Very Low Power Pipelines using Significance Compression

Ramon Canal†, Antonio González† and James E. Smith‡

†Departament d´Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona
{rcanal,antonio}@ac.upc.es

‡Department of Electrical and Computing Eng.
University of Wisconsin-Madison
jes@ece.wisc.edu

Abstract

Data, addresses, and instructions are compressed by maintaining only significant bytes with two or three extension bits *appended to indicate the significant byte positions. This* significance compression *method is integrated into a 5-stage pipeline, with the extension bits flowing down the pipeline to enable pipeline operations only for the significant bytes. Consequently register, logic, and cache activity (and dynamic power) are substantially reduced.*

An initial trace-driven study shows reduction in activity of approximately 30-40% for each pipeline stage. Several pipeline organizations are studied. A byte serial pipeline is the simplest implementation, but suffers a CPI (cycles per instruction) increase of 79% compared with a conventional 32-bit pipeline. Widening certain pipeline stages in order to balance processing bandwidth leads to an implementation with a CPI 24% higher than the baseline 32-bit design. Finally, full-width pipeline stages with operand gating achieve a CPI within 2-6% of the baseline 32-bit pipeline.

1. Introduction

There are many microprocessor applications, typically battery-powered embedded applications, where energy consumption is the most critical design constraint. In these applications, where performance is less of a concern, relatively simple RISC-like pipelines are often used [8][10]. A variety of circuit and microarchitecture techniques are employed to conserve energy when the processor is operating, and power-down "sleep" modes are invoked when the processor is not in use. In current CMOS technology, most energy consumption occurs when transistor switching or memory access activity takes place [3]. Therefore, in this paper we focus on reducing dynamic energy consumption. Dynamic energy consumption is proportional to the switching activity, as well as the load capacitance and the square of the supply voltage. Thus, an important energy conservation technique is to reduce switching activity by "gating off" portions of logic and memory that are not being used.

Recently [1] it was proposed that rather than basing logic gating decisions entirely on *operation* types, certain *operand* values could also be used to gate off portions of execution units. In particular, arithmetic involving short-precision operands only needs to be performed on the (relatively few) numerically significant bits. Operands containing insignificant bits (typically leading zeros or ones) can yield simpler computations or can be used to avoid computations altogether. Note that this operand-based gating targets a different source of energy consumption than operation-based gating, and both operation- and operand-based gating techniques can be used concurrently.

We generalize the notion of operand gating to all stages of the pipeline as a way of reducing switching activity and hence, dynamic energy consumption. The key principle is the use of a small number of *extension bits* appended to all data and instructions residing in the caches, registers, and functional units. In Fig. 1, the extension bits are shown along the bottom of a basic pipeline. These bits correspond to portions of the datapath, and they flow through the pipeline to gate-off unneeded energy-consuming activity at each stage, including pipeline latching activity. New extension bit values are generated only when there is a cache line filled from main memory (although they could also be maintained in memory) and when new data values are produced via the ALU. The points where extension bits are generated are indicated in Fig. 1 by circled "G"s.

For the instruction caches, extension bits allow a simple form of compression targeted at reducing instruction fetch activity, rather than reducing the number of bits in the program's footprint. For other datapath elements, they enable a form of compression where memory structures actively load and store only useful (significant) operand bytes. For arithmetic and logical operations, the extension bits enable operand gating techniques similar to those proposed in [1].

Given that only significant bytes require datapath operations and storage, pipeline hardware can be simplified by using byte-serial implementations, where the datapath width may be as narrow as one byte, and a pipeline stage is used repeatedly for the required number of significant bytes. Although there are many alternative implementations with different degrees of parallelism, they all have some serialization in the pipeline. In particular, low-order byte(s) and extension bits are first accessed and/or operated on; then additional bytes may be accessed and/or operated on if necessary. We describe and evaluate several pipeline implementations of this type.

When compared with a conventional 32-bit pipeline, significance compression can reduce activity by 30-40% for each pipeline stage. The simplest implementation (byte-serial) suffers a CPI (cycles per instruction) increase of 79% but wider pipelines incur a performance loss as little as 2-6%.

The paper is organized as follows. Section 2 presents several techniques to reduce the activity at each stage of the pipeline. The experimental framework is described in section 3. Sections 4, 5, and 6 present implementations with differing levels of complexity and performance. Finally, section 7 contains a summary and conclusions.

2. Techniques for Reducing Activity Levels

In this section, we develop methods for reducing memory and logic activity for each pipeline stage. Because activity in

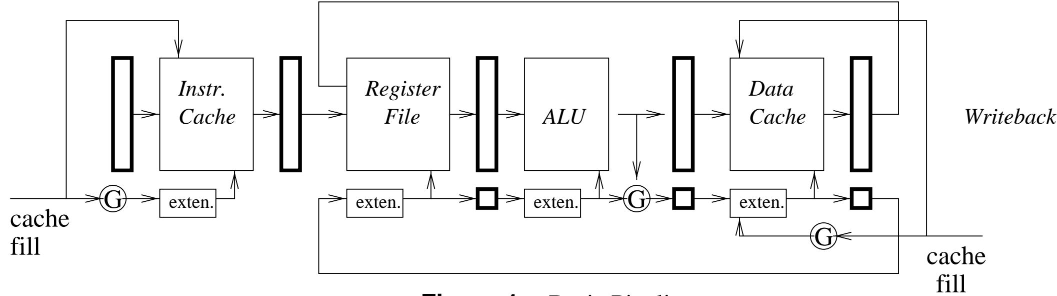

Figure 1: Basic Pipeline

the simple pipeline depends primarily on data values and instructions, we first undertake a trace-driven study to determine the required activity for each of the major pipeline operations. Then, in later sections, we propose and study pipelined implementations that come close to achieving the minimum "required" activity levels.

This work is based on a simple 5-stage pipeline with in-order issue as is often used for low power embedded applications. We consider the 32-bit MIPS instruction set architecture (ISA) and focus on integer instructions and benchmarks -- commonly used in the low power domain.

2. 1. Data Representation

The basic technique for representing data is to tightly compress data bits that do not hold significant data. For example, a small two's complement integer has only a few numerically significant low-order bits and a number of numerically insignificant higher order bits (all zeros or all ones).

In principle, one could consider significance at bit-level granularity, i.e. store and operate on exactly the numerically significant bits and no more. However, implementations are likely to be simpler and more efficient overall if a coarser granularity is used. Consequently, we primarily consider byte granularities and focus on the significant bytes rather than bits. Byte granularity is rather arbitrarily chosen, but it seems to be a good compromise of implementation complexity and activity savings. For comparison we also provide some results for halfword (16-bit) granularities. In general, one could consider non-power-of-two bit sequences and dividing words into sequences of different lengths, but this remains for future study. Because the lowest order data byte is very often significant, we will always represent and operate on the low order byte. Then we will use a very small number of bits (2 or 3) to indicate the significance of the other 3 bytes (of a word).

A simple encoding is to add two extra *extension bits* to encode the total number of bytes that are merely sign extensions. For example, the 32-bit number 00 00 00 04 (in hexadecimal) can be encoded as - - - 04 : 11. This is a mixed hexadecimal/binary notation that uses hexadecimal for significant (represented) bytes, a dash for the insignificant (non-represented) bytes, and a binary pattern after the colon for the values of the extension bits. In the above example, the only significant byte is 04 with three sign extension bytes, so the extension bits encode a binary three. This simple method also works for two's complement negative numbers if it is assumed that the high order significant bit of the most significant data byte is extended. For example, the number FF FF F5 04 can be represented as - - F5 04: 10. I.e. it has two significant bytes, and the most significant bit of these two bytes is extended to fill out the full 32-bit number. This encoding works well and has an overhead of two bits per 32-bit word (about 6 percent).

After inspecting commonly occurring data/address patterns, it is apparent that there are other, easily compressible values. In these cases there are some "internal" bytes that are all zeros or all ones, and these bytes are in a sense insignificant (slightly abusing the meaning of "significance"). An important case occurs for memory addresses in upper memory. These addresses often have nonzero upper bits, nonzero lower bits, but zero bits in between. For example, the data segment base of our experimental framework (see section 3) is set at address 10 00 00 00, thus a variable may be located at address 10 00 00 09.

To handle these cases, we propose a scheme with three extension bits (approx. 9% overhead). In this scheme, the extension bits apply on a per-byte basis. Each extension bit corresponds to one of the upper three data bytes (as before, the least significant byte is always fully represented). If an extension bit is set to one, it indicates that the previous byte position is sign extended; if the extension bit is zero, it indicates the corresponding byte is significant. Consequently, the earlier example 10 00 00 09 is represented as 10 - - 09: 011. As a more complex example, FF E7 00 04 is represented as - E7 - 04 : 101

The three-bit extension scheme allows for eight different patterns of significant/insignificant bytes (assuming the low order byte is always significant). We performed a study with the Mediabench benchmarks [6] to determine the relative frequency of occurrence of each (see section 3 for more details of the experimental framework). Table 1 lists the results. In the table, the notation "sess" indicates that the first, third, and fourth bytes are all significant and that the second byte is the sign extension of the third. The data show that the four most common cases include about 94% of operand values, and these four cases are the same as those that can be encoded with the two extension bit format described earlier. This suggests a trade-off between the two- and three-bit schemes. The former reduces the overhead from 9% to 6% whereas the latter may potentially reduce activity for about 6% more operands. We chose to study the 3-bit scheme, although one could reasonably argue that the 2-bit scheme is better due to simplicity and overhead advantages; in any case, the performance results are likely to be very similar for both schemes.

Table 1 also indicates the high level of compression that is possible. About 60 percent of the data values used in the

Table 1: Frequency of significant byte patterns

Cases	Register values	Ld/St values	Overall values	Acc.
eees	61.8	45.7	61.0	61.0
eess	13.3	20.3	13.6	74. 6
ssss	12.3	17. 6	12.6	87.2
esss	7.1	12.2	7.4	94.6
sses	1.8	0.3	1.8	96.4
sess	1.6	2.9	1.6	97.9
eses	1.4	0.8	1.4	99.2
sees	0.8	0.3	0.8	100

pipeline have only one significant byte and 75 percent have at most two. In the following subsections, we consider each stage of the instruction pipeline and describe ways that activity can be reduced via appended extension bits.

2. 2. PC Increment

Incrementing the PC is at the very beginning of the pipeline. When incrementing the program counter, we do not literally append extension bits to the operands. One of the operands is always +1 (the PC is word resolution), so it is known to have only one significant bit. The PC, on the other hand, is held to full 30-bit precision. The PC increment is performed byte-serially to reduce activity. In particular, we first increment only the low order byte. If a carry out is produced, the next byte is incremented on the next cycle, etc. If a carry out is not produced at any stage, no additional byte additions need to be done.

This method very often saves adder and PC latching activity for higher order bytes, but it can lead to some performance loss in the uncommon cases when there is a carry beyond the low order byte, and instruction fetch is temporarily stalled while additional byte additions are performed. A brief analysis sheds some light on this trade-off. In general, one can consider a block-serial implementation where the block size is not necessarily a byte. The size of the block determines the performance and the activity savings. Performance is maximized by the biggest block size (i.e. 30 bits), but the activity savings are null. On the other hand, a smaller block may have a slightly lower performance but may produce significant activity savings.

If the block size is N bits, and we assume that at a random point in time all instruction addresses have the same probability, then we can calculate the probability that i stages (each of size N) are required to compute PC+1, and based on this probability, we can then compute the average number of bits operated on (Activity) and the average number of cycles to compute a PC (Latency).

Table 2 shows the Activity and Latency statistics for values of N ranging from 1 to 8. Higher values of N are not interesting because they hardly improve performance, and activity increases significantly. Minimum activity is achieved for $N=1$, but this incurs a 1-cycle penalty per instruction on average (as expected). $N=5$ may be a good trade-off because activity is reduced by 83%, and performance is degraded by just 3%. Finally, $N=8$ provides negligible degradation in performance with an activity reduction of 73.2%. In section 2.9 we validate these analytical figures with an empirical analysis. As indicated above, we assume a block size of 8 bits for the PC increment.

Table 2: Activity and latency estimates for PC updating

number of bits per block	Activity (bits operated on)	Latency (cycles)
1	2.0000	2.0000
2	2,6667	1.3333
3	3.4286	1.1429
4	4.2667	1.0667
5	5.1613	1.0323
6	6.0952	1.0159
7	7.0551	1.0079
8	8.0314	1.0039

2. 3. Instruction Cache

To save instruction cache activity, instruction words are stored in a permuted form. The goal is to reduce the number of instruction bytes that have to be read, written, and latched. This objective is somewhat related to the more common instruction compression techniques [4,5,12,13,18] that attempt to store more instructions in a given amount of memory. In our case, each instruction is still allocated a full word in the instruction cache. However, not all bits have to be read/written/latched each time an instruction is placed in the cache or is fetched. Simple permutation-based compression schemes are important because the energy consumption of the decompression task should not offset the benefits of reducing the number of bits to be processed. Permutation methods of this type are likely to be specific to the ISA, and we consider methods that work well for the MIPS ISA. While the exact methods may not extend entirely to other ISAs, similar methods are likely to be applicable –at least for RISC ISAs.

Although we considered a number of methods, two basic schemes seem to work well for the MIPS ISA and probably provide a significant majority of the benefit that can be achieved. First of all, we observe that the MIPS ISA very often uses one of two formats[1] [11]:

- R-format: A 6-bit opcode, three 5-bit register fields, a shift amount field, and a 6-bit function code.
- I-format: A 6-bit opcode, two 5-bit register fields, and a 16-bit immediate value.

In the R-format, the number of significant instruction bits can frequently be reduced to three bytes by recoding the six-bit function field so that the most common eight cases use three bits of the field with zeros in the other three bits. For these eight common cases, only three instruction bytes must be fetched and latched. In the other less common cases, all four instruction bytes must be fetched. Shifts that use the shift amount field do not use the first register field (*rs*), so the fields can be permuted by moving the shift amount (*shamt*) into what is normally the *rs* field.

The permutation for R-format consists of shuffling bits in a minor way and re-encoding the function bits. Figs. 2a and 2b show the permutations for the R-format instructions. The function field is split into two 3-bit fields, f1 and f2, as noted above. To determine which function re-encoding to use, we first traced the Mediabench benchmarks and counted the dynamic frequency of each of the function codes. The results are in Table 3. Thus, the most common eight function codes are recoded to 6-bit encodings, where the last three

[1] There is a third format (J-format), but it only accounts for 2.2% of the executed instructions in the Mediabench.

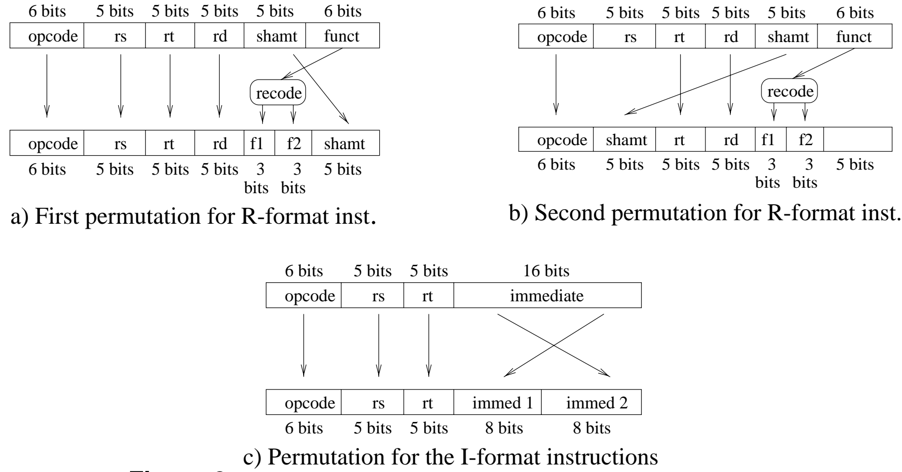

Figure 2: Permutations for the different instruction formats

bits are all zeros (and do not have to be fetched). All the other function code patterns are mapped to the remaining six bit patterns. From the table we see that 86.7% of all the R-format instructions require three bytes when modified in this manner.

For the I-format, we simply note that often eight or fewer immediate bits are actually significant, and in these cases three instruction bytes are again adequate. Fig. 2c shows the permutation for the I-format instructions. For I-format instructions we also traced the benchmarks and determined the sizes of the immediate values. It was found that 59.1% of all instructions use immediate values and 80% of these immediates require only eight bits.

Although there are a few cases where it can be done, we do not attempt to reduce the number of fetched instruction bytes to fewer than three. Consequently, we add a single "extension" bit to the instruction word portion of the instruction cache. This bit indicates whether three or four bytes should be fetched and latched. Note that only one bit is used and it serves multiple purposes depending on the actual 6-bit opcode. For typical R-format opcodes it indicates that the low order three function bits (field *f2*) are zeros. For the shift amount R-format opcodes, it also moves the *shamt* field, and for I-format opcodes it indicates an 8-bit immediate.

Overall, in the Mediabench suite a total of 36.9% of instructions are R-format that use the function field; 4.1% are R-format but the function field is not used; 56.9% are I-format, and 2.2% are J-format. Combining this with the immediate and function code frequency statistics, the average number of bytes fetched and latched per instruction is 3.17 bytes (3.29 if we include the extension bit). This represents a savings of about 20% (at an overhead of 3% for the extra bit per word). There is also additional overhead during instruction cache fill for permuting/modifying the instruction bits, but this is a relatively small amount of additional activity, assuming a reasonable instruction cache miss rate. Finally, note that the order of the rearranged instruction bytes is chosen so that the bytes needed earlier in the pipeline are toward the most significant end. This enables better performance for implementations (to be given later) that read instruction bytes serially. For example, after an implementation fetches the first two bytes, there is enough information to perform the initial opcode decode and register read operations. The other bytes give the immediate bits, a result register field, and/or ALU function bits that are not needed until later in the pipeline

Table 3: Dynamic frequency of function codes

Opcode	Percentage	Cumulative
ADDU	36.0	36.0
SLL	16.2	52.2
SRA	9.1	61.3
SLT	8.2	69.5
SUBU	8.2	77.7
SLTU	3.3	81.0
XOR	3.1	84.1
MFLO	2.7	86.8
Others	2.5 .. 0.0	100

2. 4. Register File Access

For the register file, extension bits as described in Section 2.1 are used. When the register file is accessed, first the low order data byte and the extension bits are read. Depending on the values of the extension bits, additional register bytes may be read during subsequent clock cycle(s). In a study of the Mediabench suite described below, we determined that the extension bits result in large register file activity savings. On average, the number of bits that are read is reduced by 47%.

To implement the single-bank, 32-bit register file of the baseline configuration and each of the 8-bit register banks required by the pipelines proposed in this work, different layouts can be used. In particular, the physical arrangement of the data array of each bank has a significant impact on the performance of the register file. Splitting the data array into multiple arrays, either horizontally or vertically, or widening the number of bits per word line has a significant impact on the access time as shown by Wada, Rajan and Przybylski [17], as well as power consumption. The layout that minimizes access time may not be optimal with respect to power consumption. Computing the optimal layout in terms of power consumption or finding the best trade-off between access time and consumption is an interesting work but it is beyond the scope of this paper. In the following discussions, we assume that each bank is implemented through a single array (i.e., 32 word lines of 32 bits each for the 32-bit base-

line configuration, and 32 word lines of 8 bits each for the proposed pipelines).

Under these assumptions, note that even in the worst case when all 32 bits are required, the multiple access do not necessarily increase energy consumption. The word line consumption of each single access is reduced by a factor of about four, since every bank is about one fourth the width and thus, word lines are about one fourth as long. Bit line consumption is reduced by about four, since the number of bit lines in each bank is reduced by a factor of four. Sense amplifier consumption is also reduced by a factor of four for each access, since the number of sense amplifiers matches the number of bit lines. Thus, four accesses result in approximately the same word line, bit line and sense amplifier energy consumption as the 32-bit bank file.

2. 5. ALU Operations

ALU operations are performed using only the numerically significant register bytes and the extension bits as input operands. The ALU produces significant result bytes as well as the extension bits that go with them.

ALU operations are performed in a byte-serial fashion. Because additions/subtractions, memory instructions, and branches all require an addition, and they collectively account for 70.7% of the executed instructions in the Mediabench suite, this operation is the most critical one to be implemented efficiently. For each byte position, there are three major cases, depending on which of the operands have significant byte(s) in the position being added.

- Case 1: Both bytes are significant. In this case, the byte addition must be performed.
- Case 2: Only one of the operands has a significant byte. If the non-significant byte is zeros (ones) and the carry-in from the preceding byte is zero (one), the result byte will be equal to the significant byte. If the non-significant byte is zeros (ones) and the carry-in is one (zero), the result byte is the significant byte plus one (minus one). In all these cases one could simplify logic, for example by bypassing the addition. However, we do not include these potential optimizations in activity statistics.
- Case 3: Neither of the operands has a significant byte in the position being added. Consider the addition of two bytes, $C_i=A_i+B_i$, where A_i and B_i are both sign extensions of their preceding bytes, A_{i-1} and B_{i-1}. There is a general rule with some exceptions. The general rule is that the result byte C_i is not significant, and the result is computed simply by setting the extension bits of the result because C_i will also be a sign extension of C_{i-1}. In the exceptional cases, the ALU must generate a full byte value. Table 4 lists all exceptions to the general rule.

To understand the exceptions to the general rule of case 3, consider the example where A_{i-1}=00000001, B_{i-1} =01111111; A_i and B_i are both sign extensions (i.e. they are equal to zero). Then the addition of A_i+B_i will obviously be zero, but because byte C_{i-1} has a one in its most significant bit, C_i is not the sign extension of C_{i-1}. In this case, the processor has to generate the full byte value, although the addition is not actually necessary.

Finally, note that in some cases a result byte may not be significant although the two source operand bytes are significant (e.g. 3 + -3 = 0). To handle these cases, there is simple logic that examines each result byte and generates extension

Table 4: Cases in which byte C_i has to be generated

Values of A_{i-1} and B_{i-1} (the order is not significant)		Extra conditions
00xxxxxx	01xxxxxx	5th bit produces carry
01xxxxxx	01xxxxxx	-
11xxxxxx	10xxxxxx	5th bit produces carry
10xxxxxx	10xxxxxx	-
00xxxxxx	11xxxxxx	5th bit produces carry
01xxxxxx	10xxxxxx	5th bit produces carry

bits accordingly. This logic basically checks whether all bits of a byte and the most significant bit of the previous byte have the same value. It then sets the extension bits for the result accordingly.

Another common ALU operation that can be optimized is the comparison, which represents the 6.9% of the instructions in our benchmarks. In this case, the normal byte processing order can be reversed, with the computation starting at the most significant byte and finishing with the least significant one. However, as soon as the two compared bytes are different, no more bytes must be computed, even for comparisons of the type greater-than or less-than.

This reversal of access order can be implemented with different levels of complexity depending on the particular processor design. For instance, for the byte-serial implementation described in section 4, the reversal is easily implemented since this design has a single byte-wide register file and a byte-wide ALU. In other cases such as the byte-parallel skewed implementation described in section 6, the order reversal is more complex and may require an additional register port for avoiding structural hazards.

Finally, bit-wise logical operations, which represent 4.2% of the instructions in our benchmarks, can also be byte-pipelined. In this case, whenever two bytes are sign extensions, the result will also be a sign extension. Note that other optimizations are feasible when just one of the operands is a sign extension, but we have not considered them. For instance, $A\ AND\ 0 = 0$, $A\ AND\ {-1} = A$, etc. As shown below in Section 2. 9, extension bits result in an average reduction of the ALU activity of 33% for Mediabench.

2. 6. Data Cache Operation

The data cache holds data in a manner similar to the register file. I.e. extension bits are appended to each data word and only the bytes containing significant data are read and written. The address bytes may also be formed sequentially, beginning with the low order byte. This means that the cache index will be computed before the tag bits, and that the tag bits may be formed as part of multiple byte additions.

Consequently, the tag comparison may be done in two sections as the tag bits become available. If the lower order tag bits do not match the cache tags, then an "early miss" can be signaled, and the higher order address and cache tag bits do not have to be formed and compared, resulting in reduced activity. However, because the miss rate is often relatively low, the activity saving is likely to be insignificant.

There is similar activity for cache writes; the extension bits for the store data are read from the register file and written alongside the data. For cache fills, the extension bits must be generated as data is brought from memory

Table 5: Activity reduction (%) for datapath operations (8 bit)

Benchmark	Fetch	RF read	RF writes	ALU	D-cache data	D-cache tag	PC increment	Latches
cjpeg	17.3	46.0	41.7	30.3	39.6	0.2	73.3	40.4
djpeg	19.4	41.1	37.1	22.7	39.9	0.1	73.3	36.6
epicdec	19.1	45.1	35.2	30.0	20.3	0.1	73.3	39.7
epicenc	20.6	42.5	46.7	42.0	0.9	0.0	73.3	46.7
g721dec	19.3	50.5	47.3	39.6	43.9	1.9	73.3	45.4
g721enc	19.3	50.7	46.9	39.7	44.5	1.0	73.3	45.4
gs	13.6	47.7	47.2	35.5	39.9	1.3	73.3	42.4
gsmdec	17.4	45.7	40.3	33.1	38.1	0.2	73.3	41.0
gsmenc	19.0	45.1	35.4	31.7	13.0	0.7	73.3	39.9
mesamipmap	16.2	37.5	30.5	19.4	12.4	3.6	73.3	32.9
mesaosdemo	15.6	33.7	26.9	15.2	19.0	0.5	73.3	30.2
mesatexgen	16.8	40.5	36.7	23.2	12.3	1.9	73.3	35.6
mpeg2dec	17.0	43.7	38.9	28.3	36.1	0.4	73.3	38.9
mpeg2enc	20.9	47.0	48.7	32.2	30.2	0.0	73.3	42.6
pgpdec	19.0	38.9	29.8	20.5	25.0	2.8	73.2	34.3
pgpenc	18.0	39.7	34.2	25.3	30.6	0.0	73.3	36.0
rasta	16.6	43.9	38.3	27.2	12.2	2.8	73.3	38.0
rawcaudio	20.0	71.7	69.2	67.6	57.4	0.0	73.3	67.4
rawdaudio	20.0	71.7	69.2	67.6	57.4	0.0	73.3	65.4
AVG	18.2	46.5	42.1	33.2	30.1	0.9	73.3	42.2

Table 6: Activity reduction (%) for datapath operations (16 bit)

Benchmark	Fetch	RF read	RF writes	ALU	D-cache data	D-cache tag	PC increment	Latches
AVG	18.2	35.9	30.3	22.1	23.4	0	46.7	34.9

(although the extension bit concept could also be maintained in main memory). We show below in Section 2. 9 that the above techniques reduce the activity on the data cache by 31% for the data array and 1% for the tag array.

2. 7. Register Write Back

During the register write-back stage, only bytes holding significant values have to be written into the register file. The extension bits also have to be stored. For ALU data, the bits are generated as described above in Section 2. 5. For memory data, the extension bits read from the data cache are used. We show below in Section 2. 9 that extension bits result in an average reduction of 42% in register file write activity.

2. 8. Pipeline latches

Significant energy is consumed in pipeline latches [16], not just the major datapath elements. The extension bits are used for gating the pipeline latches in the normal way [9,14]. Only the PC bytes that change require latch activity. Based on extension bits, only significant register, ALU and cache bits need to be latched. Hence, activity savings in the datapath elements is reflected directly in activity savings in the pipeline latches immediately following the datapath elements. Furthermore, clock signals can be gated at the byte level, threby reducing clock activity.

Latch activity depends on the particular implementation. The lowest latch activity is achieved by the implementations with fewer pipe stages. This is the case for instance of the byte-serial implementation described in section 4. In this case, we show in the next section that the latch activity can be reduced on average by 42%.

2. 9. Activity performance

To determine the activity savings for the techniques described above, we performed a trace driven simulation of the Mediabench [6]. Only byte activity indicated by the extension bits was performed. Table 5 provides the overall results for byte granularity, and for comparison, Table 6 contains average results for halfword granularity significance compression. The tables show percent activity savings.

The byte-serial PC increment operation saves 73% activity, because the great majority of the time, only the least significant byte is changed, as predicted by the analysis in Section 2.2. I-cache activity saving is 18%, and is quite uniform across all benchmarks. On average 47% of the Register read activity is saved, with individual benchmarks saving from 34% to 72%. ALU activity saving averages 33% (ranging from 15% to 68%) and data cache activity saves an average of 30% (ranging from 1% to 57%). The data cache activity is measured for data fills, reads and writes. The average saving on the data bank is 31% (ranging from 1% to 57%) whereas the saving for the tag bank is negligible. Register writeback saving is on average 42% (ranging from 30% to 69%). Finally, for implementations where the number of stages is not increased beyond the basic 5-stage pipeline, the latch activity is reduced by 42% on average and between 30% and 67% for individual benchmarks.

The 16-bit serial savings remain substantial (Table 6), but are somewhat less than the byte serial activity savings, as expected. The primary advantage of the 16-bit granularity is in implementation simplicity and in performance, as will be shown in the next section.

Holding and maintaining the extension bits adds an overhead of 9% when three bits are used, and the PC increment and fetch stages have much less overhead.

The bottom line is that the net overall activity savings (and therefore the overall energy savings) can be substantial. Major savings are possible in each of the pipeline stages. Finally, note that these results are for a 32 bit architecture; if

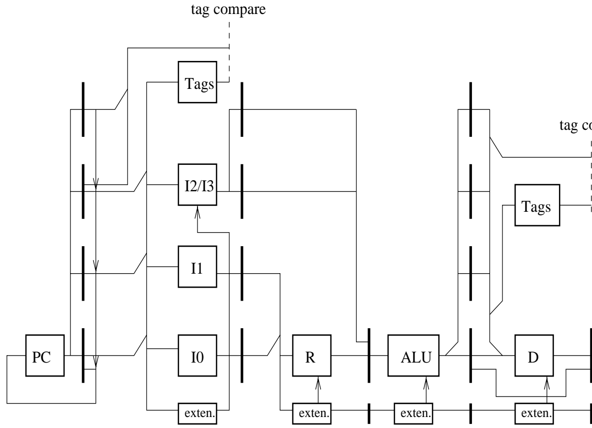

Figure 3: Byte-serial implementation

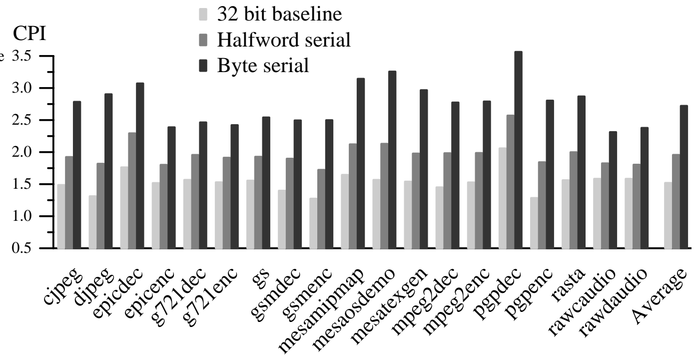

Figure 4: Performance of the byte-serial implementation

a 64-bit ISA were to be used (as in [1]), the savings will likely be much greater.

3. Experimental Framework

We developed a simulator for several proposed pipeline implementations using some components of the SimpleScalar toolset, primarily the instruction interpreter and the TLB and cache simulators. In all cases we assumed an in-order issue processor, with the following microarchitecture parameters:

- First level split instruction and data cache: 8 KB, direct-mapped, 32-byte line, 1-cycle hit time.
- Second level unified cache: 64 KB, 4-way., 32-byte line, 6-cycle hit, 30-cycle miss.
- I-TLB: 16 entries, 4-way, 1-cycle hit,30-cycle miss.
- D-TLB: 32 entries, 4-way, 1-cycle hit, 30-cycle miss.

The processor does not perform any type of branch prediction, thus every branch stalls the fetch stage until the branch is resolved in the ALU stage. This is in keeping with some very low power embedded processors, although the trend is toward implementing branch prediction. The implications of branch prediction will be the subject of future study.

We used the Mediabench benchmark suite [6], which were compiled with the gcc compiler with "`-O3 -finline-functions -funroll-loops`" optimization flags into a MIPS-like ISA. As a baseline for comparison we use a conventional 32-bit wide processor, with 5 pipeline stages: Instruction Fetch, Decode and Register Read, Execute, Memory, and Write Back.

4. Byte-Serial Implementation

Having established potential activity reductions that can be achieved (and therefore energy reductions), we now consider implementations that attempt to achieve these levels while providing good performance. Implementations will differ in total hardware resources although they may not necessarily differ in circuit activity.

First, we consider a simple *byte-serial* implementation that has a one byte wide data path. If more than one data/address byte is needed at a given stage, then that pipeline stage will be used sequentially for multiple cycles. While later sequential data bytes are being processed, however, earlier bytes can proceed up the pipeline. For example, if it is necessary to read 3 bytes from the register file, first the low order byte is read and passed on to the EX stage, then while the next byte is being accessed, the EX unit can perform on the first data byte and pass it to the data cache stage.

Fig. 3 shows the byte-serial implementation. In this microarchitecture there is a single register file bank (R), a single ALU, and a single data cache bank, all one-byte wide. Inter-stage latches are provided to store values on a byte basis and only the significant bytes are required to be latched. In addition, the extension bits must flow through the pipeline and a three bit latch is provided between some stages for this purpose. The ALU stage includes a special unit that operates on extension bits as described in Section 2. 5. There is one byte-wide PC increment unit that operates serially and three instruction cache banks that are accessed in the first stage along with the extension bit. Then, if the extension bit indicates that it is needed, the instruction remains in this stage for one more cycle while one of the banks is accessed again. Using a three byte wide instruction cache stage is a departure from the strictly byte serial implementation. This decision was made to avoid excessive stalls while reading instructions; otherwise, every instruction would incur at least two stall cycles because the minimum number of bytes per compressed instruction is three.

Fig. 4 shows the performance of the byte-serial implementation, expressed as cycles per instruction (CPI). For comparison, the CPI of a baseline 32-bit wide implementation is also shown. For most programs, the performance of the byte-serial implementation is significantly lower than that of the 32-bit processor. CPI is increased by 79% on average, although activity (and energy) is reduced by 30-40% for most of the pipeline functions (Table 5).

If the pipeline is widened to 16-bits, the average CPI becomes 1.96, which is just 29% higher than that of the byte-wide implementation, but the activity savings are lower (around 20-30% for most of the pipeline functions). Note that the relative performance of the pipelined schemes is quite uniform across all the benchmarks.

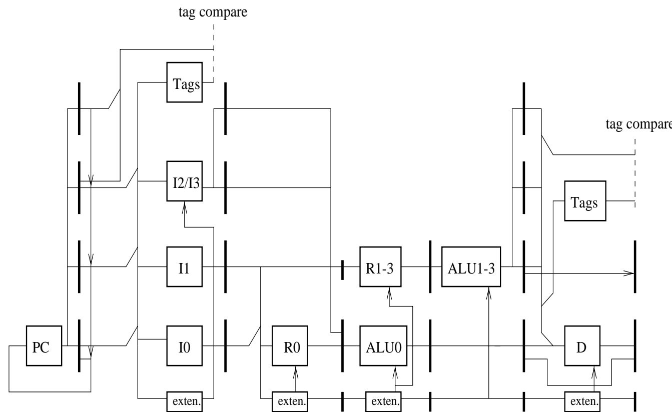

Figure 5: Byte semi-parallel implementation

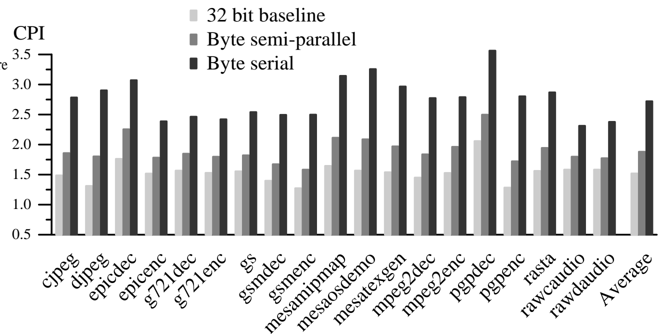

Figure 6: Performance of the byte semi-parallel microarchitecture

5. Semi-Parallel Implementations.

The byte-serial implementation achieves significant activity reduction, but at the cost of substantial performance losses with respect to the baseline 32-bit pipeline. For some applications, energy savings may be much more important than performance, and this may represent a good design point. There may be other applications, however, where performance is more important, and performance losses should be reduced. We now consider methods that retain low activity levels, but use additional hardware to improve performance.

The principle is to improve performance by adding additional byte-wide datapath elements at the various pipeline stages. For example, the register file can be constructed of two byte-wide files (rather than one) and produce a full data word in 2 cycles instead of 4. Similarly, multiple byte-wide ALUs can be used to increase throughput in the execute stage.

Adding these units does not necessarily increase circuit and memory access activity, however, because not all the units have to be enabled every cycle. For example, if a data item has only one significant byte, then a register access can be performed for one byte of a two byte wide register file, while the other byte is disabled. Similarly, if the source operands of an addition only have two significant bytes, these bytes will be operated in two of the ALUs while the others will be disabled.

Finally, the numbers of byte-wide units in each of pipeline stages do not have to be the same. That is, the number of byte ALUs or memories can be established to permit balanced processing bandwidths among the pipe stages. To determine how many parallel units and memories should be used, we first undertook a bottleneck study of the byte-serial implementation to see where the major stalls occur. We observed that in the byte-serial architecture the ALU is the most important bottleneck, 72% of the stalls were caused by structural hazards in the EX stage. Thus, increasing the bandwidth of the ALU stage is the most effective approach to increase performance. To quantify how much bandwidth is required in each stage, we did the following simple analysis.

Consider each of the major pipeline stages. First, the study in Section 2.3 shows that an instruction requires about 3.2 bytes to be fetched on average. The ALU operates on an average of 2.7 bytes, but since the maximum CPI is 1.5 (32-bit baseline processor), the activity of the ALU will not be higher than 2.7/1.5 = 1.8 bytes/cycle on average. Next, around one third of instructions access memory, and each access is 2.8 bytes wide on average. Thus, less than one byte per cycle is accessed on average. Based on this study, we determined that a good balance is achieved with an instruction cache three bytes wide, a register file and ALU 2 bytes wide, and data cache one byte wide.

An implementation for this configuration is shown in Fig. 5 and is referred to as *byte semi-parallel*. The instruction cache essentially contains three byte-wide banks and works as in the byte-serial implementation.

The register access stage is skewed with the low order byte being accessed first together with the extension bits. In the next stage the low order byte is operated on, and at the same time another register byte is read if needed according to the extension bits. If there is more than one additional byte the instruction uses this stage for multiple cycles. The next stage performs the ALU operation on the additional bytes and is used for as many cycles as the previous stage. The following stage performs the data cache access (if needed). It first reads/writes the low order byte, the tags, and the extension bits and, according to the latter, the instruction uses this stage sequentially for multiple cycles until all data are read/written. Finally, the last stage writes the result into the register file. It first writes the low order byte, the extension bits and one additional byte if needed. If more than one additional byte must be written, this stage is used for multiple cycles.

Fig. 6 shows the CPI of this microarchitecture along with that of the 32-bit baseline processor and the byte-serial implementation. On average, the CPI is 24% higher than the 32-bit baseline processor. We observe that the performance is much closer to the 32-bit implementation than the byte-serial implementation while all the activity savings are retained except for a few additional latches.

6. Fully Parallel Implementations

The above still loses some performance – bottlenecks cannot be perfectly balanced all the time because of bursty behavior that most programs exhibit. So, we consider pipelines with maximum (4 bytes) parallelism at each stage, and use oper-

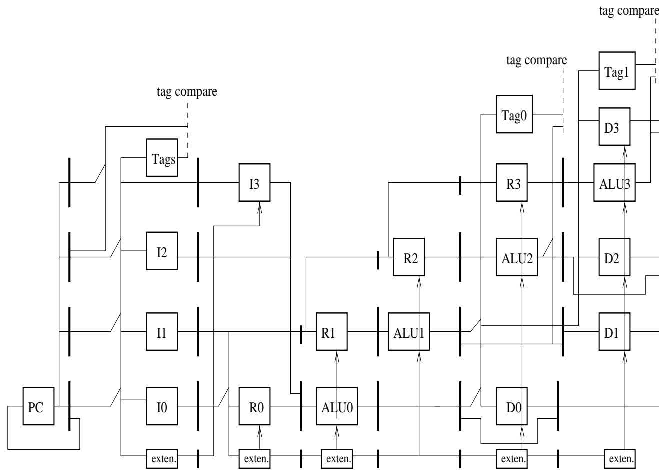

Figure 7: Byte-parallel skewed microarchitecture

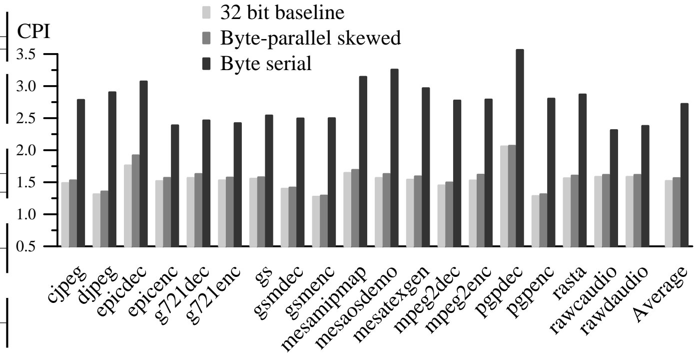

Figure 8: Performance of the byte-parallel skewed microarchitecutre

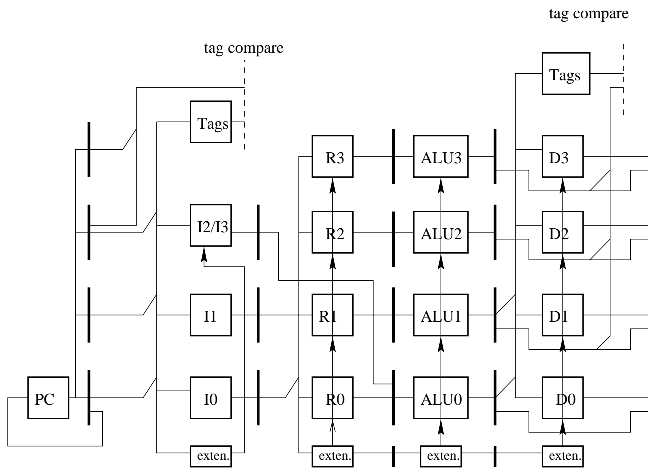

Figure 9: Byte-parallel compressed pipeline

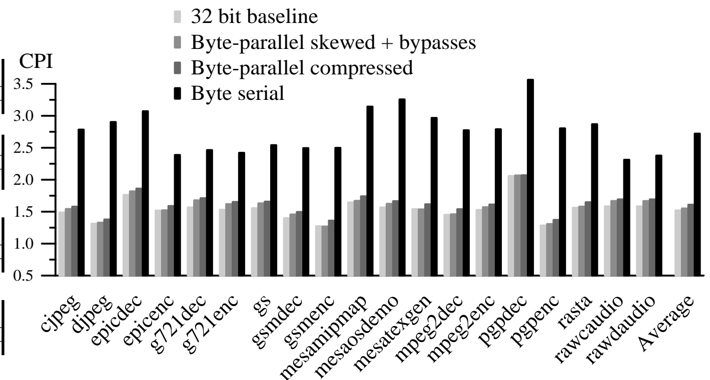

Figure 10: Performance of the byte-parallel compressed and skewed + bypasses microarchitecture

and gating to enable only those datapath bytes that are needed. This requires a skewing of stages in a similar way to the semi-parallel implementation described in the previous section. A block diagram of a portion of the microarchitecture, which is referred to as byte-parallel skewed, is depicted in Fig. 7.

This pipeline is optimized for the long data case, i.e. where the pipeline keeps flowing even if each operand is a full 4 bytes. No stage is used more than once (except for the PC computation in very few cases). Although the activity of the functional units is the same as that of the byte-pipelined and semi-parallel implementation, the longer pipeline of the byte-parallel skewed implementation implies more latch activity and more backward bypasses. The performance of this microarchitecture is shown in Fig. 8. We can observe that the CPI is very close to that of the 32-bit baseline processor for all programs in which case the byte serial implementation would be a very good design choice.

Another alternative is a "compressed" parallel pipeline implementation (see Fig. 9). In this case, the pipeline consists of the original 5 stages. Each instruction spends one cycle in the Ifetch stage to read 3 bytes and an additional one if a fourth byte is needed. Then it moves on to the second stage where it reads the low order byte and the extension bits. If more bytes are needed, the instruction spends one more cycle in the same stage to read all of them in parallel. Then the instruction moves on to the ALU stage where it executes in a single cycle, using only the functional units that operate on significant bytes. Then it moves on to the memory stage where it reads first the low order byte and the extension bits, and if needed, it spends an additional cycle to read all the remaining bytes. If it is a store, all the significant bytes along with the extension bits are written in a single cycle. Finally, all significant bytes and the extension bits are written into the register file in a single cycle.

This design works well for short data because the pipeline length is kept minimal and this reduces the branch penalty and the number of backward bypasses. Furthermore, functional unit and latch activity is kept minimal (equal to the byte-serial implementation). However, full-width (32-bit) data operations suffer stalls in some stages, which result in performance losses when compared with the full parallel implementation. Performance is shown in Fig. 10. The CPI increase compared with the 32-bit baseline processor is 6% on average, which is quite close to the performance of the byte parallel skewed configuration.

We can get the best of both (performance wise) by putting forwarding paths into the byte-parallel skewed pipeline. In this way, when a short operand is encountered, it can skip the stages where no operation is performed. This reduces the latch activity to the same level as that of the byte-serial implementation, and at the same time the effective pipeline length is shortened, which reduces the branch penalty. However, the number of backward bypasses is the same as that of the byte-parallel skewed implementation.

The performance of this architecture is also shown in Fig. 10. Now performance is very close to the baseline 32-bit processor (the CPI is only 2% higher on average) while the activity is reduced around 30-40% for most of the stages. A disadvantage, however, is that this design has rather complicated control and many data paths (for forwarding) -a more detailed analysis is required and will be a subject of future study.

7. Summary and Conclusions

The significant bytes of instructions, addresses, and data values essentially determine a minimal activity level that is required for executing a program. For a simple pipeline design, we showed that this level is typically 30-40% lower than for a conventional 32-bit wide pipeline. Every stage of the pipeline shows significant activity savings (and therefore energy savings).

We proposed a number of pipeline implementations that attempt to achieve these low activity levels while providing a reasonable level of performance. The byte-serial pipeline is very simple hardware-wise, but increases CPI by 79%. For some very low power applications, this may be an acceptable performance level, in which case the byte-serial implementation would be a very good design choice. We should also point out that the narrower data path may result in a faster clock, which will reduce performance loss, but this was not considered in this paper.

For higher performance, the pipeline stages can be widened. A rough analysis indicates that three bytes of instruction fetch, two bytes of register access and ALU, and one byte of data cache might provide a good balance of bandwidths. For this configuration, the CPI is 24% higher than that of the full width baseline design. Activities are still at their reduced levels, and this design may provide a very good design point for many very low power applications.

Finally, we considered designs with a four byte wide datapath at each stage. Operand gating is retained for reducing activity, but under ideal conditions throughput is no longer restricted. These designs can come very close in performance to the baseline 32-bit design while again retaining reduced activity levels. The disadvantage of these schemes is an increased latch activity, or additional forwarding paths or more complex control. We believe that these may be a very important class of implementations however, because of their high performance levels, and they deserve additional study.

Note also that different designs may imply a variation in the load capacitance, which also affects dynamic energy consumption. In particular, a narrower data-path may shorten some wires and thus reduce its capacitance. This paper focused on pointing out the potential of these architectures to reduce pipeline activity. The final quantification of energy requires a further detailed circuit-level analysis of the implementations.

Acknowledgements

This work was supported by the IBM University Partnership Program, the CICYT project TIC 98-0511. We would like to thank Jaime Moreno and George Cai for many insightful comments on this work. Ramon Canal would like to thank his fellow PBCs for their patience and precious help.

References

[1] D. Brooks and M. Martonosi, “Dynamically Exploiting Narrow Width Operands to Improve Porcessor Power and Performance”, in Proc. of 5th. Int. Symp. on High- Perf. Comp. Arch., 1999.

[2] D. Burger, T.M. Austin, S. Bennett, “Evaluating Future Microprocessors: The SimpleScalar Tool Set”, Technical Report CS-TR-96-1308, University of Wisconsin- Madison.

[3] G. Cai and C.H. Lim, “Architectural Level Power/ Performance Optimization and Dynamic Power Estimation”, in the Cool Chips tutorial of the 32nd Int. Symp. on Microarchitecture 1999.

[4] K.D. Kissell, “MIPS16: High-density MIPS for the Embedded Market”, SGI MIPS group, 1997.

[5] M. Kozuch and A. Wolfe, “Compression of Embedded Systems Programs”, in Proc. of the Int. Conf. on Computer Design, 1994

[6] C. Lee, M. Potkonjak and W. H. Mangione-Smith, “Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, in Proc. of the 30th Int. Symp. on Microarch., Dec. 1997, pp. 330-335.

[7] C.R Lefurgy, E.M Piccininni and Trevor N Mudge, “Evaluation of a High Performance Code Compression Method”, in Proc. of the 32nd Int. Symp. on Microarchitecture 1999.

[8] J. Montanaro and et al. “A 160-MHz, 32-b, 0.5 W CMOS RISC Microprocessor”, Digital Tech. J’rnal, v.9. Dec, 1997.

[9] E. Musoll, “Predicting the usefulness of a block result: a micro-architectural technique for high-performance low-power processors”, in Proc. of the 32nd Int. Symp. on Microarchitecture 1999.

[10] “PowerPC 405CR User Manual”, IBM/Motorola, 6/2000.

[11] C. Price, “MIPS IV Instruction Set”, MIPS Tech. Inc, 1995.

[12] J. Turley, “Thumb Squeezes Arm Code Size”, Microprocessor Report, vol 9. n. 4, March 1995.

[13] J. Turley, “PowerPC Adopts Code Compression”, Microprocessor Report, October 1998.

[14] S. Manne, A. Klauser and D. Grunwald, “Pipeline Gating: Speculation Control for Energy Reduction”, in Proc. of the 25th Int. Symp on Comp. Arch. ,June 1998, pp.132-141.

[15] T.Sato and I. Arita, “Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values”, in Proc. of the 2000 Int. Conf. on Supercomp., May 2000, pp.196-205.

[16] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, S.H. Kim and W. Ye, “ Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower”, in Proc. of the 27th Int. Symp on Comp. Architecture, 2000, pp. 95-106.

[17] T. Wada, S. Rajan and S. Przybylski, “An Analytical Access Time Model for On-Chip Cache Memories”, IEEE Journal of Solid-State Circuits, v.27, n. 8, pp. 1147-1156, Aug. 1992

[18] A. Wolfe and A. Channin, “Executing Compressed Programs on an Embedded RISC Architecture”, in Proc. of the 19th Int. Symp. on Microarchitecture, 1992.