better suited to hardware implementation, such as dynamically swapping double precision instructions for single precision when full precision was not vital. That wasn't feasible, but it got us thinking about the precision versus speed tradeoff. Various ideas in this realm included a floating point unit where computation latency was variable and depended on the amount of precision desired, an A/D converter which could be read more quickly but with fewer bits of precision, or being able to dynamically skip non-essential instructions (leading to a less precise result) if performance demands were high enough. These techniques would present the processor with a data word where only some of the bits were valid, and the rest were garbage. So we started exploring how the processor itself could use the reduced valid word width to speed up calculations. This led to the concept of variable word width computation for speed. Unfortunately, we were having difficulty with both how and why one would do such a thing. Dynamically optimizing for speed in embedded applications makes little sense, because performance gains beyond the real-time constraints (which must be guaranteed anyway) are usually unnecessary.

But when we entertained the idea of using power as our optimization criteria, everything fell into place. The "why" was obvious and the "how" was feasible. We had a project.

References

[1] J. Hennessy, D. Patterson, *Computer Architecture, a Quantitative Approach.* Morgan Kaufman Publishers, Inc, San Francisco, 1996

[2] J. Rabaey, Digital Integrated Circuits (First Edition) Prentice Hall, Upper Saddle River, New Jersey, 1996

[3] J. Rabaey, Digital Integrated Circuits (Second Edition) 1999

[4] Piguet, Architectural and Circuit Design for Portable Electronic Systems. Advanced Engineering Course from the Electronics Laboratories of the Ecole Polytechnique Federale de Lausanne, Switzerland

[5] S. Goldstein, et al., "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," *Proceedings of the 26th International Symposium on Computer Architecture*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1999. p.28-39

[6] M. Wirthlin, B. Hutchings, "A Dynamic Instruction Set Computer," *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines,* Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1995. p.99-107

[7] J. Montanaro, et al., "A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor," IEEE JSSC, vol. 31, no. 11, November 1996