Address Generation for Memories Containing Multiple Arrays

Herman Schmit and Donald E. Thomas
Department of ECE
Carnegie Mellon University, Pittsburgh, PA

Abstract

This paper presents techniques for generating addresses for memories containing multiple arrays. Because these techniques rely on the inversion or rearrangement of address bits, they are faster and require less hardware to compute than offset addition. Use of these techniques can decrease effective access time to arrays and reduce address generation hardware. The primary drawback is that extra memory space is occasionally required by these techniques, but this extra memory space is on average only 4% and no worse than 25.2% of the utilized memory space. This amount of wasted address space is less than the amount required by similar techniques [1].

1.0 Introduction

Many behavioral synthesis systems allow specifications that include arrays. One task of the synthesis system should be to intelligently map the arrays in the behavior to physical memories in the implementation of that behavior. Recent research [1][2] has explored the concept of "clustering" arrays into physical memories in the implementation. The clustering of arrays requires that each element of each array has a unique address in the memory. Therefore, each array must be assigned to an address space in the memory that does not intersect with the address space of any other array in the memory. Traditionally, disjoint address spaces have been created by adding offsets to the indices of the arrays. An addition operation may require the allocation of more hardware and may slow the effective access time. This paper presents three alternative address generation techniques generate disjoint address spaces using only the inversion or rearrangement of address bits, which are fast and inexpensive operations. The effectiveness of these techniques is demonstrated for groups of two or more arrays.

The performance gained by these techniques depends on the time required by an addition operation as compared to the time required to perform a memory access. This ratio varies greatly according to the technologies used for memory and data path. In this paper, it will be assumed that the addition of an offset before a memory access would degrade performance to unacceptable levels.

The concept of clustering different arrays into memories was introduced by Ramachandran [2], who adopted offset addition for synthesis. To reduce the time required to perform an addition, Karchmer [1] rounded array sizes up to the nearest power of two, which usually wastes address space. Our techniques also reduce the time and hardware required to compute the address, but our techniques waste less address space.

The input to the problem addressed in this paper is a set of single-dimensional, zero-offset arrays (arrays with the first element at index zero) with defined dimensions. The elements of each of these input arrays must be mapped to addresses in a single memory so that no two different array elements are mapped to the same memory address. A good mapping minimizes the time required to perform the address generation functions and also minimizes the size of the memory.

2.0 Address generation for two arrays

Let **a** and **b** be zero-offset arrays with n and m elements, respectively, numbered $\mathbf{a}[0], \mathbf{a}[1], \ldots, \mathbf{a}[n-1]$, and $\mathbf{b}[0], \mathbf{b}[1], \ldots, \mathbf{b}[m-1]$. The *two-array address generation problem* is to determine two one-to-one functions, α and β, such that for any i and j with $0 \le i < n$ and $0 \le j < m$:

$$0 \le \alpha(i) < p \text{ and } 0 \le \beta(j) < p \text{ for some } p, \text{ and} \quad (1)$$

$$\alpha(i) \ne \beta(j). \quad (2)$$

Given these two functions, arrays **a** and **b** can be mapped onto a new array **c** with p elements, numbered $\mathbf{c}[0], \mathbf{c}[1], \ldots, \mathbf{c}[p-1]$, by locating $\mathbf{a}[i]$ in $\mathbf{c}[\alpha(i)]$ and locating $\mathbf{b}[j]$ in $\mathbf{c}[\beta(j)]$. The two conditions, (1) and (2), assure that the address spaces created by α and β are within the limits of array **c** and do not intersect. Because the new array **c** is also zero-offset, it can be further combined with other arrays allowing us to use our address generation techniques for groups of three or more arrays.

Address generation for both arrays can be implemented with hardware like that in Figure 1. This figure assumes there is only one address port on the memory and therefore

This research was supported by the Semiconductor Research Corporation, under contract DC-95-068, and the National Science Foundation, under contract MIP-9408457.

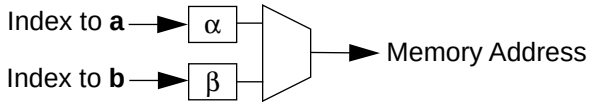

Figure 1. Address generation: two arrays

the results of the α and β functions have to be multiplexed onto a single address bus.

One solution to the two-array address generation problem uses addition to offset the address spaces:

$$\alpha(i) = i + m, \beta(j) = j \text{ or} \quad (3)$$

$$\alpha(i) = i, \beta(j) = j + n. \quad (4)$$

This solution always minimizes *p*, because it equals *n*+*m*. The following three techniques also minimize *p* (*p*=*n*+*m*) for pairs of arrays with certain size relationships and can be computed much faster and cheaper than addition.

2.1 Technique 1: Banking

When banking is used to map the address of two arrays, one array is assigned to the odd elements of memory address space, and the other is assigned to the even elements. If either $n = m$, $n = m - 1$, or $m = n - 1$, then $p = n + m$. The functions α and β for banking are:

$$\alpha(i) = 2i, \beta(j) = 2j + 1 \text{ , if } n = m \text{ or } n = m - 1, \quad (5)$$

$$\alpha(i) = 2i + 1, \beta(j) = 2j \text{ , if } n = m \text{ or } m = n - 1. \quad (6)$$

These functions can be performed with no actual logic, but with rearrangement of address wires and the propagation of constants.

2.2 Technique 2: Address bit inversion

The technique of address bit inversion is based on the following observation: If the bit-wise OR of *n* and *m*, ($n \mid m$), equals the sum of *n* and *m*, then two disjoint address spaces can be created by performing a bit-wise XOR (⊕) on the index of one array and the limit of the other array. Specifically the functions α and β are:

$$\alpha(i) = i \oplus m, \beta(j) = j \oplus n \text{ , if } n \mid m = n + m. \quad (7)$$

Two other variations of this technique also apply under slightly different conditions:

$$\alpha(i) = i \oplus m, \beta(j) = j \oplus (n-1) \text{ , if } (n\text{-}1) \mid m = n + m - 1, \quad (8)$$

$$\alpha(i) = i \oplus (m-1), \beta(j) = j \oplus n \text{ , if } n \mid (m\text{-}1) = n + m - 1. \quad (9)$$

A proof that these are solutions to the two-array address generation problem is available in [3]. It is worth noting that when array sizes are both powers of two, or both sum up to a power of two, they always meet one of the conditions required for this technique. This technique is a generalization of Karchmer's technique of rounding arrays up to the nearest power of two.

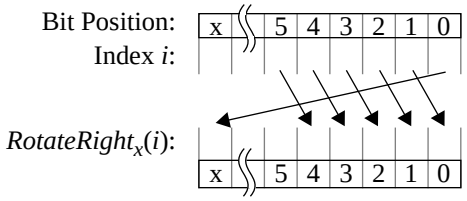

Figure 2. RightRotation

2.3 Technique 3: Rotation and inversion

To perform a right rotation, the bits of the index are shifted right one bit and the LSB, which would be shifted off, is placed in the 2^x position, for some *x* such that 2^x is greater than the maximum index of the array. This operation is illustrated in Figure 2.

This operation bisects the address space of an array, with the even elements being mapped in contiguous space starting at zero, and the odd elements starting at 2^x. If another array has $2^x - \frac{n}{2}$ elements, it can be mapped into the space between the upper and lower sectors of the bisected address space of the first array by inverting *x* of its address bits. Therefore if $\frac{n}{2} + m = 2^x$ or $\frac{m}{2} + n = 2^x$, for some *x*, array **a** and **b** can be mapped into $p=m+n$ address spaces. The functions α and β are:

$$\alpha(i) = RotateRight_x(i), \beta(j) = j \oplus \left(2^x - 1\right), \text{ if } \frac{n}{2} + m = 2^x, \quad (10)$$

$$\alpha(i) = i \oplus \left(2^x - 1\right), \beta(j) = RotateRight_x(i) \text{ , if } \frac{m}{2} + n = 2^x. \quad (11)$$

2.4 Using the techniques

Suppose that for a pair of arrays with sizes *n* and *m*, the conditions for one of the techniques are met. The same technique can then be used to generate addresses for arrays of size $n2^x$ and $m2^x$, where $x > 0$, by letting the *x* least significant index bits be unchanged by the address generation functions. Therefore, any common power of two should be divided out of *n* and *m* prior to determining whether any address generation conditions is met. For example, the banking condition is not met for $n = 10$, and $m = 8$, but if each is divided by 2, then the banking condition is met because 8/2 = 10/2 - 1. In this example, the banking is performed on all but the least significant bit of the index.

The function for determining whether *n* and *m* meet one of the above condition for mapping into *n*+*m* address locations without using offset addition is named **Conditions-Met** and shown in Figure 3. This procedure first divides out any common power of two, and then checks whether the conditions for any of the three techniques are met.

```
function ConditionsMet(n,m)
    while (LSB(n) = 0 and LSB(m) = 0)
        Shift n and m right one bit;
    if ((n = m) OR (n - 1 = m) OR (n = m- 1)) then
        return TRUE;
    if ((n | m = n + m) OR
       (n - 1 | m = n + m - 1) OR
       (n | m - 1 = n + m - 1)) then
        return TRUE;
    x := Most significant bit position of (n + m);
    if ((n / 2 + m = x) OR (n + m / 2 = x)) then
        return TRUE;
    return FALSE;
```

Figure 3. ConditionsMet Function

The **ConditionsMet** function returns TRUE for all n and m when $n+m \leq 16$. Unfortunately, as n and m grow the probability that **ConditionsMet** returns TRUE decreases. If it returns FALSE, the arrays can be enlarged by appending extra addresses so that **ConditionsMet** returns TRUE. The appended addresses are wasted, but this waste can be minimized. Figure 3, summarizes the algorithm, called **MinimalGrow**, which returns the minimal size required to map the arrays using non-additive techniques.

```
function MinimalGrow(n,m)
    if (ConditionsMet(n,m)) then
        return n+m
    i1 := 1
    while (TRUE)
        for i2 = 0 to i1
            if (ConditionsMet(n+i1-i2,m+i2)) then
                return n+m+i1
        i1 := i1 + 1
```

Figure 4. MinimalGrow Function

A series of experiments have been conducted to measure the size returned by **MinimalGrow** for pairs of arrays with random sizes uniformly distributed in the range $[1...2^z]$. Fourteen of these experiments were conducted with parameter z ranging from 2 to 15. In these experiments p is determined by calling **MinimalGrow**, and the fraction of wasted address space is computed with the equation $\frac{p}{n+m} - 1$. The average fractional waste is graphed versus the maximum array size, 2^z, in Figure 5. The sample size in these experiments was determined by the number of samples required to make the standard error fall below 10^{-4}. At z = 15, 92,000 pairs of arrays were evaluated.

The average wasted address space is zero for array sizes less than eight, and for larger arrays, the waste levels off at about 3.1% of the total. When it levels off, the standard deviation of these samples is 3.0%, which is almost as large as the average, indicating that in most of the cases the waste is between zero and 6%. The worst case waste for all array sizes tested was 18.6%. For comparison, the address space wasted by Karchmer's technique[1] on average is 37% for z = 15, and in the worst case is 99%.

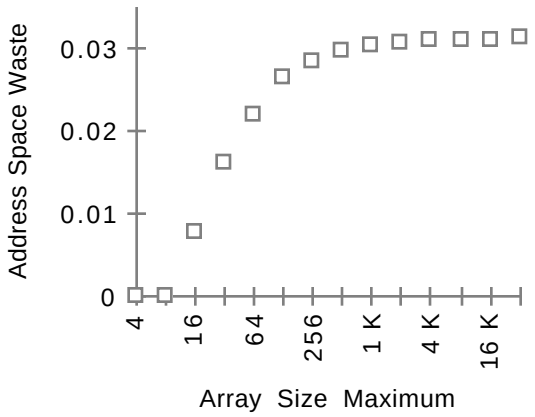

Figure 5. MinimalGrow Performance

In our experience, it is usually easier to generate addresses for arrays from real behaviors than for randomly generated arrays. Real sets of arrays frequently contain pairs of arrays with equivalent sizes, which enables the use of banking. Furthermore, real arrays frequently have sizes equal to powers of two, which enables the use of address bit inversion.

3.0 Address generation for three or more arrays

To apply these address generation techniques to groups of more than two arrays, a binary tree is constructed with the arrays as leaf nodes of the tree. The **MinimalGrow** function is used to determine the size of non-leaf nodes using the size of the children of the nodes as arguments to the function. The size of the root node of the tree determines the size of the whole group of arrays.

The problem with this approach is that the structure of the binary tree determines the resulting size of the root node. The number of possible binary trees with x leaf nodes grows exponentially with x, which makes finding the best structure very expensive for large x.

As the total number of arrays in the group increases, the optimal address space waste actually decreases, as can be seen in Figure 6. To generate this graph a set of n arrays with random sizes in the range between 1 and 2^z were generated. Every tree possible with these arrays as leaf nodes was evaluated to find the optimal root node size. For comparison, Figure 7 shows the waste of randomly generated binary trees.

As the number of arrays increases, finding the optimal tree is increasingly impractical, yet it is not acceptable to allow the address space waste to grow as in Figure 7. The

following two heuristics create and improve a tree structure so that the size of the root node is minimized.

3.1 Greedy pairing

The greedy pairing heuristic takes as input a list of arrays that need to be packed into one memory address space. In each step of the heuristic, a cost function is evaluated for every pair of arrays in the behavior. The pair with the minimal cost are paired together, and are replaced in the list of arrays by a new array, whose size is determined by the **MinimalGrow** function. This procedure is repeated until there is only one array in the set.

A simple and fairly effective cost function for this heuristic is to determine the size of the array resulting from the pairing, using **MinimalGrow**, and subtract the sum of the array sizes. This cost function equals the amount of address space waste caused by this pairing.

A more effective cost function extends this idea by looking ahead to how each pair of array can be paired with remaining arrays. To do this, the heuristic finds for every pair of arrays, **a** and **b**, the minimum address space wasted by combining the pair (**a**, **b**) with every other remaining array. This minimum waste is added to the previous cost function. If there are only three arrays, this cost function equals the address space wasted for the whole tree if the given pairing is made, and is therefore optimal for three arrays.

The performance of the greedy pairing heuristic using the look-ahead cost function has been quantified with an experiment similar to the experiments performed for the two array case. The set of points labelled “Pairing” in Figure 8, shows the average fraction of address space wasted versus the number of arrays in the group using the pairing heuristic. In this figure the maximum array size is fixed at 32K (z = 15). For comparison, Figure 8 also shows the average fraction of address space wasted for groups constructed through random pairing of arrays. The sample size for each point in this graph is at least 7800 and up to 202,000. For all points, the standard error is below $5x10^{-4}$. The standard deviation for each of these trials was approximately half of the average, indicating that most cases give results near the average.

3.2 Tree rotation

Figure 9 illustrates a subtree in a binary tree, and two possible rotations of that tree on either side. The root nodes in this figure may be the root node of the whole tree or the child of some other node, and the leaf nodes in this figure may be arrays, or may be trees themselves. The figure only shows the rotations possible if the right child of the root node has children. If the left child has children, two more mirror image rotations are also possible.

Our second heuristic works by performing these rotations on the binary tree and evaluating the impact of those rotations on the size of the root node of the subtree. This improvement on subtrees is recursively applied to the whole tree as shown in Figure 10. The **GreedyRotate** procedure is called with the root node of the tree as an argument, and all the subtrees are first improved, followed by the best rotation of the top level of the tree. A local minimum is reached by repeatedly running this procedure on the root node of the tree until its size does not improve.

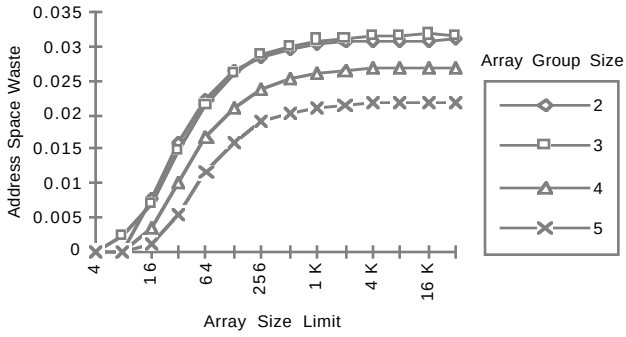

Figure 6. Optimal array grouping: Average case

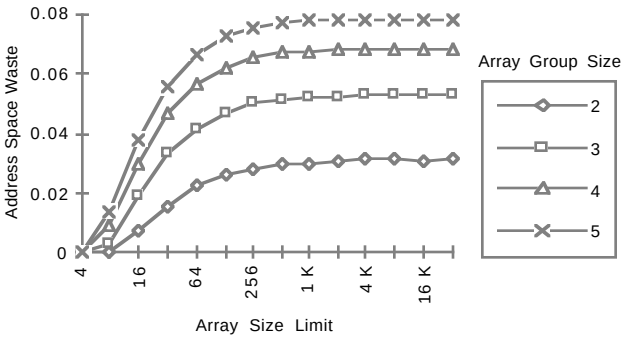

Figure 7. Random Array Grouping: Average case

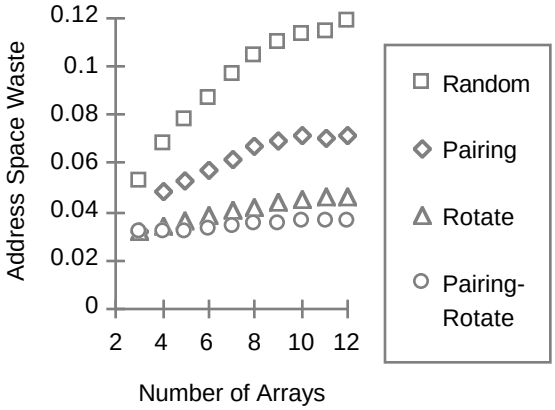

Figure 8. Address space waste

Two or three applications of this procedure are typically necessary to reach the local minimum. Since the number of nodes in a binary tree is a linear function of the number of leaf nodes in that tree, the run time for this procedure increases linearly as the number of arrays increases.

In Figure 8, the set of points labelled “Rotate” show the results of applying **GreedyRotate** repeatedly to randomly generated binary trees, and the set of points labelled “Pairing-Rotate” show of applying **GreedyRotate** to the trees generated by the greedy pairing heuristic. The combination of heuristic approaches is most effective at minimizing address space waste. With a group of twelve arrays, the address space waste using the combination of heuristics averages less than four percent of the total address space. The sample size for these trials were the same size for four sets of data in Figure 8, and the standard deviation for each trial was approximately half of the average. Using the combination of the greedy pairing and **GreedyRotate** heuristics, the worst case address space waste for over one million cases tested was 25.2%.

The **GreedyRotate** procedure essentially evaluates every 3-node subtree in the graph and improves it. For that reason it is similar to the greedy pairing construction technique. **GreedyRotate** could be improved by evaluating all 4-node or 5-node subtrees, but as the size of the evaluated subtree increases, the run time will increase exponentially. This extension is probably not worth the computational cost.

3.3 Address generation hardware

A simple hardware implementation of address generation hardware can be based on the structure of the binary tree of arrays, as shown in Figure 11. This implementation is constructed by placing a multiplexor and the appropriate α and β functions at each non-leaf node of the tree. The α and β functions can be moved past the multiplexors and combined into a single function for each array in the group, as shown on the right in Figure 11. This approach allows the multiplexor structure to be rearranged to allow

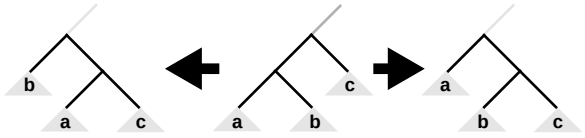

Figure 9. Tree Rotations

```
GreedyRotate(treenode)
    GreedyRotate(treenode.left)
    GreedyRotate(treenode.right)
    Choose the rotation which minimizes treenode
```

Figure 10. GreedyRotate procedure

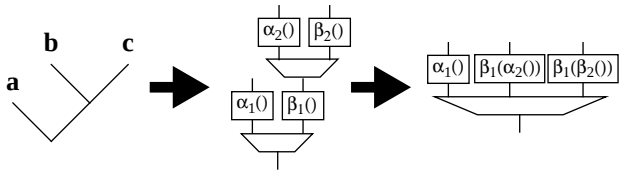

Figure 11. Address generation hardware

the fastest address path for the most time-critical array accesses. Because all α and β functions only require the inversion or rearrangement of address bits, any composition of α and β functions also only requires bit inversions and rearrangements. Therefore the computation time for a composed address generation function does not grow as the depth of the binary tree representation grows (although the time required to multiplex the addresses may grow). Composing the α and β functions will likely improve the performance of the address generation hardware by eliminating the multiple bit inversions that may occur on the path of a single address bit.

4.0 Conclusions

This paper described three techniques that can be used to map arrays to disjoint address spaces. When compared with the use of addition for address space mapping, these techniques require much less hardware and can be computed in as little as a single gate delay. The average amount of extra address space required by these techniques is less than 4%, and the worst case measured was 25.2%, which is superior to the only known published technique [1]. These techniques increase the address generation alternatives available for the design of application specific memory sub-systems, and are useful for both human designers and synthesis tools.

5.0 References

[1] D. Karchmer, and J. Rose, “Definition and Solution of the Memory Packing Problem,” *Proc. of ICCAD*, pp.53-58, San Jose, CA, Nov. 1994.

[2] L. Ramachandran, D. D. Gajski, and V. Chaiyakul, “An Algorithm for Array Variable Clustering,” *Proc. of European Design and Test Conference (EDAC)*, 1994.

[3] H. Schmit and D. Thomas, *Array Mapping for Behavioral Synthesis*, Technical Report, CMU-CAD 94-46. Carnegie Mellon University, Pittsburgh, PA, 1994.