# Abridged Addressing: a low power memory addressing strategy

***Abstract*— The memory subsystem is known to comprise a significant fraction of the power dissipation in embedded systems. The memory addressing strategy, which determines the sequence of addresses appearing on the memory address bus as well as the switching activity in the addressing logic, has a major impact on the memory subsystem power dissipation. We present a novel addressing strategy, *Abridged Addressing*, that helps reduce system power dissipation by substantially reducing both the address bus switching as well the addressing logic power. The strategy, which relies on minimizing register accesses in the addressing logic, helps overcome some of the limitations of existing approaches: the address bus switching is low; there is very little area, performance, and power overhead; and the addressing hardware is simpler, making the technique suitable for both on-chip and off-chip memory, as well as single-port and multi-port memories.**

## I. Introduction

Memory accesses constitute an important target for power optimizations since memory accounts for an increasing fraction of the total area and power dissipation of embedded systems [1], [2]. Since dynamic power dissipation in CMOS circuits is a function of the total switching capacitance, particular attention has been paid to the memory address and data buses because these are wide, and are typically long with high capacitance. Lower switching on these buses leads to lower overall power dissipation. Although it may not be possible to optimize well for the data bus because the relevant data values are typically not known in advance, the switching on the address is often known statically and is controllable by the synthesis process. In typical design descriptions, the access patterns of the arrays – the loop induction variables and the array index expressions are known and usually exhibit some regularity. This regularity results in a strong correlation in the sequence of addresses placed on the address bus, which can be exploited by a power optimizing synthesis tool.

The idea of encoding the address bus with the objective of minimizing switching has been extensively studied [1]. In Bus-invert coding [3], an additional bit is appended to the bus to indicate that the bus is inverted – this is invoked if too many bits (more than half) are switching on the bus. In [4], the instruction address bus was encoded using Gray code. The T0 encoding scheme [5], where an extra bit encodes the information that the following address is the successor of the previous address, results in zero transitions in the best case – i.e., where addresses are strictly sequential.

Recognizing that programs tend to spend a lot of time accessing arrays inside loops, [6] proposed an address encoding technique called Working Zone Encoding (WZE). The technique involves the construction of encoding and decoding circuits that keep track of currently active memory zones. During memory accesses, the zone is selected and an encoded offset is transmitted on the address bus, which is used by the decoder to recover the actual address. More general correlations on memory data buses have been studied, which rely on having static access to data streams so that frequently occurring patterns can be identified and assigned appropriate codes [7], [8]. In [9], the authors present an encoding/decoding strategy consisting of adaptive self-organizing lists to handle complex interactions arising out of multiplexed address buses. In [10], the authors first profile an application to determine which regions of memory are most accessed. To reduce power dissipation, these regions are assigned to smaller memory modules. In [11], a strategy was described for arranging data in memory so that switching on the address bus is minimized. In works such as [11], [12], the authors propose an array interleaving strategy which can help sequentialize the addresses to some extent. However, this strategy does not generalize well because interleaving requirements in different loops might conflict. In this paper, we assume that the start addresses of arrays have been determined, and the structure of the individual arrays is not modified; interleaving could be combined with our proposed strategy.

For data memory, the encoding strategies indicated above work well for sequential addresses, but not for correlated addresses that are not necessarily sequential, although WZE and [9] address this deficiency to some extent. In a recent work [13], the authors present a customized data memory interface (CDMI) that uses static information about the loop strides and the order of memory accesses to generate addresses within the data memory subsystem, thereby minimizing communication across the processor-memory address bus. Although proposed in a processor context, a similar strategy can also be applied to synthesized hardware. The technique requires a relatively power-expensive decoding circuit in the memory subsystem, which is justified in case of off-chip power-memory bus, because power savings on the address bus dwarfs the decoding overhead. However, in case of on-chip memory, the address bus capacitances are relatively smaller, and the decoder hardware's power overhead becomes significant. We present a novel strategy, *Abridged Addressing*, where we optimize register file accesses and simplify the addressing logic by harnessing memory address correlations. This not only reduces address bus energy, but also significantly reduces power dissipation in the addressing logic, thereby making the addressing mechanism equally applicable to both the on-chip

as well as off-chip memory interfaces.

## II. Memory Addressing

When designing optimizations targeting the data memory interface, particular attention is paid to loops in the specification program because most of the execution time is known to be spent inside loops, irrespective of whether the target is software or hardware; consequently, most memory accesses occur during loop executions. Since typical loops in a large variety of application domains (such as DSP, graphics, vision, image processing, etc.) have regular loop strides and array index expressions that are affine (linear) in the loop induction variables, it is possible to analyze the array access patterns statically and synthesize power-efficient address generation, encoding, and decoding circuits that minimize the overall power consumption during memory accesses. Consider a loop of the form:

```
for (i = 0; i < n; i = i + 2)
    x = a[i] + b[i+1];
    ...
```

Assume that the implementation target uses a dual-port memory because of performance constraints. The addressing logic now drives two memory address buses. A straightforward implementation of the addressing logic section of the datapath is shown in Figure 1. This architecture suffers from some serious disadvantages when the memory is physically far apart from the datapath. For example, when the memory is off-chip, the power dissipation in the address buses is large due to the transitions in the off-chip wires with large capacitances. Several optimization strategies target this interface typically with encoders on the datapath side, and decoders on the memory side to minimize the transitions in the address bus. A simplified version of an example efficient implementation, based on the CDMI proposal in [13] is shown in Figure 2. Because of the regular memory access pattern in the loop, there is no need to send the addresses explicitly on the address bus in the steady state. Instead, the initial addresses are sent to registers in the decoders only once, and the registers are updated after every access so that subsequent memory addresses are generated in the decoder itself. A small counter-based FSM (which is initialized before each loop begins) keeps track of the register updates. The technique is very effective in minimizing address bus traffic on the datapath-memory interface because the datapath updates the address only once for every loop.

The above strategy yields significant power savings when the datapath-memory interface consists of high capacitance wires, especially for off-chip memories where the switching capacitances are about three orders of magnitude higher than typical on-chip wire capacitances. However, if the memory is on-chip, then the power overheads of the decoding circuitry may overwhelm the power savings due to reduced address bus switching; the overall power savings depends on the physical placement of the datapath and memory blocks. Since the addressing decisions are made very early in the design phase, when the physical information is not known, it is important to design the addressing logic in such a way that power overheads are minimized.

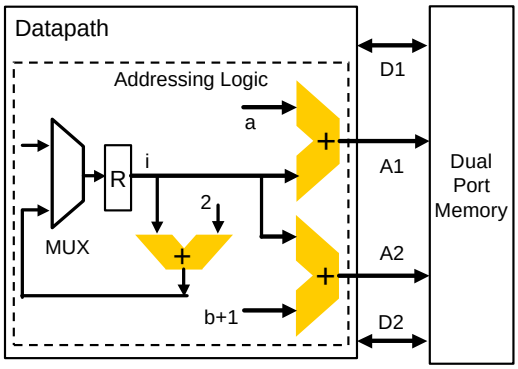


Fig. 1. The datapath/memory interface

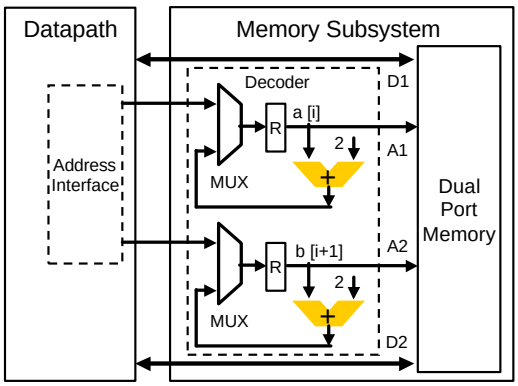


Fig. 2. Minimizing address bus transitions

Figure 3 shows an alternate address generation mechanism, *Abridged Addressing*, where the address for $a[i]$ is generated using a register as before, but the address for $b[i+1]$ is generated from that of $a[i]$ by just adding $(B-A+1)$ where $A$ and $B$ are the start addresses of $a$ and $b$ respectively. For simplicity of discussion, we have used only word addresses in this paper. Since $(B-A+1)$ is a constant known during synthesis, the addressing logic becomes simpler. In the previous approach (Figure 2), we need to keep a register in the decoder for every memory access, and if there are multiple ports, we may organize them into one register file for each port. However, reading from and writing to these register files in each cycle is power-inefficient. Accesses to the decoder's register file results in extra power consumption during read/write of the individual registers as well as during the register file address decoding. The alternative we propose in Figure 3 maintains only one address register for a loop body, and derives the remaining addresses from this register using simple arithmetic operations. Our experiments show that replacing the expensive register file access operations by simple additions cause a 23% reduction in power dissipation for the above example.

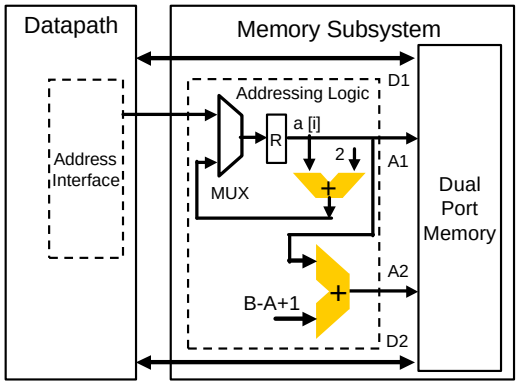


Fig. 3. Abridged Addressing: minimizing address bus transitions and addressing logic

## III. Address Decoder Circuits

The CDMI-based approach, although proposed in the processor context in [13], can be extended in a straightforward way to synthesized hardware. Consider a general $n$-level nested loop accessing an $r$-dimensional array $A[N_1][N_2]...[N_r]$ as follows:

**for** $(i_1 = l_1; i_1 < h_1; i_1 = i_1 + s_1)$
  **for** $(i_2 = l_2; i_2 < h_2; i_2 = i_2 + s_2)$
    ...
    **for** $(i_n = l_n; i_n < h_n; i_n = i_n + s_n)$
      READ $A[c_{11}i_1 + c_{12}i_2 + \ldots + c_{1n}i_n + k_1]$
        $[c_{21}i_1 + c_{22}i_2 + \ldots + c_{2n}i_n + k_2]$
        ...
        $[c_{r1}i_1 + c_{r2}i_2 + \ldots + c_{rn}i_n + k_r]$

The difference in address locations between two successive iterations of the innermost loop is:

$$A_d = (N_2N_3 \ldots N_r)c_{1n} + (N_3 \ldots N_r)c_{2n} + \ldots + N_r c_{r-1n} + c_{rn}$$

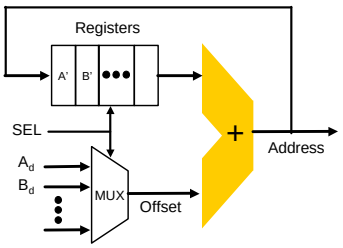


Fig. 4. Address decoder for affine array indices

For affine indices, all the $c_{ij}$'s and $N_i$'s are constants, so $A_d$ can be computed at compile time and set as an input to the MUX shown in the generalized addressing circuit shown in Figure 4. The MUX inputs $A_d$, $B_d$, etc., represent the address offsets to the respective arrays in successive iterations of the $i_n$-loop. The values in the registers, $A'$, $B'$, etc., represent the addresses for arrays A, B,...respectively from the previous iteration. The address for the current access is determined by the SEL signal from a small counter-based FSM selecting both the appropriate register and the offset input to the adder. The register file is initialized before the loop execution begins from the address bus in the datapath interface, but in the steady state, the bus from the datapath never changes and addresses to the memory are generated within the decoder itself. This is the result of an application of a combination of the strength reduction and induction variable elimination compiler optimizations.

## IV. Abridged Addressing Logic Structure

The structure of the typical addressing logic in Abridged Addressing is shown in Figure 5. $A[i]$, $B[3i]$, $C[11i]$, and $D[11i+1]$ represent the address computations required; there is one address computation output corresponding to every array access in the loop body. There will be a final level of multiplexing between these independent computations depending on the number of ports in the memory and on the assignment of individual addresses to ports. As mentioned earlier, we store only one value in a register and derive all the address outputs from it. In this case, we choose to store $i$, the loop induction variable, which is incremented by the loop stride every iteration. A network of adders is required to generate all the memory addresses, as shown. If the array indices are affine, the multiplicative coefficients are constants and there is no need to instantiate expensive multipliers. Also, the shift circuits shown in the figure do not correspond to any actual hardware; shifting is realized though concatenation of zeros. Such a structure results in considerably lower power dissipation than the CDMI-based strategy, which would consist of a register file with four registers – each storing the updated memory address; the CDMI decoder performs reads and writes to the register file during every memory access. In Abridged Addressing, the $i$ register is updated only once per loop iteration. Even though there may be more adders in Abridged Addressing, each adder is activated only once per loop iteration (and not on every memory access), which may result in only slightly worse power dissipation in adders than CDMI, where a single adder is activated on every access. In the example of Figure 5 there are six adders activated once whereas in CDMI, we would have one adder activated four times. However, there is a very considerable power saving due to avoiding the expensive register file accesses.

Although we chose the loop induction variable $i$ for storage in Figure 5, there are other cases when this choice does not necessarily result in the most efficient design. A simple example is shown in Figure 6. To generate the $A[11i]$ and $D[11i]$ addresses in every loop iteration, we can just store $A[11i]$ in the register and simply derive $D[11i]$ from it by adding the constant $D - A$. If the loop stride was '1', then we add '11' to the current $A[11i]$ address to generate the value in the next iteration.

## V. Efficient Addressing Circuits

We formulate and solve the problem of determining an efficient addressing logic structure from a given set of array

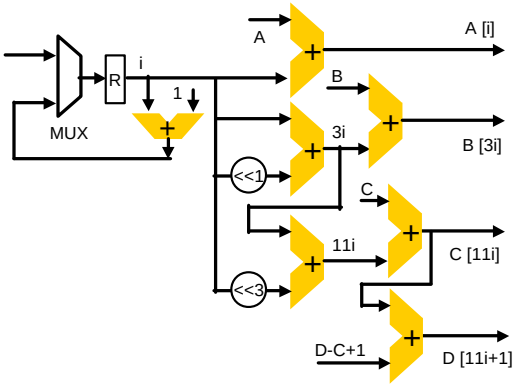


Fig. 5. Addressing logic structure

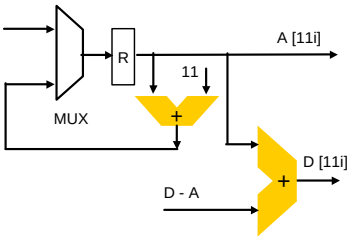


Fig. 6. Addressing logic for a different example

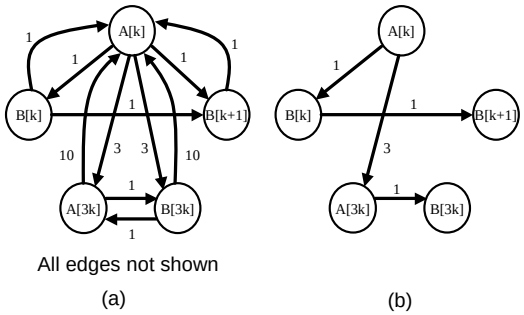


Fig. 7. (a) Adder network graph (b) Directed minimum spanning tree

references, each translating to a memory address output. The overall strategy is similar in principle to that used in designing multiplier-less filters with minimum resources (e.g., [14]), but differs in several important steps.

We construct an adder network graph $G(V, E)$, with a set of nodes $V$ and edges $E$. Each node corresponds to an array access, and there is an edge $i \rightarrow j$ with edge weight $w(i, j)$ if it is possible to derive the address corresponding to $j$ from the address corresponding to $i$ using $w(i, j)$ adders. An example adder network graph is shown in Figure 7(a). The edges $A[k] \rightarrow B[k]$, $B[k] \rightarrow A[k]$, $B[k] \rightarrow B[k+1]$, etc., have weight 1 because it is possible to derive the target address from the source address using just one adder. The edge $A[k] \rightarrow B[k+1]$, has weight 1 because we also need only one adder to generate $B[k+1]$ from $A[k]$ (add the constant '$B - A + 1$' to $A[k]$). The edge $A[k] \rightarrow B[3k]$, has weight 3 because we need three additions – first subtract $A$ to get $k$, then add $k + 2k$ to get $3k$, finally add $B$ to get $B[3k]$. Edges such as $B[3k] \rightarrow B[k]$, have a higher weight because the derivation requires more expensive circuitry – in the worst case, an appropriate divider. The multiplication operation corresponds to an appropriate number of adders, since the multiplicative coefficients are all constants. The edge weights in the graph are a measure of how many adders would be activated to get from one memory address to another, and reflect the relative power dissipation of the circuit compared to a simple adder. The same metric can be used for modeling the area of the addressing logic, and with small modifications, the delay also.

From the adder network graph, we can derive the simplest adder network covering all the nodes by finding the *minimum spanning tree* in the graph. The minimum spanning tree is a subset of the edges of the graph that connects all the nodes and minimizes the sum of the weights of the edges selected. The standard minimum spanning tree algorithms (which operate on undirected graphs) cannot be used for this purpose because our adder network graph is directed. This is because the circuit to generate $i$ from $j$ may be more complex than the reverse. For example, $3i$ can be generated from $i$ by just one addition, but generating $i$ from $3i$ requires more complex hardware. In other words, $w(i, j) \neq w(j, i)$ in general.

To obtain the minimum spanning tree of a directed graph, we can use Chu-Liu's algorithm [15], an efficient polynomial time algorithm for computing the directed minimum spanning tree (DMST). The algorithm works by first selecting, in an outer loop, different nodes of the graph as the possible root, deleting all incoming edges to it, and then choosing the lowest cost incoming edge to all the other nodes. If no cycle is formed, we have the DMST. If there is a cycle, it is collapsed into a super-node $x$ and the incoming edge weight to each node $y$ in the cycle from nodes $z$ outside the cycle is recomputed according to the equation:

$$w(z, x) = w(z, y) - (w(\text{pred}(y), y) - \min_i(w(\text{pred}(i), i))$$

where pred($j$) is the predecessor node of $j$ in the cycle. For each super-node, we select the incoming edge with smallest updated weight, which replaces the original incoming edge. We repeat the the procedure for the collapsed graph. The root of the resulting tree corresponds to the element we wish to store in the register of the addressing circuit. The DMST for the adder network graph of Figure 7(a) is shown in Figure 7(b).

There may be cases when the addition of an auxiliary node to the adder network graph results in a more efficient addressing logic structure (i.e., a lower cost DMST). In the example graph shown in Figure 8(a), the cost of deriving each node from the other is high. In such cases, it is beneficial to augment the graph with the node $k$ corresponding to the loop induction variable, add the edges from this node to the others, and compute the DMST, which may result in lower overall cost. The addition of the new node results in the graph of

Figure 8(b), and the resulting DMST is shown in Figure 8(c), which has lower overall cost than the DMST of the original graph. Note that the addition of the loop variable $k$ node automatically results in the implicit (zero cost) addition of nodes $2k$, $4k$, $8k$, etc., since no extra hardware is required to generate them. The weights of edges from $k$ to the other nodes takes this into account. There is no need to explicitly add these other nodes.

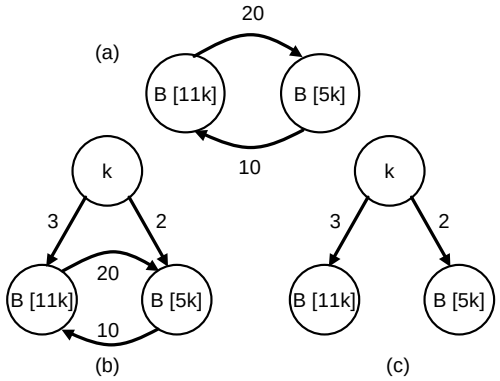


Fig. 8. (a) Example graph (b) Augmented graph (c) DMST of augmented graph

The overall strategy in Abridged Addressing is to find the DMST for both the original graph $G$ and the augmented graph $G'$ (with the loop variable node), and choose the solution with the lower overall cost. The addressing logic structure is inferred directly from the DMST.

### A. Analysis

The key idea in Abridged Addressing is to minimize the storage of redundant information in the addressing logic. For example, the continuous updation of two address registers $a[i]$ and $b[i]$ within a loop is unnecessary because the same pattern is repeated. Minimizing the storage helps reduce power-expensive register file access operations. It is important to note that the addressing logic is not an overhead – the computation would performed in the datapath in any case. We have merely transferred the address computation to the memory subsystem. The Abridged Addressing strategy is independent of the number of ports in the memory. There is only one address bus in the interface between the datapath and the memory subsystem. There is a final level of multiplexing of the computed addresses depending on the number of ports and the port assignment.

In the addressing logic, we have relied on the start addresses of the arrays and loop strides being constants known at synthesis time. If this is not the case, then the inputs to the adders come from registers which are initialized by the datapath through the single address bus. Note that although this increases the number of registers in the addressing logic, it does not affect the power dissipation adversely because these registers are not updated during the loop iterations. When there are multiple loops, the analysis of each loop is performed independently and sharing decisions of common parts of the addressing are taken later. For example, in Figure 5, if the strides of the two loops are different (say 1 and 2), the input to the adder that updates the register comes from a multiplexer whose inputs are 1 and 2, and whose select signal comes from the small FSM in the decoder that also controls the final address output. The power overhead of this FSM is small in comparison the typical power dissipation in adders. Conditionals in loop bodies do not require any additional support. The addresses of all memory accesses in the loop are computed, and the FSM selects the final output depending on the current state of the datapath subsystem, which is presented to the memory interface through the single address bus (as in [13]). When we have array accesses that are not part of loop bodies, then the address is sent over the address bus and presented as it is to the memory. In this case, there might be a small theoretical performance overhead because Abridged Addressing presents only one address bus to the datapath instead of two for a dual-port memory, but such cases occur a relatively small number of times – most memory accesses occur within loop bodies.

On investigating the possible performance overhead due to the addressing structure, we found that: (1) the cascaded structure leads to a very minimal increase in delay over a single adder. Note that the critical path delay of two cascaded adders is only marginally more than that of a single adder – e.g., the delay of two cascaded 32-bit ripple carry adders is that of one 33-bit adder; (2) more importantly, the critical path in a loop body never passes through the addressing structure, since the addresses are generated as soon as the loop index is updated, independent of when they are needed. In our experiments, we found that the addresses were generated in the first one or two cycles of a loop, whereas the remaining computation (which was on the critical path) took much longer.

## VI. Experiments

We verified the addressing optimization strategy using several examples from literature, an IBM ASIC library with a dual-port RAM as the data storage element, and the Synopsys synthesis tools. Behavioural synthesis was first performed to determine the schedule of memory accesses and the memory port assignment. Following this, the addressing logic was generated and synthesized. The synthesized designs were simulated, and the resulting switching activity file was fed into a power simulation framework (Synopsys Prime Power) to generate the total power dissipation for the application. *SOR* (Succesive Over Relaxation) and *Compress* are popular examples from numerical/scientific computing. *MM* and *Dprod* are the matrix multiplication and dot product functions. *Laplace* and *Lowpass* (accentuating low frequencies in an image) are frequently used in the image processing domain. The examples are data-intensive and have arrays indexed by affine expressions accessed in the inner loop bodies.

In our first experiment, we compared the relative power dissipation of the synthesized design examples. Here, the memory is on-chip, and the address bus capacitances, while

still significant, are not orders of magnitude larger than typical nets. The results are summarized in Figure 9. For each example, we compare the power dissipation for three different implementation strategies: WZE, the CDMI method, and Abridged Addressing. WZE is used because it is tuned for data memory accesses and we found that it performs better than other simpler techniques such as Gray code, T0, etc. for data memory.

We notice that the CDMI, in spite of eliminating address bus switching activity during the inner loop array accesses to a large extent, gives better results than WZE in only 3 out of the 6 examples. This is primarily because the decoder overhead is significant compared to the address bus power saved. Abridged Addressing results in average power savings of 40% over WZE and 44% over CDMI. The difference is significant in examples such as SOR with a large number of array accesses in the inner loop because in such cases, CDMI maintains a register file with many registers, thereby incurring a larger overhead due to the register accesses. We found that the addressing logic occupied, on an average, almost 65% less area than the CDMI decoder. This shows that, although there may be some extra adders in the addressing logic, the area saved due to reduced registers is more significant. The details are omitted due to lack of space. There was no performance overhead because of the reasons described in Section V-A.

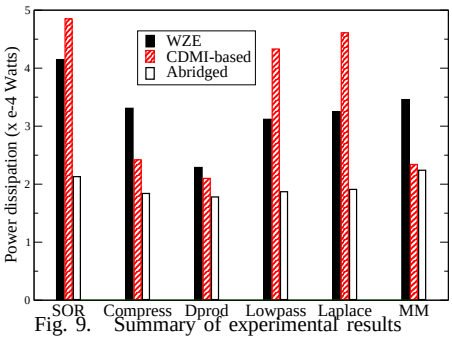

Fig. 9. Summary of experimental results

In our second experiment, we observed the variation of the power dissipation due to changes in the address bus capacitance for one of the examples (SOR). In the power simulation, the bus capacitances were set to different values to simulate longer wires. As the wire capacitance increases, we expect that the decoder overhead in techniques such as CDMI will be a relatively smaller fraction of the address bus power. This variation is observed in the comparison of WZE and CDMI curves in Figure 10. Abridged Addressing performs well in comparison, because on one hand it uses the same strategy as CDMI to reduce the address bus power, and on the other, it uses a simpler addressing logic that results in lower overall power. Lower capacitance values in Figure 10 correspond to on-chip memory, and higher capacitances correspond to either off-chip memory, or long on-chip address buses. Abridged Addressing performs well throughout the range because the technique results in just a logical transfer of the addressing computation from the datapath into the memory subsystem; there is very little extra encoding and decoding involved. This makes the technique suitable for both on-chip and off-chip memor

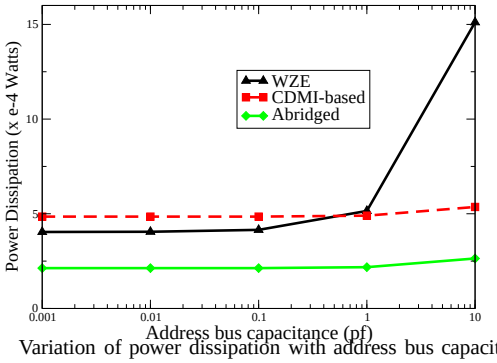

Fig. 10. Variation of power dissipation with address bus capacitance

## VII. Conclusion

We presented Abridged Addressing, a strategy for generating efficient memory addressing circuits that minimize power in both the address buses and the addressing logic. Instead of updating address registers during each array access, we maintain only one address register throughout the loop execution and generate the other addresses through simple arithmetic computations off this register, which minimizes power-expensive register file accesses. Address bus switching is minimized by generating the addresses in the memory subsystem itself. Experimental results indicate that the addressing technique is suitable for both the off-chip as well as on-chip memory because there is very little decoding overhead. Future work in this direction includes handling multiple memory modules in a design, and analyzing the impact of static power.

## References


[1] L. Benini and G. De Micheli, "System level power optimization: Techniques and tools," *ACM TODAES*, Apr. 2000.

[2] F. Catthoor et al., *Custom Memory Management Methodology*, Kluwer Academic Publishers, 1998.

[3] M. R. Stan and W. P. Burleson, "Bus-invert coding for low power I/O," *IEEE TVLSI*, Mar. 1995.

[4] C.-L. Su and A. M. Despain, "Cache design trade-offs for power and performance optimization: a case study," in *ISLPD*, 1995.

[5] L. Benini and G. de Micheli, *Dynamic power management: Design Techniques and CAD Tools*, Kluwer Acad. Publ., 1998.

[6] E. Musoll et al., "Working-zone encoding for reducing the energy in microprocessor address buses," *IEEE TVLSI*, Dec. 1998.

[7] S. Ramprasad et al., "A coding framework for low power address and data buses," *IEEE TVLSI*, July 1999.

[8] L. Benini et al., "Power optimization of core-based systems by address bus encoding," *IEEE TVLSI*, Dec. 1998.

[9] M. Mamidipaka, D. Hirschberg, and N. Dutt, "Low power address encoding using self-organizing lists," in *ISLPED*, Aug. 2001.

[10] L. Benini, A. Macii, and M. Poncino, "A recursive algorithm for low-power memory partitioning," in *ISLPED*, Aug. 2000.

[11] P. R. Panda and N. D. Dutt, "Low-power memory mapping through reducing address bus activity," *IEEE TVLSI*, Sept. 1999.

[12] C. Kulkarni, F. Catthoor, and H. De Man, "Advanced data layout organization for multi-media applications," in *PDIVM'2000*, May 2000.

[13] P. Petrov and A. Orailoglu, "Low-power data memory communication for application-specifi c embedded processors," in *ISSS*, 2002

[14] K. Muhammad and K. Roy, "A novel design methodology for high performance and low power digital fi lters," in *ICCAD*, Nov. 1999.

[15] E. Lawler, *Combinatorial optimization: networks and matroids*, Saunders College Publishing, Cambridge, MA, 1976.