BEAM: Bus Encoding Based on Instruction-Set-Aware Memories

Yazdan Aghaghiri
University of Southern California
3740 McClintock Ave
Los Angeles, CA 90089
Tel: 213-740-4437
Fax:213-740-9803
yazdan@sahand.usc.edu

Farzan Fallah
Fujitsu Laboratories of America
595 Lawrence Expressway
Sunnyvale, CA 94085
Tel: 408-530-4544
Fax: 408-530-4515
farzan@fla.fujitsu.com

Massoud Pedram
University of Southern California
3740 McClintock Ave
Los Angeles, CA 90089
Tel: 213-740-4458
Fax: 213-740-9803
pedram@ceng.usc.edu

Abstract – This paper introduces a new approach for minimizing power dissipation on the memory address bus. The proposed approach relies on the availability of smart memories that have certain awareness of the instruction format of one or more architectures. Based on this knowledge, the memory calculates or predicts the instruction and data addresses. Hence, not all addresses are sent from the processor to the memory. This, in turn, significantly reduces the activity on the memory bus. The proposed method can eliminate up to 97% of the transitions on the instruction address bus and 75% of the transitions on the data address bus with a small hardware overhead. The actual power savings of 85% for the instruction bus and 64% for the data bus were achieved for a per-line bus capacitance of 10pF.

1 Introduction

With the rapid increase in the complexity and performance of VLSI chips and the popularity of battery-powered, handheld electronic systems, power consumption has become a key consideration in the design process. In every processor, a considerable number of I/O pins (i.e., address and data bus pins) are dedicated to interfacing with external memory. To reduce the power consumption, the values sent over these buses can be encoded so that the bus switching activity is minimized. A number of coding techniques have been proposed in the literature to reduce the number of transitions in memory buses ([2] - [13]). In all of these techniques, encoders and decoders are employed to encode and decode the addresses and/or data that are sent over the bus. Consequently, when evaluating the performance of an encoding technique, one must consider not only the percentage of switching activity reduction on the bus, but also the delay and power consumption overhead of the encoding and decoding circuitry. Notice that this circuitry may be implemented in the processor and in the memory chips in order to minimize the power dissipation and speed overhead.

In this paper we propose a new technique for minimizing power dissipation on the memory address bus. This technique, which we will call BEAM for "<u>B</u>us <u>E</u>ncoding based on instruction-set-<u>A</u>ware <u>M</u>emories" is based on programmable smart memories that can be configured to attain certain awareness about the instruction set architecture of the processor. Consequently, to some extent, they can calculate or predict the instruction or data addresses, thereby, eliminating the burden on the processor to send these addresses on the memory bus. This will in turn reduce the switched capacitance of the bus. Therefore, in the BEAM technique, memory attempts to calculate or predict the next address based on the information that it has collected from the current executed instruction and its address. The processor supervises the memory's address generation. If the generated address by the memory is correct, the processor does not send anything on the bus; this results in less traffic and lower activity on the address bus. The BEAM technique uses a simple decoder and encoder when compared to other bus encoding techniques as will be illustrated in the subsequent sections.

The reminder of this paper is organized as follows.[1] Section 2 gives an overview of the BEAM technique. Section 3 describes the BEAM technique in detail for instruction and data addresses. In Section 4 we present a quantitative evaluation of BEAM when different levels of prediction are progressively included into the system. Section 5 includes our power saving results, whereas Section 6 concludes the paper.

2 Basic Approach

Embedded processors dominate the total number of shipped processors. Many of these processors are designed for applications that do not require a very high performance when compared to the state-of-the-art general-purpose processors. The clock frequency of these processors is also much lower than their general-purpose counterparts. Low power consumption is often a vital design criterion for the embedded processors, especially when these processors are designed for use in battery-powered systems. Tight constraints on power consumption prevent the embedded processors from having complex micro-architectural characteristics such as memory management, branch prediction, and out-of-order execution. Furthermore, many of the embedded processors do not have on-chip caches.[2]

In this work we assume a processor without internal cache and out-of-order execution, which is a good example of a low power embedded processor. In such a system, the energy consumed on the external memory bus of the processor can be a large portion of the total power dissipation. Therefore, low power bus encoding techniques can greatly reduce the overall power consumption of such a system.

In a typical embedded system, to access an instruction or data, addresses are generated in the processor and sent over an address bus to the memory. In the BEAM technique, in most cases, the address is generated inside the memory; consequently, there is no need for the processor to send anything over the bus. The address is either calculated or predicted in the memory. In the first case the generated address in memory is always correct, whereas in the second case it may not be correct. The processor uses the same technique as the memory to calculate or predict the next address. If it is possible to exactly calculate the address, the processor knows

1 This research was supported in part by NSF grant no. 9988441 and by a grant from Fujitsu Laboratories of America.
2 Some fast growing category of applications, in which on-chip caches are not used, are the stream processing and network processing. Using a cache does not help to increase the performance for these applications.

that the memory will be able to calculate the next address correctly. Otherwise, the processor checks the correctness of the address predicted in the memory. If the predicted address is correct, the processor does nothing. Otherwise, it intervenes by sending a signal or the correct address to the memory. In the remainder of this paper we use calculating and predicting interchangeably.

Predicting the address values leads to a reduction in the switched capacitance of the bus; therefore, it reduces the power consumption. Both instruction and data addresses can be predicted. In this paper, we examine instruction and data addresses separately and propose methods that are appropriate for each of them. The two sets of methods can easily be integrated so as to predict both instruction and data addresses on a multiplexed bus. For the memory to predict the addresses, it should know format of the instructions of the processor. Note that it is not necessary to design a specific memory for each Instruction Set Architecture (ISA). The Instruction formats of many RISC architectures are very similar. Hence, a smart memory can have several registers, which can be programmed by the processor during an initialization phase to make it compatible with a target ISA.

Figure 1 shows a block diagram of the BEAM address calculator/predictor unit in the memory. The current instruction is analyzed in an ISA-aware unit. This module identifies the instruction type. There is also a prediction/calculation module that generates the next address. This module requires the following inputs: current address, current instruction, the value on the bus, and some information from the ISA-aware module. The ISA-aware unit determines how to calculate the next address. The current instruction is needed because it may have an immediate value, which is the offset of the next address. In the next two sections, we give details of the proposed prediction schemes for instruction and data addresses.

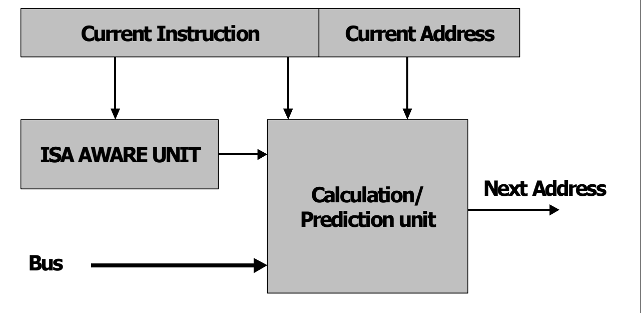

Figure 1- Block diagram of the calculation/prediction unit in memory.

3 Next-Address Prediction in BEAM

First we describe our encoding method for the instruction addresses. Next we present our data address encoding technique. We define the **transition cost of an instruction/data** address as the number of bit-level transitions that occur on the memory bus when the address of the next instruction/data is sent over the bus (with or without encoding.)

3.1 Instruction Addresses

In a typical program, one out of every seven instructions is a control flow instruction [1]. This implies that, most of the time, instructions are fetched from consecutive memory locations. This huge spatial correlation in the instruction addresses is the key factor in many low power bus-encoding techniques such as T0 method [2]. For example in the T0 method, if the new address is one (stride value) larger than the previous address (i.e., the new address is sequential), then the new address is not sent over the bus and the bus is frozen. The new address will only be sent over the bus if it is not sequential. T0 consists of one extra bit for informing the memory whether or not the bus is frozen. We adopt the same approach as T0 to predict all sequential addresses. Furthermore, we add other prediction schemes to decrease the switching activity on the bus.

To eliminate the transitions of non-control flow instructions in the BEAM technique, memory should distinguish them from control flow instructions and calculate the next address. The interesting difference between T0 and BEAM is that the latter does not require any redundant bit. Suppressing the redundant bit in T0 results in ambiguity on the memory side every time the target of a control flow instruction is equal to the value on the bus [7]. However, this problem does not exist in BEAM because the memory looks at the current instruction to find out whether the next address is sequential or not.

For control flow instructions, the memory may be able to correctly predict the next target address, thereby, eliminating the need for the processor to send the target address on the bus. If the prediction is not accurate, the processor has to send the address or a hint to help the memory correct its prediction.

In the sequel, we use SimpleScalar processor architecture [17] to describe our method. SimpleScalar is a MIPS-based processor, which is a good representative of current commercial RISC machines. SimpleScalar has two main types of control flow instructions, jumps and branches. A branch is a conditional control flow instruction that, based on the evaluation of a specific condition, can either cause a forward/backward movement in the execution flow or have no effect on the flow. The amount of movement is determined by an immediate offset specified in the branch instruction. Taking the branch can depend on a variety of conditions such as the value of a register being equal to zero. The jump instructions are similar to branches except that they do not check any conditions. Jumps are deterministic and are classified into four different types, namely J (jump), JAL (jump and link), JR (jump register), and JALR (jump and link register). Although jumps are deterministic, the target of a jump may not be known at compile time. This category of jumps is usually called indirect jumps. This means that the jump instruction itself does not include the offset of the jump. In SimpleScalar, for J and JAL instructions, the offset is specified as an immediate value in the instruction, whereas for JR and JALR instructions, it is the value of a register that determines the target of the jump; thus, it is not known beforehand.

Going back to the BEAM technique, we start by reducing the transitions for J and JAL. Note that J is used to implement unconditional jumps in the program, whereas JAL is used to implement function calls. When JAL is executed, it links the return address to a special register, which will be used later when returning from the function. If the memory recognizes J and JAL, it can easily compute the address of the next instruction by adding the offset embedded in the jump instruction to the current address. This is exactly how the processor computes the next address. However, in this case, the memory itself calculates the target address. On the other hand, processor does not need to send any address for the next instruction if the current instruction is J or JAL. This completely eliminates the switching activity on the memory address bus for J and JAL instructions.

The next step is to tackle the branch instructions. A not-taken branch behaves the same way as a non-control flow instruction does. The target of a taken branch can be easily calculated by extracting the offset from the instruction and adding it to the current address exactly like J and JAL. The problem is that the outcome of a branch is unknown in the memory and adding extra hardware to compute it is impractical. Therefore, the only possibility is to predict the outcome of branches in the memory. The prediction scheme should be as simple as possible. Suppose that memory predicts all branches as 'taken' and then calculates the targets of those branches. When executing the branch in the processor, if it is taken, the memory's prediction is correct. Hence, it is not necessary to send anything from the processor to the memory. Otherwise, memory has failed in its prediction and the processor sends a signal to the memory indicating that the branch is not taken and memory fetches the next sequential address. Therefore, memory locally predicts the branches and processor sends a signal to the memory whenever memory's prediction is not correct. To decrease the power consumption, a single bit transition on a specific line of the bus is used to signal the memory. As a result each branch will cause at most one transition on the bus. Since in a typical program about 70% of all branches are taken [1], this scheme leads to elimination of a significant number of transitions. To further improve the result, better prediction schemes may be used for predicting branches. Modern branch prediction schemes have up to 99% accuracy [1]. However, their hardware overhead is not tolerable in our system.

The last category of instructions to be tackled is JR and JALR instructions. JR is mostly used to implement function returns. To do this, the program usually reads the return address from the stack and writes it to a register. After that a jump to that address is performed by executing a JR instruction. Another usage of JR is in implementing case statements. However, it is important to note that the majority of JR instructions are used to return from functions. For the SPEC92 benchmark programs, according to [1], procedure returns account for 85% of the indirect jumps on average. We, therefore, propose a technique for predicting the target of JR instructions whenever they are used to implement function returns. JALR is mostly used to implement pointer to functions, i.e., a pointer that can point to different functions and call them from a specific place in the program. Because JALR is rarely used, it has a minor impact on the total number of transitions. Therefore, in our scheme processor sends the next address of a JALR instruction without attempting any prediction.

Our technique to reduce the transition cost of JR is as follows. The return address is saved whenever there is a function call. Later, when there is a JR instruction, the saved addresses are used to predict the target address. Therefore, a stack is used to save the return addresses. Since this scheme will not work for all JR instructions (because not all JR instructions are used to implement function returns), the processor has to have a stack as well to find out if the memory is able to correctly predict the address. Upon encountering a JAL or JALR, the return address (the same address that is linked during execution of these instructions) is pushed into the stacks. Later, when a JR is executed, if it is a function return instruction, its return address ought to be present in the stack. If the JR is not a function return, then its target will not be in the stack and cannot be predicted. This is confirmed in the processor by comparing the value of the register to which JR is jumping with the value stored on the top of the stack. If they match, then the processor knows that memory correctly predicts the target and the processor does not change the value on the bus. If the target of the jump is not equal to the value stored on top of the stack, then the memory prediction will be incorrect and the processor will simply send the new address over the bus. Memory detects the activity on the bus and concludes that its predicted value was incorrect. Consequently, it uses the address received on the bus instead of the address on top of the stack.

An essential question is how large the size of the stack should be. For any stack with finite number of entries, there is always the possibility of an overflow. Once the stack overflows, no JR will be predicted correctly until enough function returns are executed so that the number of nested function calls becomes less than the size of the stack, which is not suitable because many nested functions may not return until the very end of the program. However, if we make the stack circular, this problem will be solved. Although there is still the possibility of overflow for circular stacks, the most recent return addresses will not be lost in the case of overflow and the prediction scheme will perform better. In Section 4 we will address the issue of the stack size quantitatively.

3.2 Data Addresses

In this subsection we describe how data addresses can be predicted. Reading or writing to memory in SimpleScalar is done only with load and store instructions. There are two different addressing modes in this architecture. The first one is displaced addressing in which the address to be accessed is calculated by adding a register value to an offset embedded in the instruction,

Rd <= MEM(Rs + Offset).

The second type is indexed addressing in which the address is calculated by adding the values of two registers,

Rd <= MEM(Rs + Rt).

If the memory wants to calculate the address accessed by these instructions, it has to know the value of the registers. However, the register file is in the processor and the memory does not have access to it. The solution we adopt is to implement a *shadow register file* in the memory. If the shadow register file is kept completely coherent with the processor register file, then the accessed addresses can be easily calculated in the memory. However, making the two register files coherent is very expensive in terms of the number of required bus transactions. Thus, the values of the registers in the shadow register file are updated only when there is a memory access instruction with displaced addressing. Every time the processor sends the data address to the memory, the memory subtracts the offset embedded in the instruction (which is known to the memory) to calculate the value of the register used in the instruction. Therefore, the register value can be updated in the memory shadow register file without additional overhead on the bus. Now this "semi-coherent" register file in the memory can be used to predict data addresses. On the other side, the processor can easily determine whether the value that memory has for a register is valid. This is done by keeping track of all registers that have been modified (i.e., have been the destination of a move instruction) since the last time they were used in memory access instructions as pointers. If a register has an updated value when it is used in the memory access instruction, the data address can be correctly calculated in the memory. When the processor detects that the memory does not have the updated value of the register, it sends the new address value on the bus. At the same time memory knows when the register value is not valid and will read the address from the bus instead of calculating it.

Additionally, memory uses this new value to update its register file. On both sides, a valid flag corresponding to the register is set indicating that the register value is again valid.

SimpleScalar has 32 general-purpose registers. Hence, the shadow register file should have 32 registers, but this can be expensive for a low power scheme. To reduce the number of registers, we consider the fact that compilers usually use a small set of registers as pointers in the memory access instructions. So it is possible to use a small cache to hold the value of some of the registers. Whenever a new register is used in a memory access instruction, that register occupies one entry of the cache. Therefore, the address of the next instruction using that register to access memory can be correctly predicted. In fact we will show in the next section that using a 4-entry cache instead of 32 registers will have a marginal effect on the switching activity reduction while it reduces the hardware overhead significantly. To avoid evicting registers that are more frequently used, we use a saturating counter [1].

4 Experimental Results

In this section we examine the actual transition reduction obtained by applying the BEAM technique. First we focus on instruction addresses. Figure 2 gives the percentage of different types of instructions used in SPEC2000 benchmark programs. Forward and backward branches as well as 'taken' and 'not-taken' branches have been reported separately. As we mentioned previously, JALR is rarely used in the programs. Furthermore, the number of JAL and JR instructions are almost the same in all programs, which shows that most of JR instructions are function returns for the corresponding function calls implemented by JAL instructions.

Next we compare the transition cost of different types of instructions in the original instruction address trace. The results have been reported in Table 1. By looking at this table, it is possible to determine the transition saving of our method when different instructions are handled. For example, according to the table, if jump transitions are suppressed by predicting jump targets, around 5% of the total transitions will be saved. This is not an exact number, because when a new encoding is applied to cancel the transition cost of jumps, other transition costs may also be affected. This is because the transition cost of an instruction depends on the current value on the bus and the next address. Notice that our method is similar to T0 for predicting the sequential addresses. However, because our method does not require an extra bit, it performs better than T0. According to our experimental results, the BEAM technique using sequential prediction only outperforms T0 by 5%.

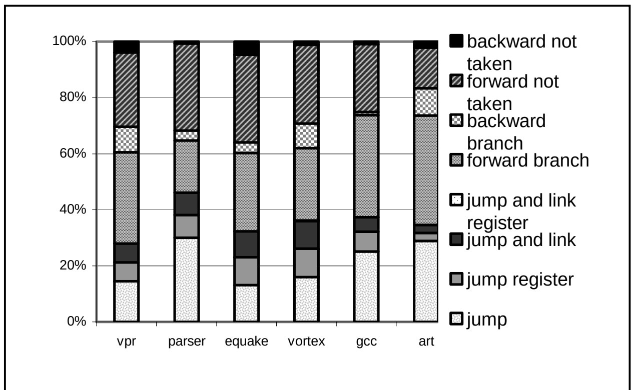

Figure 2- Number of different control flow instructions.

Table 2 shows the savings achieved by the BEAM technique when different levels of prediction are used. As one can see, adding the branch prediction method to simple sequential prediction increases the transition saving to 86.2% up from 65.0%. If the prediction of J and JAL instructions is also included, the saving increases to 92.1%. Until this point, the required extra hardware is essentially negligible; we only need to use an adder (in order to add the extracted offset to the current address), several multiplexers and some logic for detecting the instructions' types. To predict the target of JR instructions, we have to use two stacks to store return addresses: one in the processor and the other in the memory. In Table 2, it is assumed that a 10-entry circular stack is used for this purpose.

Table 1- Percentage of transition costs for different types of instructions.

	Jump		Jump and link		Jump register		Jump and link register		Forward taken branch		Backward taken branch		Not taken branch		Non-control flow	
vpr	1742	5%	1716	5%	1584	5%	1	0%	2935	9%	879	3%	2021	6%	22789	68%
parser	2405	7%	1412	4%	1104	3%	0	0%	1696	5%	127	0%	2008	6%	23425	73%
equake	1400	4%	1775	5%	1955	6%	1	0%	2310	7%	532	2%	2029	6%	24274	71%
vortex	979	3%	1709	5%	1630	5%	35	0%	2206	7%	589	2%	1279	4%	25252	75%
gcc	2319	7%	1123	3%	1383	4%	12	0%	3081	9%	102	0%	1513	4%	24136	72%
art	1768	6%	253	1%	216	1%	1	0%	1001	3%	357	1%	569	2%	27151	87%
Average	5.3%		3.8%		4.0%		0.0%		6.7%		1.3%		4.6%		74.3%	

Table 2- Transition saving for different stages of our proposed method for instruction address bus.

	Transition Saving			
	predict seq. ins	+ predict branches	+ predict J and JAL	+ predict JR
vpr	58.2%	84.9%	90.6%	95.5%
parser	64.3%	85.2%	91.5%	97.1%
equake	59.3%	83.7%	89.4%	96.7%
vortex	65.1%	87.2%	92.7%	97.8%
gcc	60.6%	84.8%	91.8%	97.2%
art	82.1%	91.4%	96.6%	99.2%
Average	65.0%	86.2%	92.1%	97.3%

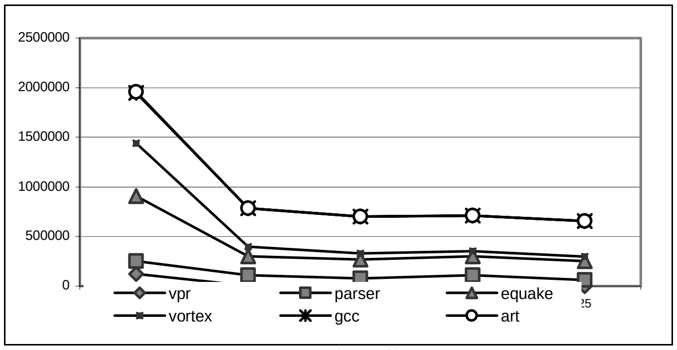

Figure 3- JR transitions for different stack sizes.

Figure 3 shows the effect of the size of the circular stack on transition saving. Vertical axis shows the number of transitions caused by JR instructions. According to the figure, a 10-entry stack is sufficient.

By using our method, up to 97.3% of all transitions of instruction addresses can be suppressed. The remaining 3% of the transitions are dominated by transitions caused by the misprediction of the branches. More saving can be achieved by using better branch prediction schemes, but this would require more complex hardware.

Next we quantitatively investigate the prediction of data addresses. We have used the same set of benchmark programs and generated the data address traces for them. A precise prediction represent a case when the value of a register is valid in the memory shadow register file and the memory can exactly predict the data address. Table 3 shows the percentage of transition saving and the precise predictions when we use a 32-entry shadow register file in the memory. In Table 4 we have reported the results when using a 4-entry cache (with direct mapping of registers into the cache entries) instead of the full size shadow register file. The average saving in transition count declines by only 6%.

Table 3 -Transition saving for data addresses and the percentage of accesses precisely predicted for a full size shadow register file.

	Transition saving	Precise predictions
vpr	86.1%	74.7%
parser	80.5%	65.1%
equake	76.3%	82.1%
vortex	74.2%	74.3%
gcc	81.9%	71.1%
art	95.4%	85.6%
Average	82.4 %	75.4%

Table 4- Transition saving for data addresses, cache hit, and the percentage of accesses precisely predicted for a 4-entry cache.

	Transition saving	Cache hit	Precise predictions
vpr	80.3%	87.3%	68.3%
parser	74.6%	81.3%	60.6%
equake	70.7%	88.4%	74.8%
vortex	66.9%	87.0%	69.1%
gcc	77.3%	81.5%	61.0%
art	88.1%	88.1%	84.7%
Average	76.3%	85.6%	69.8%

5 Power Analysis

We report the actual power saving (i.e., one that accounts for the power dissipation overhead of the codecs) achieved by the BEAM technique. We do this evaluation separately for instruction and data address traces. We do the power estimation for the blocks used in the memory only, namely, memory codecs. This is because, on the processor side, most of the required hardware is already in place or there are similar logic blocks, into which the required codec hardware can be integrated with a minor overhead. For example, processor needs to identify the type of the instructions such as branches and jumps. This task is already done in the instruction decode unit. In fact in some cases, our technique even decreases the burden on the processor. As an example, the processor no longer has to calculate the target address for a jump or a branch since this is always done in the memory. The outcome of the branches still has to be determined, but the actual target calculation is no longer needed. This is similar to moving the adder for calculating the target of jumps and branches from the processor to the memory chip. Notice that we consider a version of the BEAM technique that does not employ the JR prediction. The reason is that the 5% reduction in the transition activity that can be achieved by implementing the JR prediction does not justify using two different stacks in many cases.

Figure 4-1 shows the BEAM codec, which is used in memory for predicting branches and direct jumps. The instruction-set aware unit only determines whether the instruction is a JR/JALR, a branch or a sequential instruction. If it is a branch, the leftmost multiplexer selects the branch offset. Otherwise, the jump offset is selected. Either this value or the instruction stride is added to the current address. If the current instruction is not a JR or a JALR, this value will be the next address. Otherwise, the value that is received from the bus will determine the next address. The JR/JALR signal generated by the Instruction-Set Aware Unit is used for controlling the multiplexer that chooses between these two addresses.

Figure 4-2 shows the blocks required in the memory for calculating data addresses. There is a cache with four entries that can hold the values of four registers. The Rs field in the current instruction is used to index into the register cache. The Rt field shows that a register has been used as a target register and is used to invalidate a cached register. The Instruction-Set Aware Unit sends two signals to the cache, namely, invalidate and Mem-Access. Any access to the cache may lead to a hit or a miss. Even if it hits, the register value may be invalid meaning that the value of the register has been modified by an instruction after the last memory access. Thus, only if there is a valid hit, the value is used for calculating the address by adding it to the offset embedded in the instruction. If there is no valid hit, the value will be received on the bus from the processor and the rightmost multiplexer will select this value as the data address. At the same time the value is used to update one of the entries in the cache using direct mapping.

Table 5- Results of hardware analysis and power estimation.

	Instruction codec	Data codec
Num of Literals	686	720
Area (* 1000 λ^2)	343	588
Num of Gates	311	528
Original Bus Power (uW)	5205	20050
Bus Power with BEAM (uW)	416	6055
Power of BEAM memory Codec (uW)	364	1144
Codec power + Bus Power with BEAM	780	7199
Power Saved with BEAM (uW)	4421	12851
Percentage Saving over bus	85 %	64%

To estimate the actual overhead of the above memory codecs, first, we generated the netlist of each circuit in Berkeley Logic Interchange Format (BLIF). The netlists were optimized using the SIS script.rugged and mapped to a 1.5 Volt, 0.18µ CMOS library using the SIS technology mapper. The I/O voltage was assumed to be 3.3V. The number of literals, the area and the number of gates have been reported for both the instruction and data codecs in Table 5. Next we calculated the power consumption of these circuits. These values are needed to determine the actual power reduction of the bus. Therefore, instruction and data address traces of the benchmark programs were fed into the codecs and the power consumption was estimated using sim-power [16], a gate-level tool. The results for a 100 MHz system clock are reported in Table 5 as "power of BEAM memory codec." Assuming bus capacitance of 10pF/line, we have calculated the original bus power (i.e., when no encoding is used) using the same address traces that we used for the estimation of the power by sim-power. The total power saving considering extra on chip codecs and the percentage of saving are also reported in Table 5.

6 Conclusion

In this paper we described a new method for encoding instruction and data address buses. Our method can achieve up to 97% reduction in switching activity for an instruction address bus. For a data address bus, the saving is approximately 64%. The small hardware overhead makes our method practical. Our experiment shows that the power consumption of memory instruction and data codecs are 0.36mW and 1.15mW, respectively. In practice, when using our method, several modules are moved from the processor to the memory and some new blocks are added to the processor. Therefore, the processor power consumption remains almost the same or even decreases. Our techniques can be combined to predict the address in a multiplexed address bus.

7 References

[1] Hennessy, Patterson, *Computer Architecture, A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, 1996.

[2] L. Benini, G. De Micheli, E. Macii, D. Sciuto, C. Silvano, "Asymptotic Zero-Transition Activity Encoding for Address Buses in Low-Power Microprocessor-Based Systems," *Proc. 7th Great Lakes Symposium on VLSI*, Urbana, IL, pp. 77-82, Mar. 1997.

[3] W. Fornaciari, M. Polentarutti, D.Sciuto, and C. Silvano, "Power Optimization of System-Level Address Buses Based on Software Profiling," *Proc. International Symposium on Hardware/Software Codesign*, pp. 29-33, Apr. 2000.

[4] S. Ramprasad, N. R. Shanbhag, and I. N. Hajj, "A Coding Framework for Low-Power Address and Data Buses," *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 7, No. 2, pp. 212-221, Jun. 1999.

[5] S. Ramprasad, N. R. Shanbhag, I. N. Hajj, " Sigal Coding for Low Power: Fundamental Limits and Practical Realizations", IEEE Transactions on Circuits and Systems, II, Vol. 46, No. 7, pp. 923-929, Jul. 1999.

[6] E. Musoll, T. Lang, J. Cortadella, "Exploiting the locality of memory references to reduce the address bus energy," *Proc. International Symposium on Low Power Electronics and Design*, pp. 202-207, Aug. 1997.

[7] Y. Aghaghiri, F. Fallah, M. Pedram, "Irredundant Address Bus Encoding for Low Power," *Proc. International Symposium on Low Power Electronics and Design*, pp. 182-187, Aug. 2001.

[8] Y. Shin, S. I. Chae, K. Choi, "Partial Bus-Invert Coding for Power Optimization of System Level Bus," *Proc. International Symposium on Low Power Electronics and Design*, pp. 127-129, Aug. 1998.

[9] N. Chang, K. Kim, J. Cho, "Bus Encoding for Low-Power High-Performance Memory Systems", *Proc. 37th Design Automation Conference*, Jun. 2000.

[10] L. Benini, G. De Michelli, E. Macii, M. Poncino, and S. Quer, "System-Level Power Optimization of Special Purpose Applications: The Beach Solution," *Proc. International Symposium on Low Power Electronics and Design*, pp. 24-29, Aug. 1997.

[11] M. Ikeda, K. Asada, "Bus Data Coding with Zero Suppression for Low Power Chip Interfaces," *Notes of the International Workshop on Logic and Architecture Synthesis*, pp. 267-274, Dec. 1996.

[12] P. P. Sotiriadis, A. Chandrakasan, " Bus Energy Minimization by Transition Pattern Coding (TPC) in Deep Submicron Technologies,", *Proc. International Conference on Computer Aided Design*, pp. 317-321, Nov. 2000.

[13] L. Macchiarulo, E. Macii, M. Poncino, "Low-energy for Deep-submicron Address Buses", Proc. *International Symposium on Low Power Electronics and Design*, pp.176-181, Aug. 2001.

[14] P. Chang, E. Hao, Y. N. Patt, " Target prediction for indirect jumps", Proc. *24th International Symposium on Computer Architecture*, pp. 274-283, Jun. 1997.

[15] J.E. Smith, " A Study of Branch Prediction Strategies", *Proc. 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[16] S. Iman, M. Pedram, " POSE: Power Optimization and Synthesis Enviroment," *Proc. 33rd Design Automation Conference*, pp. 21-26, Jun. 1996.

[17] http://www.simplescalar.com.

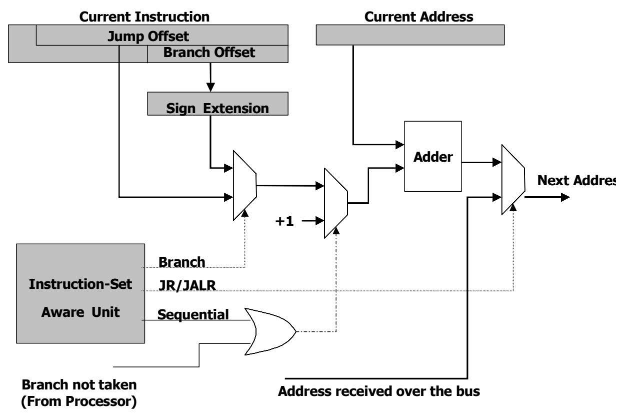

Figure 4-1 Hardware implemented in memory for predicting instruction addresses (jump and links, jumps and branches).

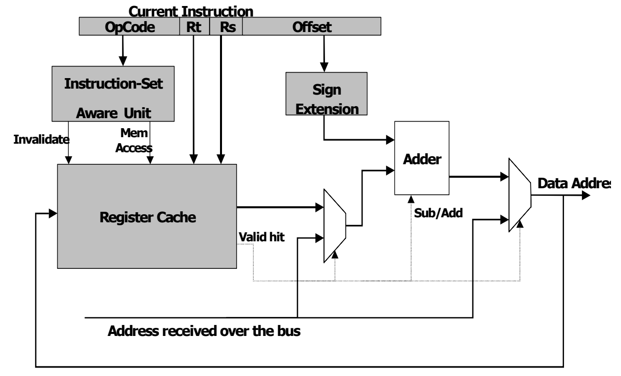

Figure 4-2 Hardware implemented in memory for predicting data addresses.