An Algorithm for Allocation of Multiport Memories Using a New Layout Memory Model

Loganath Ramachandran
Daniel D. Gajski
Viraphol Chaiyakul

Technical Report #93-13

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

ramachan@ics.uci.edu

Abstract

Array variables are extensively used in many behavioral descriptions especially for digital and image processing applications. During synthesis, these array variables are implemented with memory modules. In this report, we show that simple one-to-one mapping between the array variables and the memory modules lead to inefficient designs. We propose a new algorithm (MeSA) for efficient allocation and mapping of array variables onto memory modules. MeSA computes, (a) the number of memory modules required, (b) the size of each module (c) the number of ports on each module and (d) and the grouping of array variables that map onto each memory module. It also considers the effects of address translations that are required when two or more array variables are stored in one memory module. While, most previous research efforts have concentrated on optimizing the scalar variables, the primary focus in this report is deriving efficient storage mechanisms for array variables. We show the efficiency of our technique on some standard benchmarks.

Contents

List of Figures

1 Introduction

High Level Synthesis [1, 2] consists of automatically synthesizing hardware from a given behavioral description. Many behavioral descriptions that manipulate large amounts of data use **array variables** to represent data storages. During synthesis these array variables are generally implemented with memory modules. However, many synthesis systems either do not handle such array variables, or at best store each array variable in a separate memory module.

In this paper, we propose a new Memory Synthesis Algorithm, (MeSA) that derives an efficient allocation of memory modules for storing the array variables given in the description. The allocation derived by MeSA will provide: (a) the number of memory modules (b) the size of each module and (c) and the number of ports on each module to satisfy the required performance. The important contribution of MeSA is **array-variable clustering**, whereby one or more array variables gets stored in the same memory module based on cost and performance considerations.

Let us illustrate the advantages of array-variable clustering using a very simple example. In Figure 1(a), a one-line description containing two array variables A and B of size 1000 x 8 is shown. The results of synthesis using a simple memory allocation scheme is shown in Figure 1(b). Here, each array variable is stored in a separate memory module. The size of these memory modules correspond directly to the size of the array variables. If we assume that A[i] and A[j] can be accessed in the same state, MemA would require 2 ports and MemB would require 1 port. The scalar variables i, j and k are stored in registers which are connected to the address ports of the memory.

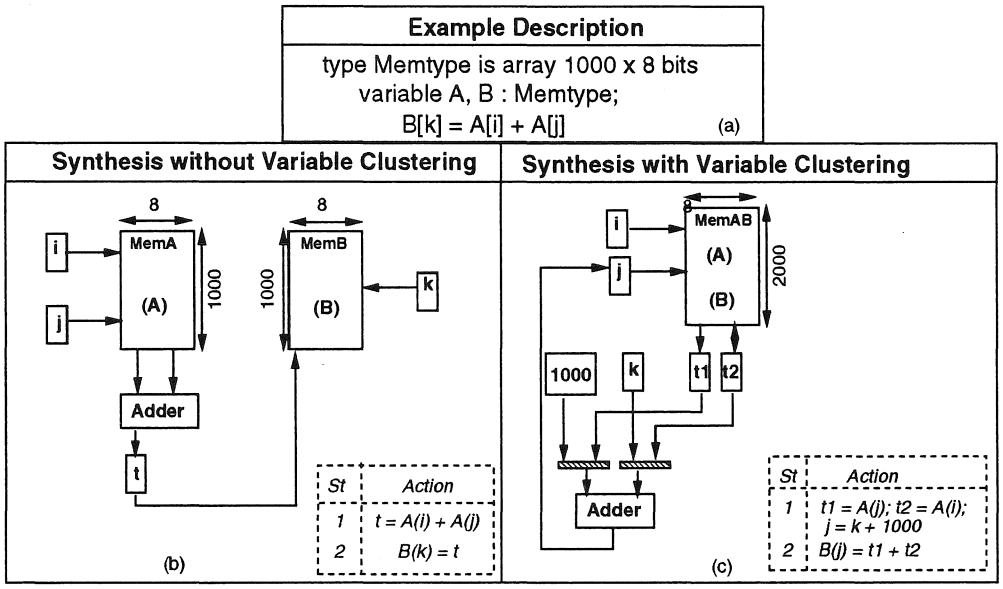


Figure 1: Array-Variable Clustering Example

On the other hand it is possible to store both the array variables A and B in the same memory module (Figure 1(c)). Now, all accesses to variable B require an address translation. Since the adder is also shared for address translation, additional muxes are required in the design. The important characteristics of this design can now be studied in further detail:

- **Size of the memory modules** - Since many variables are stored in the same memory, large memory modules are required. The use of larger memories may lead to increased access delays, but could lead to reduction in area since decoding and sensing circuitry are shared.
- **Number of memory modules** - Fewer memory modules are required because of variable sharing. This would lower the number of nets required to be routed, possibly resulting in smaller designs.
- **Number of ports** - The total number of ports decreases because of efficient port sharing. In our example the total number of ports decreased from 3 to 2. This would reduce the total number of address lines required, thereby reducing the overall routing area.
- **Address Translation operations** - One of the important drawbacks of storing two array variables in the same memory module is *address translation* requirements. Any reference to the variable B (say $B[k]$) will have to translated to $B[k + 1000]$, resulting in extra add operations which could decrease the performance of the design. Additional hardware (like multiplexers) is required to perform these new operations.
- **Ordering of Variables** - Since address translations are required only for variables stored with an offset, the ordering of variables plays an important factor. In Figure 1(c), one translation was required for B[k]. Ordering A after B in the memory would result in two address translations, possibly degrading the performance further.

All the above effects are taken into account by MeSA when it performs hierarchical clustering of array variables. It assumes an initial configuration, wherein each array variable is stored in a separate memory module. All possible merges are considered and a new configuration is created by clustering two variables that results in the least cost. The cost of a particular merge depends on the actual layout costs of the memory modules and also on the performance (i.e., the number of clock cycles required to schedule all the operations and address translations). In order to estimate the layout costs for a particular configuration we have also developed an extensive layout model for memories. The details of this model are shown in Section 5.

The rest of the report is organized as follows. In the next section, we provide some of the previous work in memory and register allocation and explain how MeSA is different from such previous approaches. The representation scheme for the CDFG is shown in Section 3 and the details of the algorithm are shown in Section 4. Finally we present the results of our algorithm on some examples and present conclusions.

2 Previous Work

Many optimization techniques have been proposed in the literature for scalar variables. [3, 4, 5, 6] presented algorithms for reducing the number of registers by storing variables with non-overlapping lifetimes into the same register. In [3], this was formulated as a clique partition problem on a graph formed according to overlapping lifetimes. Kurdahi and Parker [4] solved the register allocation problem by using the left edge algorithm also used for the minimization of the tracks in standard cell layout. Stok [5] used the edge coloring algorithm to group the variables into registers.

Another group of research papers have focussed on reducing the wiring area of the design, by storing variables into regular structures like register files or n-port memories instead of distributed registers. This is achieved by mapping all the scalar variables on a *scheduled flowgraph* into a minimal set of register files based on access patterns of the variables. In [7], this problem was formulated as a 0-1 integer linear programming problem. After variables are assigned to individual registers, a large register file is formed by grouping the registers into a multiport memory. This process is repeated till all the registers are grouped. In [8], the minimum number of multiport memories was derived by dividing the variables into groups using a 0-1 ILP formulation. [9] focussed on minimizing the number of interconnections, when minimizing the number of registers.

These above approaches have two important disadvantages: (i) These were formulated mainly for scalar variables. They cannot be directly applied to array variables, unless each array variable is broken down into its individual components. This would be impractical since the number of components in an array variable could be extremely large. (ii) Most of these approaches were intended to be used on a scheduled flowgraph. Since scheduling is done prior to and independent of the memory allocation phase these approaches would lead to an inefficient solution. As an example, the schedule may not control the number of the number of memory accesses in each state resulting in a large bandwidth for memory accesses. Our approach overcomes these disadvantages, since it is specifically formulated for array variables. Also, in MeSA the allocation of the memory modules is done prior to scheduling, taking into account the possible bounds within which data will be accessed. This would lead to more efficient solution.

In [10, 11, 12] efficient algorithms are proposed to analyse data streams and minimize the storage requirements for each stream. Such algorithms are required when the descriptions are written with an applicative language. Each variable in such a description represents an infinite stream of data and it is necessary to compute the exact size of storage required for each stream. However, after computing the minimum storage requirements MeSA could be used to compute an efficient memory allocation to store the variables.

3 Internal Representation

In this section we provide a scheme for representing flowgraphs containing array variable accesses. Two special dataflow node types *READ_MEM* and *WRITE_MEM* are used, similar to the representation scheme presented in [13].

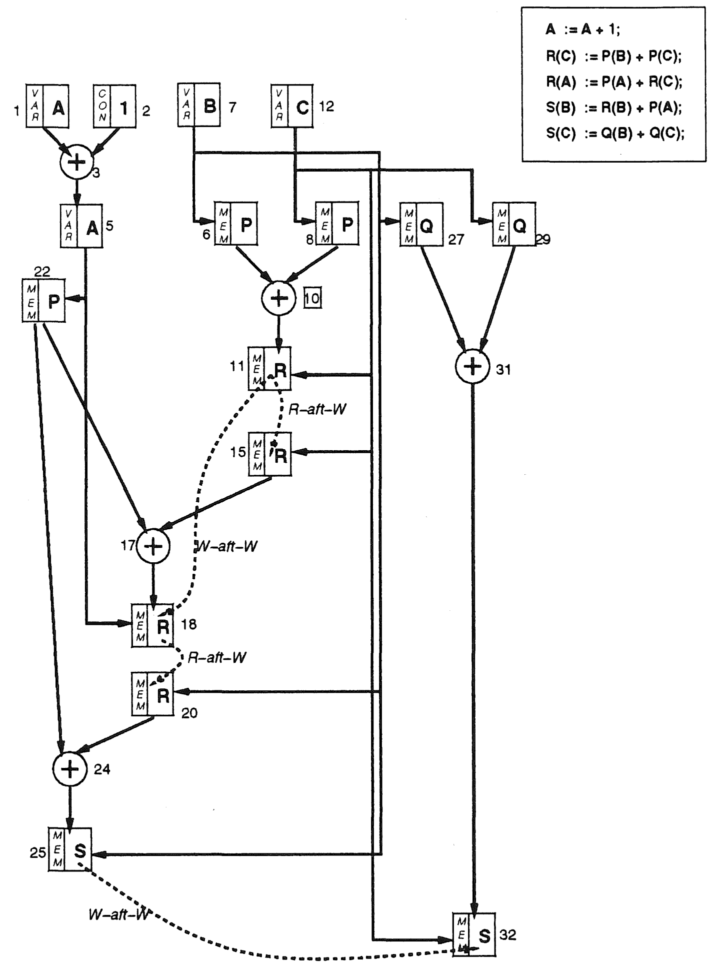
The *READ_MEM* node contains one input and output. The input line gets the index values which determines the location that is being read. The actual data is available on the output line. The *WRITE_MEM* node contains two input lines, one for the index value and the other for the actual data that is to be written. When the particular *WRITE_MEM* node is executed the value on the data line is written to the location pointed to by the index value.

In addition to these new dataflow nodes, the representation contains three different types of dependency arcs. These arcs are:

- *R-aft-W arcs:* If a particular memory location is written to in statement (st_i) and read from in statement (st_j) where $j > i$, then a Read-after-Write dependency exists. This dependency forces the scheduling algorithm to schedule st_j only after scheduling st_i, in order to ensure that the new value of the memory is used. This is represented by a *R-aft-W dependency arc.*
- *W-aft-W arcs:* If two statements (st_i, st_j) update a particular array, then the right order must be maintained during synthesis. This is represented by a *W-aft-W dependency arc* and it ensures that the scheduling algorithm maintains the order between the statements.
- *W-aft-R arcs:* If a particular memory location is read in statement (st_i) and updated in statement (st_j) where $j > i$, then a Write-after-Read dependency exists. This dependency forces the scheduling algorithm to schedule st_j only after scheduling st_i, in order to ensure that the new value is not written till the old value is read. This is represented by a *W-aft-R dependency arc.*

In Figure 2 we show the CDFG for a simple for example. Each of the twelve array access is represented by a READ_MEM or a WRITE_MEM node. The integer or constant access are represented by a READ_VAR or a READ_CONST node. Four dependency arcs are shown by dotted arcs, while the actual data transfer is shown as bold arcs. The figure also shows the id of each node in the flowgraph.

We use a two step algorithm to create the dependency arcs. When the dataflow graph is created we take a pessimistic view of the dependencies. Hence dependency arcs are created between the READ_MEM and WRITE_MEM of the same variable if the index value is not a constant. (In case the indices are constants, arcs are created only if the constants are equal). In the second phase, standard compiler optimization techniques are used to delete some of the arcs. Some typical techniques are explained in [14, 13]. As a simple example an arc connecting two WRITE_MEM nodes can be deleted



```
A    := A + 1;
R(C) := P(B) + P(C);
R(A) := P(A) + R(C);
S(B) := R(B) + P(A);
S(C) := Q(B) + Q(C);
```

Figure 2: Representation Scheme

```
Notation:
    Let G = (N, E) be the CDFG consisting of a set of nodes N
        and a set of edges E
    Let $V = \{v_1, v_2, v_3 ... v_m\}$ be the set of 'm' array variables.
    Let $A_{v_i} = \{v_i1, v_i2, ...\}$ be the set of array access nodes for $v_i$.
    Let $\tau = \{g_1, g_2 .. g_m\}$ be a set of groups, where each group $g_i$ represents the cluster of
        variables (i.e., $\{v_i, v_j, v_k ..\}$).

Algorithm Outline: MeSA.
01.    Assume an initial configuration $\tau = \{\{v_1\}, \{v_2\}, .... \{v_m\}\}$, where
         each variable is stored separately. Let 'm' be number of array variables.

02.    Create a table (MAFT) with 'm' columns each column representing one group of variables.
03.    foreach array access node $v_{i_j}$ in $v_1, v_2 ... v_m$
04.        Compute the earliest and latest access bounds $(E(v_{i_j}), L(v_{i_j}))$.
05.        Compute the access probabilities for each memory module and update MAFT.
06.    end for
07.    $ImpCost$ = Compute the implementation cost

08.    Create a completely connected closeness C graph with 'm' vertices.
09.    foreach edge $(e_{i,j})$ connecting variables $g_i, g_j$ in C
10.        Create new CDFG with required address translations after merging $g_i$ and $g_j$ ;
11.        Repeat steps 2 to 7 on the new CDFG to compute new cost NewCost
12.        Update edge weight $((e_{i,j}))$ = NewCost.
13.    end for

14.    Cluster groups $g_i$ and $g_j$ if Cost(edge) is minimum and update the cluster growth tree.
15.    Create a new configuration with variables in groups $g_i$ and $g_j$ merged together and repeat
         steps 2 to 14 with one less group
```

Figure 3: MeSA Algorithm

if the indices are 'k' and 'k+1' since we can be sure that these two indices will not be equal. We will not delve further into these optimization techniques since they have been well documented in the current literature.

4 Memory Synthesis Algorithm

4.1 Overview

The main goal of MeSA can be stated as: **Given a behavioral description and a set of resource and performance constraints determine the memory configuration that would result in the least area for the design.**

```
Algorithm 1.0: FuConstrainedAsap Scheduling.
  InsertReadyOps(V, $RList$);
  Cstep = 0;
  while($RList \neq \phi$)do
    Cstep = Cstep + 1;
    ScheduleOps($RList$, $N_{fu}$, Cstep);
    ScheduleArrayAccessNodes($RList$, Cstep);
    InsertReadyOps(V, $RList$ );
  endwhile
```

Figure 4: FuConstrainedAsap Algorithm

MeSA is based on a hierarchical clustering approach similar to [15]. Initially, each variable in the description is assigned to a separate group. Each group is assumed to be mapped to a single memory module. A closeness graph is created where each node represents a group and each edge weight represents the *implementation cost* if the two groups were to be merged. The *implementation cost* is a resultant of two factors: (a) the layout cost for the memory modules and (b) the performance cost which measures the performance degradation because of merging two variable groups into a single memory. After all the edge weights are computed, two groups with the lowest closeness factors are merged.

A cluster growth graph is updated after each merge operation. Finally MeSA selects the clustering level (based on performance criteria) that has the least implementation cost, and allocates memory modules based on the selected variable groups at that level. The details of the algorithm are shown below in Figure 3. The rest of the section will explain the important steps of the algorithm in further detail.

In the algorithm explained in Figure 3, let $V = \{v_1, v_2, v_3 ... v_m\}$ be the set of 'm' array variables in the description. The description is represented by a CDFG consisting of nodes that represent operations and edges that represent dependency between operations. The operations could be of 2 types: (a) arithmetic or logic operations (b) read and write operations of variables. Read and write operations of array variables are represented by array-access nodes in the CDFG. Let $A_{v_i} = \{v_{i_1}, v_{i_2}, ...\}$ be the set of array-access nodes for the array variable v_i.

4.2 Initial Configuration

Let g_i represent a group of variables $g_i = \{v_k, v_l, v_m ...\}$. Since all the variables belong to exactly one group during any iteration of the algorithm we can state $\forall_{i \neq j} : g_i \cap g_j = \phi$ and $\exists_k \mid v_i \in g_k$

We define a configuration, τ, as a grouping of all the array variables given in the description, ($\tau = \{g_1, g_2 .. g_m\}$). In the initial configuration, each variable is mapped to a separate group. The initial configuration τ_0 can be represented as: $\tau_0 = \{\{v_1\}, \{v_2\}, \{v_3\}, ..\{v_m\}\}$.

```
Algorithm 1.0: FuConstrainedAlap Scheduling.
  InsertReadyOps(V, $RList$);
  Cstep = MaxCstep;
  while($RList \neq \phi$)do
    Cstep = Cstep - 1;
    ScheduleOps($RList$, $N_{fu}$, Cstep);
    ScheduleArrayAccessNodes($RList$, Cstep);
    InsertReadyOps(V, $RList$ );
  endwhile
```

Figure 5: FuConstrainedAlap Algorithm

4.3 Earliest and Latest bounds

For all the nodes in the CDFG we compute the earliest (E) and the latest bounds (L) that they can be scheduled into. Specifically for the array access nodes v_{i_j}, we define $E(v_{i_j})$ and $L(v_{i_j})$ as the earliest and latest bound for accessing the variable from the memory.

In order to compute the earliest bound $E(v_{i_j})$ for each access node v_{i_j}, we define a new procedure called the **FuConstrainedAsap** scheduling algorithm. This algorithm is a list-based scheduling algorithm which handles the 'operator' nodes and the array access nodes differently. It schedules the 'ready' operator nodes only when a functional unit is available but it schedules all array access nodes as soon as they become ready. The results of the *FuConstrainedAsap* algorithm ($E(v_{i_j})$) indicates the *earliest state* into which the array access node v_{i_j} can be scheduled, under the specified functional unit constraints. Hence the name *FuConstrainedAsap* algorithm.

The FuConstrainedAsap scheduling algorithm (Figure 4) maintains a priority list ($RList$) of ready nodes (similar to [16]). A ready node is a node on the CDFG that has all predecessors already scheduled. The priority list is always sorted with respect to a priority function (i.e., Mobility). During each iteration the function *ScheduleOps* scans the ready list and schedules all the 'operator' nodes for which functional units are available. Read and Write nodes of the array variables are scheduled by the function *ScheduleArrayAccessNodes* as soon as they get onto the $RList$. Scheduling an operation may make some other non-ready operations ready. These operations are inserted into the list according to the priority function.

In order to compute the latest bound for each access node, we define another new procedure called the **FuConstrainedAlap** scheduling algorithm (Figure 5), which is similar to the FuConstrainedAsap Algorithm. The results of the *FuConstrainedAsap* algorithm ($L(v_{i_j})$) indicates the *earliest state* into which the array access node v_{i_j} can be scheduled, under the specified functional unit constraints.

```
Algorithm: ComputeAccessProbabilites.
   foreach array variable $v_i$;
      Determine the group $g_k$ that the $v_i$ belongs to;
      foreach array access node $v_{i_j}$ of variable $v_i$;
         Let p = $E(v_{i_j})$;
         Let q = $L(v_{i_j})$,
         foreach index between $p$ and $q$;
            MAFT(i, index) = MAFT(i, index) + 1(q-p+1);
         end for;
      end for;
   end for ;
```

Figure 6: ComputeAccessProbabilities

4.4 Access Probabilities

Let us first define *Access Flexibility* of a data access node(i.e., $F(v_{i_j})$), as the difference between its $L(v_{i_j})$ and $E(v_{i_j})$ values. The probability that an access will occur between the bounds is assumed to be uniform. Therefore the probability that a node v_{i_j} will be accessed in any of the states between $E(v_{i_j})$ and $L(v_{i_j})$ is $1/F(v_{i_j})$.

The details of computing the access probabilities is shown in Figure 6. The access probabilities are collated in a table called the Memory Access Flexibility Table (MAFT) which contains one column for each variable group and one row for each possible state. Each entry in the MAFT indicates the *expected number of accesses* of a variable belonging to that group during that state. For example if MAFT(2,3) has a value of '2', then this implies that two variables belonging to group g_2 will be accessed during state 3.

The maximum entry in the kth column determines the number of ports on memory module M_k (represented as P_{M_k}). The number of words W_{M_k} and the bitwidth B_{M_k} can be derived from the sizes of all the variables in group g_k.

4.5 Implementation Cost

The implementation cost is computed as a resultant of two different cost measures (a) the *LayoutArea* cost (b) the *PerformancePenalty* cost. The *LayoutArea* cost reflects the estimated area of the allocated memory modules, and the *PerformancePenalty* cost reflects the violation of the performance constraint. The overall cost function can be expressed as

$$ImpCost = LayoutArea + K * PerformancePenalty$$

An extensive memory model based on layout characteristics is provided in Section 5. This model can be used for the estimation of $LayoutArea$ costs. This model is derived in terms of three technology dependent parameters and three technology independent parameters. The technology dependent parameters include: (a) Transistor Pitch, α, which is equal to the minimal distance between two transistors, (b) Track Pitch, β, which is equal to the minimal distance between two routing tracks, and (c) Cell Height, γ. The technology independent parameters are (a) the number of memory modules, (b) the size of each memory module and the (c) the number of ports in each memory module.

The $PerformancePenalty$ is the penalty added to the cost for violating the performance constraints. If after merging two variables and taking care of the address translations imposed by such a merge, the design meets the performance constraint this $PerformancePenalty$ cost is zero. Otherwise it is the difference between the execution time of the design and the imposed performance constraint.

The weight K can be controlled by the user and provides the relative importance of area vs performance. It should be set to a very high value if the performance constraints are to be met always. The normalization factor for the layout and the performance costs are also incorporated into this weight.

4.6 Cluster Growth

A closeness graph keeps track of the costs of merging two groups of variables. Initially, this graph contains 'm' nodes one for each variable group. The weight on each edge represents the implementation costs if the two groups are merged.

Address translations have to be considered when computing the implementation costs after all the variables in groups g_i and g_j are grouped into g_{new}. In order to minimize the number of translations the variables in the group are sorted based on the decreasing order of access frequencies. As an example if group g_{new} contains variables v_1, v_2 and v_3 in the decreasing order of access frequencies, then all accesses of $v_2[j]$ are replaced by $v_{new}[j + W_{v_1}]$, and all accesses of $v_3[j]$ are replaced by $v_{new}[j + W_{v_1} + W_{v_2}]$.

The edge with the least cost is selected and the variables are clustered together, and the whole process is repeated till a single large cluster is formed. While the clustering process continues, a cluster growth tree is maintained. Finally the lowest cost clustering level that meets the performance constraints is chosen from the tree. This indicates the memory allocation.

5 Layout Memory Model

The layout memory model is derived in terms of three technology dependent parameters and three technology independent parameters. The technology dependent parameters include: (a) Transistor Pitch, α, which is equal to the minimal distance between two transistors, (b) Track Pitch, β, which is equal to the minimal distance between two routing tracks, and (c) Cell Height, γ. The technology independent parameters are (a) the number of memory modules, (b) the size of each memory module and the (c) the number of ports in each memory module.

Let M be the set of memory modules that will be used in a particular implementation. (i.e., $M = \{M_1, M_2, `...M_m\}$). For each of these memory modules let us assume that N_i, B_i and P_i are also specified where N_i is the number of words, B_i is the number of bits/word and P_i is the number of ports in memory module M_i. The implementation cost for this set of memory modules consists of: (a) Cost of the Memory Modules itself , and (b) Cost of the interconnect for the nets connecting these memory modules to the rest of the datapath. This is given by the equation:

$$TotalMemoryCost = TotalAreaOfMemoryModules + TotalInterconnectArea$$

5.1 Area of Memory Module

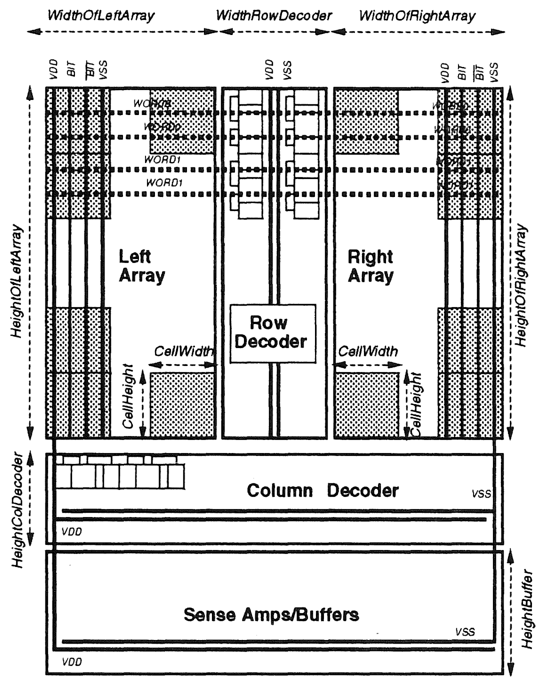


Figure 7: Memory Organization

A typical memory module consists of four distinct submodules as shown in Figure 7. The circuits for the submodules are similar to those in [17].

- *Array of Cells*, consists of a rectangular array of cells, which are used for storing the actual bit values. This rectangular array is generally shaped as a square in order to minimize the access delay. Since this array forms the most significant area of the memory module, a custom layout methodology is generally used for its implementation.

- *Row Decoder*, selects one of the rows after decoding a subset of the address lines. In order to avoid the delays associated with long polysilicon lines that run from the row decoder to each cell row, the *Row Decoder* is generally placed in the center of the array
- *Column Decoder*, selects a set of columns by decoding the remaining address lines. The data in these selected columns are output on the data lines.
- *Buffers/Amp* which enable the memory to drive reasonable loads.

5.1.1 Array of Cells

The cell array consists of a rectangular array of memory cells. Each memory cell consists of two crosscoupled inverters called the core [1]. The core is connected to the BIT and $\overline{BIT}$ by two pass transistors. For a write operation the DATA is placed on the BIT line and $\overline{DATA}$ is placed on the $\overline{BIT}$ line. The word line is then asserted to write the data into the cell. For a read operation the BIT and the $\overline{BIT}$ lines are precharged and then the word line is asserted. One of these two lines gets discharged depending on the contents of the cell. The layout for the cell, are shown in Figure 8. The innermost section of the layout (shown with dotted lines), is the core of the cell where the data is stored and the outer transistors connect the core to the appropriate BIT lines.

In order to extend the cell design for multiported RAMS two additional transistors are required. These connect the core memory to two new BIT and $\overline{BIT}$ lines, which are accessed through the second port.

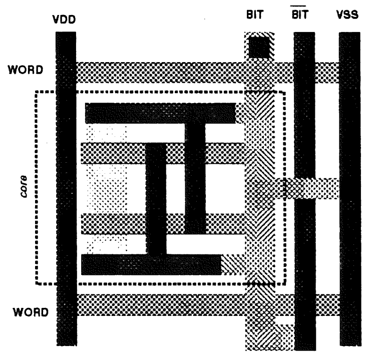


Figure 8: Cell layouts

It is clear from Figure 8 that the memory cell occupies a width of 6 tracks for the core and 2 additional tracks for each BIT and $\overline{BIT}$ lines. We therefore have the equation, $CellWidth = (6 + 2 * P_i) * \beta$. Similarly the height of the cell consists of 2 transistor pitches for the core and 2 extra pitches for the additional transistors that would be required for each additional port. Therefore the

[1]We have assumed a SRAM memory module. Similar models can be derived for other memory styles

height of the cell is determined by: $CellHeigth = (2 + 2 * P_i) * \alpha$. Since the number of cells in each column is $\sqrt{N_i * B_i}$ we have:

$$HeightofRightArray = HeightofLeftArray = \sqrt{N_i * B_i} * (2 + 2 * P_i) * \alpha$$

$$WidthofRightArray = WidthofLeftArray = (\sqrt{N_i * B_i})/2 * (6 + 2 * P_i) * \beta$$

5.1.2 RowDecoder

The Row Decoder decodes some of the address lines and provides signals that activate an entire row of cells. One n-input-AND gate is required for each row of cells where 'n' is the number of address lines.

Since there are $\sqrt{N_i B_i}$ rows, the number of address lines is $\lceil log_2(\sqrt{N_i B_i}) \rceil$. Each AND-gate would have $\lceil log_2(\sqrt{N_i B_i}) \rceil$ inputs one from each address line. Thus the number of transistors for each row of the decoder is:

$$NumTrans(perport) = (2 * \lceil log_2(\sqrt{N_i B_i}) \rceil)$$

Since the memory has P_i ports we require P_i such row decoders for each row of cells. The total number of transistors is given by

$$NumTrans(RowDecoder) = 2P_i \lceil log_2(\sqrt{N_i B_i}) \rceil$$

The layout estimate is based on a standard cell layout methodology similar to some of the earlier approaches [18, 19], where the number of standard cell strips and the number of tracks are estimated. As shown in Figure 7, the RowDecoder strips are layed out in a vertical fashion. The total number of strips that is required is given by: $NumStrips(RowDecoder) = \lceil NumTrans(RowDecoder) * \alpha / CellHeight \rceil$. This can be rewritten as:

$$Numstrips(RowDecoder) = \lceil P_i * (2 * \lceil log_2(\sqrt{N_i B_i}) \rceil)/(2 + 2 * P_i) \rceil$$

The address lines, the power and ground lines would have to be routed throughout the stretch of the row decoder. In addition we estimate that internal routing of the cells inside the strip would require one additional track. Thus the total number of tracks required is determined by the following.

$$NumTracks(RowDecoder) = \lceil NumStrips(RowDecoder) + 2 + \lceil log_2(\sqrt{N_i B_i}) \rceil \rceil$$

The total width of the RowDecoder is given by:

$$WidthRowDecoder = NumTracks(RowDecoder) * \beta + NumStrips(RowDecoder) * \gamma$$

5.1.3 ColumnDecoder

The Column Decoder is very similar to the Row Decoder described above. It decodes the remaining address lines and selects the columns of data that are to be output on the data lines. One N-input-OR gate, an inverter and a transmission gate are required for each column of cells where 'N' is the number of column address lines.

The number of address lines is $\lceil log_2(N_i)\rceil - \lceil log_2(\sqrt{(N_iB_i)})\rceil$. Each or-gate would have $\lceil log_2(N_i)\rceil - \lceil log_2(\sqrt{(N_iB_i)})\rceil$ inputs, one from each address line. Thus for each column decoder we have

$$NumTrans(perport) = 2(\lceil log_2(N_i)\rceil - \lceil log_2(\sqrt{(N_iB_i)})\rceil + 3)$$

Since the memory has P_i ports we require P_i such row decoders for each row of cells. The total number of transistors is given by

$$NumTrans(ColDecoder) = 2P_i(\lceil log_2(N_i)\rceil - \lceil log_2(\sqrt{(N_iB_i)})\rceil + 3)$$

In order to compute the number of strips of standard cells and the number of tracks required, we assume that the strips are layed out in a horizontal fashion, and the width of each strip is equal to the CellWidth. Thus the number of strips is given by: $NumStrips = NumTrans * \alpha / CellWidth$. This can be rewritten as:

$$NumStrips(ColDecoder) = \lceil 2\alpha P_i(\lceil log_2(N_i)\rceil - \lceil log_2(\sqrt{N_iB_i})\rceil + 3)/((6 + 2P_i)\beta)\rceil$$

The address lines, the power and ground lines would have to be routed throughout the stretch of the column decoder. In addition we estimate that internal routing of the cells inside the strip would require one additional track. Thus the total number of tracks required is determined by the following.

$$NumTracks(ColDecoder) = NumStrips(ColDecoder) + 2 + \lceil log_2(n_i)\rceil - \lceil log_2(\sqrt{N_iB_i}\rceil$$

The total height of the ColDecoder is given by:

$$HeightColDecoder = NumTracks(ColDecoder) * \beta + NumStrips(ColDecoder) * \gamma$$

5.1.4 Buffers

In addition to the three important components of the RAM model, we have to estimate the size of the buffers and the sense amps that would be required. For an efficient design, the size of the buffers must correlate to the actual loads driven by the RAM cell. Since this is impossible to determine before the synthesis, we assume that the buffers are of fixed size. In our model, we assume that the height of the buffer is 12 tracks.

$$HeighthBuffer = 12 * \beta$$

5.1.5 Area of Memory Modules

In summary the total height of the memory module (i.e., H_i) is given by:

$$H_i = HeightOfLeftArray + HeightColDecoder + HeightBuffer$$

Similarly the total width of the memory module is given by

$$W_i = 2 * WidthOfLeftArray + WidthRowDecoder$$

After computing the individuals (H_i and W_i) we can now determine the area occupied by the memory modules.

$$TotalAreaOfMemoryModules = \sum_{i=1}^{m} W_i * H_i$$

.

5.2 Interconnect Area

In order to compute the interconnect area, we need to determine the total number of nets that are connected to the memory modules. We then estimate the area of each net which then determines the total interconnect area for the memory modules.

The number of wires that are required for the memory modules is equal to the number of pins. (i.e., the sum of address, data and control pins in all memory modules). Thus we have

$$TotalNumPins = \sum_{i=1}^{m} P_i * (\lceil log_2(N_i) \rceil + B_i + 1)$$

The AverageWireLength is assumed to be equal to half the perimeter of the modules.

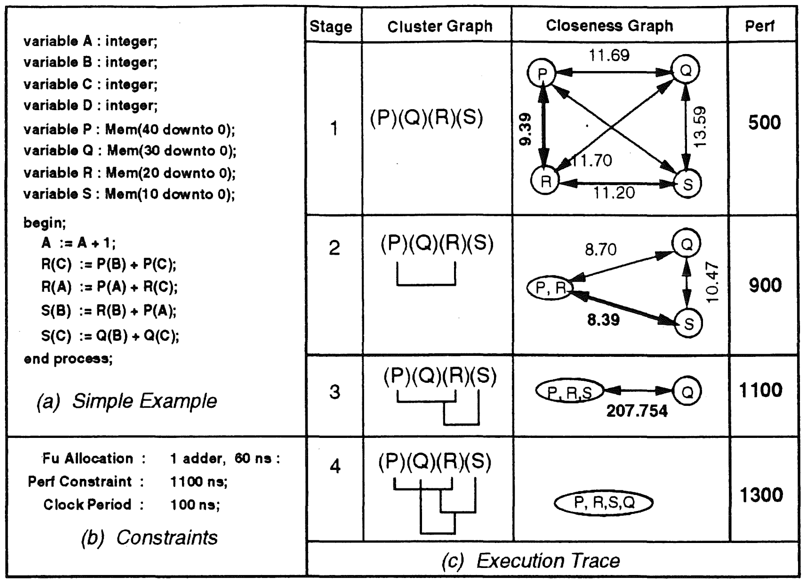
$$AverageWireLength = \sum_{i=1}^{m} W_i + Max_{1<i<m}(H_i)$$

Based on these assumptions, the interconnect Area can be derived as follows:

$$TotalInterconnectArea = AverageWireLength * TotalNumPins * \beta$$

6 Results

MeSA has been implemented in 'C' on a SUN SparcStation. In this section we present a walkthrough example and the results of running MeSA on a number of examples. For all our experiments, MeSA was provided with the high level VHDL description of the design and the functional unit allocation.



```
variable A : integer;
variable B : integer;
variable C : integer;
variable D : integer;
variable P : Mem(40 downto 0);
variable Q : Mem(30 downto 0);
variable R : Mem(20 downto 0);
variable S : Mem(10 downto 0);
begin;
  A := A + 1;
  R(C) := P(B) + P(C);
  R(A) := P(A) + R(C);
  S(B) := R(B) + P(A);
  S(C) := Q(B) + Q(C);
end process;
```

(a) Simple Example

```
  Fu Allocation :   1 adder, 60 ns :
Perf Constraint :   1100 ns;
   Clock Period :   100 ns;
```

(b) Constraints

Stage	Cluster Graph	Closeness Graph	Perf
1	(P)(Q)(R)(S)	P Q R S 11.69 9.39 13.59 11.70 11.20	500
2	(P)(Q)(R)(S)	P, R Q S 8.70 10.47 8.39	900
3	(P)(Q)(R)(S)	P, R, S Q 207.754	1100
4	(P)(Q)(R)(S)	P, R, S, Q	1300

(c) Execution Trace

Figure 9: Simple Example

6.1 A Simple Walkthrough Example

In Figure 9(a), we show a simple description containing 4 array variables. The constraints provided to MeSA are shown in Figure 9(b).

Initially each of the four variables are assumed to be in separate groups. A closeness graph is created (Figure 9(c)) showing the implementation cost if the variables are merged. In the first stage, the variables P and R are merged based on the lowest cost. In stage 2, the variable S got merged along with P and R. In the last stage, a single cluster is obtained. As the variables are merged together the performance deteriorates because of address translation. Finally the design that results in the lowest cost but meeting the performance constraint is selected from the cluster tree.

6.2 Kalman Filter

The Kalman Filter [20], contains six array variables of various sizes. With a simple memory allocation algorithm, six memory modules were allocated and the total layout area for the memory modules was 39 million sq microns. On the other hand, MeSA was able to derive a 35% more efficient allocation of 2 memory modules reducing the memory area to about 25 million sq microns. By being able to exploit the adder available in the datapath for address translations, some of the overheads associated with variable grouping were avoided and just one additional state was required due to array variable merging.

6.3 Differential Heat Release Computation

Kalman Filter	
Number of Array Variables	6 – (A, K, G, Y, X, V)
Size of Array Variables	A (256 * 16) Y (16 * 16) X (16 * 16) K (256 * 16) G (64 * 16) V (4 * 16)
Perf Constraint	3500 clocks
Clock Period	100 ns
Fu Constraint	1 ALU, 1 Multiplier

Alloc Results	**Without Clustering**	**MeSA**
#Mem Modules	6 – (M1, M2, M3, M4, M5, M6)	2 – (M1, M2)
#Ports in each Mem Module	M1 <= 1, M2 <= 1, M3 <= 1, M4 <= 1, M5 <= 1, M6 <= 4	M1 <= 1 M2 <= 2

Size of each Mem Module	M1 <= 256*16, M2 <= 16*16, M3 <= 16*16
	M4 <= 256*16, M5 <= 64*16, M6 <= 4*16
		M1(608 * 16)
		M2(4 * 16)

Vars assigned to each Mem module	M1 <= A, M2 <= Y, M3 <= X M4 <= K, M5 <= G, M6 <= V	M1 <= Y,K,A,G,X M2 <= V
Num States	40 states	41 states
Layout Area(Mem)	39.185 M sq microns	25.199 M sq microns
Performance	3168 clocks	3424 clocks

Figure 10: Kalman Filter Results

Differential Heat Release Computation Algorithm	
Number of Array Variables	4 – (P, V, B, D)
Size of Array Variables	P(469 * 16) V(469 * 16) B(469 * 16) D(469 * 16)
Perf Constraint	1800 clock cycles
Clock Period	100 ns
Fu Constraint	1 ALU, 1 Multiplier, 1 shifter

Alloc Results	**Without Clustering**	**MeSA**
#Mem Modules	4 – (M1, M2, M3, M4)	2 – (M1, M2)
#Ports	M1 <= 4, M2 <= 1, M3 <= 1, M4 <= 1	M1 <= 3, M2 <= 1
Size	M1 <= 469*16 M2 <= 469*16 M3 <=469*16 M4 <= 469*16	M1 <= 469*16 M2 <= 1407*16
Variables	M1 <= P, M2 <= V, M3 <= B, M4 <=D	M1 <= P, M2 <= V,B,D
Num States	14 states	15 states
Layout Area(Mem)	136.2 M sq microns	93.88 M sq microns
Performance	1665 clocks	1792 clocks

Figure 11: Differential Heat Release Computation

The Differential Heat Release computation algorithm models the heat release within a combustion engine. The description for this experiment was taken from [21]. This description contains 4 array variables and was synthesized with the constraints shown in Figure 11. MeSA was able to share the variables into a one port memory module and a three port memory module. The area savings was again equal to about 40% with a small loss in performance (about 8%), compared to a simple allocation algorithm.

6.4 Discrete Fourier Transform

Decimation in Frequency - DFT Algorithm	
Number of Array Variables	4 – (SigReal, SigImag, WReal, WImag)
Size of Array Variables	All variables are of size(1024 * 16)
Perf Constraint	None given
Clock Period	100 ns
Fu Constraint	1 comp, 2 adder, 1 subtractor, 1 multiplier, 1 divider

Alloc Results	**Without Clustering**	**MeSA**
#Mem Modules	4 – (M1, M2, M3, M4)	3 – (M1, M2,M3)
#Ports	M1 <= 2, M2 <= 2, M3 <= 1, M4 <= 1	M1 <= 2, M2 <= 2, M3 <= 1
Size	M1 <= 1024*16 M2 <= 1024*16 M3 <=1024*16 M4 <= 1024*16	M1, M2 <= 1024*16 M3 <= 2048*16
Variables	M1 <= SigReal M2 <= SigImag M3 <=WReal M4 <= WImag	M1<=SigReal,M2<=SigImag M3 <=WReal,WImag
Num States	36 states	36 states
Layout Area(Mem)	106.698 M sq microns	99.8 M sq microns
Performance	36 states	36 states

Figure 12: Discrete Fourier Transform Computation

The Fast Fourier Transform (FFT) converts information from the time domain into the frequency domain. This representation in the frequency domain is used for various signal processing applications. The Discrete Fourier Transform is the discrete version of the continuous FFT transforms. Efficient algorithms for the computation of a N-point DFT are discussed in [22].

In this experiment we modeled the Decimation in Frequency algorithm in VHDL. The real and imaginary values for the input signal were modeled with 2 array variables ($SigReal$ and $SigImag$). The constants for W_N were modeled with 2 more array variables, ($WReal$ and $WImag$), resulting in 4 array variables.

MeSA was invoked on this design, with an FU allocation consisting of 2 adders, 1 subtractor, 1 multiplier and 1 divider. MeSA derived an allocation containing 3 memory modules. It allocated one memory module for the variable SigReal, another module for the variable SigImag and finally a single memory module for the remaining variables WReal and WImag. By sharing the two variables in the same memory module, the total reduction in the size of the design was about 8%. The details of the results are shown in Figure 12.

It is clear from all the above experiments, that mapping each array variable to a separate memory module does not lead to an efficient solution. Grouping all the array variables into one memory does not lead to an efficient solution either, since the large memory may require multiple ports, resulting in large designs.

7 Conclusions

This paper presented a new algorithm (MeSA) for efficient allocation of memory modules, required for the implementation of array variables in a given description. MeSA computes an efficient allocation of memory modules, determines the number of ports on each of the modules and derives an efficient grouping of the array variables into the allocated memories. It also takes into account the ordering of variables in a memory module and the address translation requirements. MeSA uses a layout model to estimate the layout costs of the memory modules.

From our experiments we can conclude that MeSA produces much more efficient designs, than systems that directly allocate one memory module per array variable or allocate one large memory module to store all the array variables.

8 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support. We would also like to thank Tadoshi Ishii for his suggestions and useful discussions during the course of this project.

9 References

[1] R. Camposano and W. Wolf, *High Level VLSI Synthesis.* Kluwer Academic Publishers, 1991.

[2] Daniel Gajski, Nikil Dutt, Allen Wu and Steve Lin, *High Level Synthesis.* Kluwer Academic Publishers, 1992.

[3] C.Tseng and D. Siewiorek, " Automated Synthesis of Datapaths in Digital Systems," *IEEE Transactions on CAD*, pp. 379–395, July 1986.

[4] F.J.Kurdahi and A.Parker, "REAL: A Program for Register Allocation," in *Proc. of the 24rd Design Automation Conf.*, IEEE/ACM, 1987.

[5] L.Stok, "Interconect Optimisation during Data Path Allocation," in *Proc. of the EDAC*, 1990.

[6] F.Depuydt, G.Goossens and H.De Man, "Clustering Techniques for Register Optimization during Scheduling Preprocessing," in *Proc. of the IEEE Conf. on Computer Aided Design.*, pp. 280–283, IEEE, November 1991.

[7] M. Balakrishnan et.al, " Allocation of Multiport Memories in Data Path Synthesis," *IEEE Transactions on CAD*, pp. 536–540, April 1988.

[8] I. Ahmed and C. Chen, "Post Processor for Data Path Synthesis Using Multiport Memories," in *Proc. of the IEEE Conf. on Computer Aided Design.*, pp. 276–280, IEEE, November 1991.

[9] T.Kim and C.L.Liu, "Utilization of Multiport Memories in Data Path Synthesis," in *Proc. of the 30th Design Automation Conference*, 1993.

[10] J. Vanhoof, I. Bolsens and H.De. Man, "Compiling Multi-dimensional Data Streams into Distributed DSP ASIC Memory," in *Proc. of the IEEE Conf. on Computer Aided Design.*, 1991.

[11] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf and W.F.J. Verhaegh, "Phideo: A silicon compiler for high speed algorithms," in *Proc. of the European Conference on Design Automation.*, IEEE, February 1991.

[12] J.Vanhoof, K.Van Rompaey, I. Bolsens, G.Goossens and Hugo De Man, *High Level Synthesis for Real Time Digital Signal Processing.* Kluwer Academic Publishers, 1993.

[13] A. Orailoglu and D. D. Gajski, "Flow Graph Representation," in *Proc. of the 23rd Design Automation Conf.*, pp. 503–509, IEEE/ACM, June 1986.

[14] A. Aho and U. J., *Principles of Compiler Design.* Massachusets: Addison-Wesley, 1977.

[15] M. McFarland and T. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis," *IEEE Transactions on Computer-Aided Design*, September 1990.

[16] B. Pangrle and D. Gajski, "State Synthesis and Connectivity Binding for Microarchitecture Compilation," in *Proc. of the IEEE Conf. on Computer Aided Design.*, pp. 210–213, IEEE, November 1986.

[17] N.Weste and K.Eshraghian, *Principles of COMOS VLSI Design: A Systems Perspective.* Addison Wesley Publishing Company, 1988.

[18] F.J.Kurdahi, *Area Estimation of VLSI Circuits.* PhD thesis, Univ. of So. Calif., August 1987.

[19] C. Ramachandran et.al, "Accurate layout area and delay modeling for system level design," in *International Conference on Computer Aided Design*, IEEE/ACM, November 1992.

[20] N. Dutt and C. Ramachandran, “Benchmarks for the 1992 high level synthesis workshop,” technical report, Dept. of Information and Computer Science, University of California, Irvine, CA 92717, 1992. Technical Report : 92-107.

[21] Francky Catthoor and Lars Svesnsson, *Application-Driven Architecture Synthesis.* Kluwer Academic Publishers, 1993.

[22] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing.* New Jersey: Prentice Hall International Inc, 1989.