ALGORITHMS FOR MEMORY SYNTHESIS

R. Nunna
rnunna@csufresno.edu

Department of Electrical and Computer Engineering
California State University Fresno
Fresno, CA 93740-8030

ABSTRACT

In this paper, we present a methodology for allocating memory structures and interconnect in data paths. The algorithms presented herein take a scheduled piece of system level specification along with the functional unit allocation and determine the minimum number of multi port memory elements, and the minimum amount of memory-datapath interconnect needed to synthesize the datapath. The details of the algorithms will be discussed along with an illustrative example.

INTRODUCTION

The design of application specific integrated circuits calls for an effective utilization of the available chip area. Synthesis algorithms must cater to both area and performance constraints. A top down synthesis approach involves system level scheduling, functional unit allocation, memory allocation, interconnect allocation and finally physical synthesis. Memory allocation and interconnect are crucial in that they impact performance in terms of memory accesses, and area in terms of multiple buses in the synthesized data paths. In applications that require data parallel access capabilities in order to complete computations within a given time budget, multi-port memories are essential.

Over the years, researchers have developed several techniques to minimize number of memory modules, and determine which variables should be grouped together to form memory structures. Most methods are based on heuristics or formulated using 0-1 integer programming.[1-8]

In this paper, we present a series of algorithms that allocate memory from a given scheduled data flow graph, and following the memory allocation, the algorithms determine optimal port assignments, and the interconnect assignments with a goal of minimizing number of memories, ports, read write conflicts and multiplexers.

THE PROPOSED APPROACH

In this section, an approach to the allocation and binding of multiport memories in a datapath will be presented. The multiport memory allocation problem can be stated as follows:

Given a register transfer sequence and a schedule, the multiport memory allocation problem is to assign variables in the register transfer sequence to a set of memories such that the total interconnect area (between memory and functional units) and the number of memories are minimized.

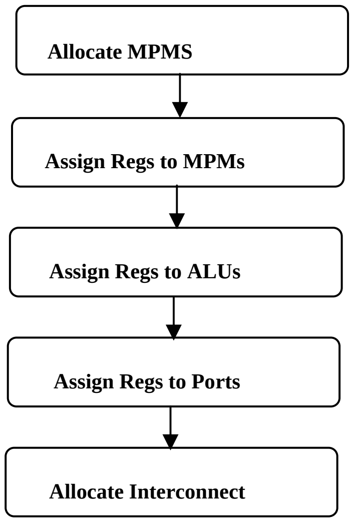
THE MEMORY MODEL

Consider a multiport memory M with p ports; of these ports, r ports are read only ports and w ports are write only ports and the rest rw = p-r-w are read write ports. Given such a scenario, and a schedule, a set of registers can be allocated to such a memory module only under the following conditions:

a. no more than r of these registers can be simultaneously read from memory
b. no more than w of these registers can be simultaneously written into
c. no more than rw of these registers can be simultaneously accesses for reading/writing.

OVERVIEW OF THE OVERALL ALLOCATION PROCESS

The figure shown below highlights the multiport memory allocation process. The first task is to allocate a set of memories that will be required for a conflict free access. This is dependent upon the schedule of register transfer operations and the number and kind of ports present in each memory module. The assumption is that each memory module is of the same kind (with fixed number of read and write ports). However, the allocation procedure can be modified easily to handle memories of different kinds.



The second task is to assign registers to the allocated memories. Many issues need to be considered in solving this sub problem. This problem has been mapped onto a graph coloring problem. The main focus of this subproblem is to map the registers in such a way as to provide for their required concurrent access. The third problem to be solved is the problem of assigning registers to the different functional units. Care has to be taken to assign certain registers to functional unit terms (for example in the case of subtracters and dividers). The next task is that of assigning registers already allocated to the memories to the different ports of the memory. The objective here is to assign registers to ports such that no more than r of these registers are accessed simultaneously (r is the number of read ports in the memory). This is also formulated as a graph coloring problem. Simultaneously with the port assignment, the interconnections from the memory elements to the functional units are also determined. In the following sections, each of these subtasks and solutions will be presented.

MEMORY ALLOCATION

The objective of this task is to allocate the minimum number of multiport memories such that there are enough ports present for the register transfer language sequences (RTL) to access data in each control step. Based upon our generalized model of the multiport memory, and a given set of RTL sequence, the minimum number of multiport memories that satisfy the simultaneous read and writes is given by:

$$\text{Min}_{\text{MPMs}} = \text{CS}(i = 1....n)\ \left\{\max \left\lceil \frac{\text{\# of simult reads}}{\text{\# of read ports}} \right\rceil, \left\lceil \frac{\text{\# of simult writes}}{\text{\# of write ports}} \right\rceil \right\}$$

From the RTL sequence, the maximum number of read and write accesses can be determined. The number of memories required is chosen such that in any control step, there are enough ports available for read and write without port conflicts.

Example: Consider the following scheduled RTL sequence and the availability of 2 port memories. One of the ports of this memory is read only and the other is read/write.

R1 = R2 + R3	R4 = R1 * R4
R5 = R4 + R1	R6 = R3 / R5
R2 = R2 + R6	R4 = R3 * R5

Given such a memory, at most two simultaneous reads can take place and one write can take place. In each of the control steps, four registers are accessed simultaneously. This means that we need at least four read ports. Using the formula shown above,

$$\text{Min}_{\text{MPMs}} = \text{CS}(i = 1....3)\ \left\{\max \left\lceil \frac{4}{2} \right\rceil, \left\lceil \frac{2}{1} \right\rceil \right\} = 2$$

we see that the minimum number of memories required is two. Using these two multiport memories with two read ports and one write port, all the access requirements by the schedule are met.

REGISTERS TO MEMORY ASSIGNMENT

Given a schedule and a set of registers that are active in the schedule, we see that two registers used concurrently in any control step cannot be mapped to a single register. They can however be mapped onto a single multiport memory if the number of available read ports are greater than or equal to two. When we assign registers to memories, our main goal is to partition the set of registers and assign them to memories such that the datapath has a conflict free access to them.

Graph coloring has been used successfully for register allocation in compilers.[9] Given a graph G=(V,E), graph coloring is the assignment of colors to each node of the graph such that if there is an edge between two nodes, then the two nodes are assigned different colors. Because graph coloring is an NP complete problem,[10] a heuristic method for coloring will be used.

Associated with the graph coloring problem is the notion of interference or conflict. The first step towards solving a graph coloring problem is to create interference graphs. In the context of register assignment, an interference graph is a graph G=(V,E), where V is a set of nodes (representing registers), E is a set of edges, and two nodes are connected by an edge if they are accessed simultaneously. The problem of assigning registers to memories now becomes the problem of coloring the nodes of the interference graph with a fixed number of colors such that no two nodes (registers) which have access conflicts share the same color. (However, these two nodes can share the same color only if there are sufficient memory ports available for them to be accessed from simultaneously).

The first step in the coloring process is the creation of interference graphs. Given a register transfer sequence, we first create two sets of nodes, called the read_nodes and write_nodes. For example, in the RTL sequence shown on the previous page,

{R1, R2, R3, R4, R5, R6} ae the read nodes and {R1, R2, R4, R5, R6} are the write nodes. The reason we partition this sequence in this manner is that registers are written into and read from at disjoint times and therefore their assignment can be considered sequential. However, while assigning colors, we make sure that the integrity of the interference is maintained.

Before the actual coloring process, we create two sets called node_stacks. These node_stacks play an important part in the coloring algorithm that we use. The main idea behind the success of the coloring algorithm is the following:

Let G=(V,E) be a graph for which we need to obtain a K-coloring. If a node N of G has a degree < K, then no matter how the graph is colored, there will always remain a color for N. N can be colored and removed from the graph. Along with the removal of N, the edges connecting N with the rest of the graph are also removed. The problem now reduces to coloring a graph with one node less and probably several edges less. Proceeding in this manner, it is possible that all the nodes in the graph will be removed.

This idea is at the heart of graph coloring solution to the register allocation problems used in compilers. However, in the case of compilers, when coloring is not possible, spill code is usually introduced. In the algorithms presented in this paper, we avoid the likelihood of spill codes because we have determined in advance, the minimum number of colors (memories) that are needed to allow for a conflict free coloring. We first create two sequences of nodes which are removed from the conflict graph one at a time according to the main idea shown above. As these nodes are removed from the graph, they are pushed onto a stack for further processing. The two sequences are created one for the read nodes and one for the write nodes. Once we have the nodes pushed onto a stack, the next procedure is to color the nodes without disturbing the conflicts that might already exist between them.

The coloring algorithm proceeds as follows. In the first step, the read stack is popped and the first element is assigned a seed color (a memory number). Next for every node popped from the stack, we check if there is an access conflict generated if this node is given the same color as the previously colored node. We check for two conditions that might arise: first if the current node is given the same color as the previous node, are there enough ports to allow for simultaneous access if necessary, and secondly we also check to see if by assigning a color to the node which is the color of an already colored node, are we creating a write hazard., ie. are we trying to write into more locations in the memory than are write ports available. After checking for these conditions, we assign a color to the current node - either a new color or an existing color. After we color a node, we mark the node as colored. This is necessary and will be used in the coloring of the write nodes. Coloring the write nodes is similar to the coloring of read nodes. The only difference is that we first check to see if the nodes have already been colored by the previous coloring process (during the coloring of read nodes). Figures a-e, show the outline of the algorithms used in the coloring process.

```
{
 LB – lowerbound(schedule) /*determine
        numbers of mems*/
Stack_setup(read_nodes) /* RHS nodes*/
/*create node_stack1 */
stack_setup(write_nodes) /* LHS nodes */
/* create node_stack2 */
no_of_colors = LB /*total colors=num of mems
*/
curr_color=1
total_nodes = number(node_stack1)
curr_node = t_o_stack(node_stack1)
color(curr_node, curr_color)
/*first node can take any color */
update_list(done_nodes, curr_color, curr_node)
rem_nodes = total_nodes -1
while (rem_nodes <> 0) do
        curr_node=t_o_stack(node_stack1)
        assign_color_read(curr_node)
        pop(node_stack1)
        rem_nodes=rem_nodes – 1
endwhile
rem_nodes = number(node_stack2)
curr_color=1
while (rem_nodes <. 0) do
        curr_node = t_o_stack(node_stack2)
        assign_color_write(curr_node)
        pop(node_stack2)
        rem_nodes = rem_nodes -1
endwhile
}
```

Figure a. The basic allocation algorithm

/*This routine is used to order the nodes in the read_nodes set for coloring by the assign_color procedures. Nodes are ordered based on their degree, */

```
stack_setup(read_nodes)
{
build interference_graph(read_nodes);
while {interference graph <> null} do
        Nr = node with the smallest degree
        Remove Nr from interference graph
        Remove all edges connecting Nr with rest
        of graph
        Push Nr onto node_stack1 /*read_stack*/
Endwhile
```

Figure b. Stack set up for read nodes

/* this routine is used to order the nodes in the write_nodes set for coloring by the assign color procedures. Nodes are ordered based upon their degree. */

```
{
build interference_graph(write_nodes);
while {interference graph <> null} do
        Nr = node with the smallest degree
        Remove Nr from interference graph
        Remove all edges connecting Nr with rest
of graph
        Push Nr onto node_stack2 /*write_stack
*/
Endwhile
}
```

Figure c. Stack set up for write nodes

/* algorithm to color a node from the read set. A set of nodes that are overlapping in time are not assigned the same color if there are insufficient number of access ports in the memory unit represented by the color */

```
assign_color_read(curr_node)
{
flag = 0; /* node not colored */
while (flag == 0) do
if (((read_time(curr_node, nodes(done_nodes, curr_color)) <= no_of_rports)) &&
        ((write_time(curr_node, nodes(done_nodes, curr_color)) <= no_of_wports))) then
        color(curr_node, curr_color)
        update_list(curr_node, done_nodes, curr_color)
        mark(curr_node);
        flag=1;
else
        curr_color = curr_color + 1
        if (curr_color > LB) curr_color = 1;
endif
endwhile
}
```

Figure d. Coloring algorithm for read nodes

/*algorithm to color a node from the write set. The node is first checked to see if it is colored and if it not, then conflict times are checked and proper coloring is done */

```
assign_color_write(curr_node)
{
if (mark(curr_node) = 1) then
        return()
        /* node has already been colored by assign_color_read */
else
        flag = 0
while (flag == 0) do
        if ((read_time(curr_node, nodes(done_nodes, curr_color) <= no_of_rports) &&
        (write_time(curr_node, modes(done_nodes, curr_color) <= no_of_wports)) then
        color(curr_node, curr_color)
        update_list(curr_node, done_nodes, curr_color)
        mark(curr_node)
        flag = 1
else
        curr_color = curr_color + 1
endif
endwhile
endif
}
```

Figure e. Coloring algorithm for write nodes

REGISTERS TO FUNCTIONAL UNIT ASSIGNMENT

The next step in the memory allocation and assignment process is the registers to functional unit assignment. We assume the following model of the functional unit. Each functional unit has two input terminals (corresponding to the two inputs) and one output terminal (corresponding to its output). Attached to the input terminals are zero or more data steering units (multiplexors). In this step of the overall process, we assign the individual registers to the input terminals of the allocated functional units. The special conditions that are to be handled during this stage are the proper assignment of registers which belong to operations that are not commutative (such as division and subtraction). The following are the main tasks during the register to functional unit assignment.

i. building the functional unit busy table
ii. building conflict (interference) graphs for each functional unit register set
iii. assigning special colors (for divide and subtract operations)
iv. for each conflict graph, coloring the conflict graph using two colors (one for each of the input terminals of the functional units)

THE FUNCTIONAL UNIT BUSY TIMES TABLE AND CONFLICT GRAPH

The functional unit busy times table is constructed from the schedule with functional units already allocated and registers assigned (after lifetime analysis). Typically, the busy table consists of N rows and M columns. The rows correspond to the N control steps of the schedule and the columns correspond to the M functional units that are allocated and bound to the operations. Each element of the table is a set of registers such that each of these registers is an input to the corresponding functional unit in the control step. For example, consider the code sequence shown below. The five sets of register transfer sequences correspond to the five control steps. The functional unit number to which a operation is allocated to is also specified. From this we can build the busy table. From the busy table, the conflict graph for each functional unit can be built.

```
R3 = R1+R2(FU1)      R12=R1
R5=R3-R4(FU2)        R7=R3*R6(FU1)       R13=R3
R8=R3+R5(FU2)        R9=R1+R7(FU1  )    R11=R10/R5(FU3)
R14=R11.R8(FU2       R15=R12orR9(FU1)
R1=R14               R2=R15
```

	Func Unit 1	Func Unit 2	Func Unit 3
CS1	R1,R2		
CS2	R3,R6	R3,R4	
CS3	R1,R7	R3,R5	R10,R5
CS4	R12,R9	R11,R8	
CS5			

Each node of the conflict graph represents a register connected to a terminal of the functional unit. An edge between two nodes signifies that the two registers are input to the same functional unit and accessed simultaneously. Therefore, they cannot arrive at the functional unit on one data line.

REGISTER SPECIAL SET

For those operations in the RTL description that are not commutative (subtraction and division), we need to take special care while assigning registers to functional unit inputs so as to preserve the ordering of the operands to the functional unit. In this step, we identify such registers and classify them as a special set and pre-color them such that during the allocation process, they will always connect to the proper functional unit input.

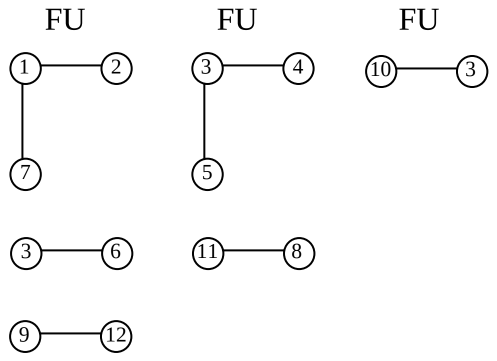


Figure f. : Conflict Graphs for the three ALUs

REGISTERS TO PORTS ASSIGNMENT

This is the final step in the memory allocation and binding process. So far, registers have been mapped onto memory modules, and also assigned to functional unit terminals. What is left to be done is the mapping of the individual registers in the memory modules to the ports of the memory and connecting these ports to the functional unit terminals. Once these tasks are completed, the datapath incorporating the multiport memories is completed.

The main objective of this phase of the algorithm is to assign the registers to memory ports such that the total interconnect cost (including multiplexer cost) is reduced. The procedure has to handle situations such as the following: if two registers are accessed simultaneously by the register transfer sequence and mapped onto the same functional unit, then these two registers cannot be accessed through the same memory port. We have to map them to separate memory ports. The following heuristic procedure handles this task effectively.

THE MODEL

Each functional unit (referred to as ALU henceforth) has two input terminals and one output terminal. There can be a multiplexer at each of the inputs and at the output of the ALU. Each memory unit has p ports out of which w are write ports and the rest are read only ports. There can be a multiplexer connected to the write port of each memory. This is needed when more than one functional unit has to write to the same memory unit. Since memories and functional units have already been allocated, we know the number of such units that are available. For notational convenience, we will use the following:

ALU[1..i] to represent all the ALUs.
Term[1..j] to represent the terminals of the ALUs (j=1,2). The two input terminals will be considered separately.
$\text{Term}_j.\text{ALU}_i$ represents the jth input terminal of the ith ALU.
Mem[1..n] to represent all the allocated memories.
Port[1..p] to represent the ports of the memories
$\text{Port}_p.\text{Mem}_n$ represents the pth read port of the nth memory module.

THE PORT ASSIGNMENT ALGORITHM

This algorithm assigns registers to ALU ports. As a first step, ALU terminal sets are created. For each ALU, there are two sets, one for each terminal. Each of these sets contain the names of the registers that have access to the corresponding terminal during all the control steps. These sets are built from the ALU busy tables. For example, for the busy table shown earlier, the two sets for functional unit 1 (ALU 1) are: {1,3,9} and {2,6,7,12}. This suggests that registers {1,3,9} be connected to terminal one of ALU 1, and registers {2,6,7,12} be connected to the other terminal. The objective is to assign these registers to ports of the memories to which they have already been assigned such that the number of multiplexers introduced at the inputs of the ALUs is minimized.

After we compile the two sets for each ALU, the next step is to classify the two sets according to the memory modules from which they are accessed from. After the registers are classified into the various memory modules for all the functional units, we try and assign registers to the individual ports of the memory modules. The overall algorithm to assign registers to ports and connect the ports to the input terminals of the functional units is as follows:

```
For each ALU(i) (i=1,…number of ALUs)
  For each term j of ALU(i) (j=1,2)
    Classify each term(j) set into the memories
     For each memory(n) (n=1,…number of memories)
      For every reg (r ) in memory set (r = 1 to Num of reg)
         If conflict(reg (r ), port1.set.mem(n)) = null, then
           Port1.set.mem(n) =  port1.set.mem(n) U reg (r )
           If connect(port1.mem(n), term(j).ALU(i)) =false
          then
              Connect (port1.mem(n), term(j).ALU(i))
               ++num_conn.term(j).ALU(i)
           endif
         else
          port2.set.mem(n) = port2.set.mem(n) U reg( r)
           If connect(port1.mem(n), term(j).ALU(i)) = false
           then
              Connect (port1.mem(n), term(j).ALU(i))
               ++num_conn.term(j).ALU(i)

              endif
             endif
          endfor
       endfor
  endfor
```

Figure f. The port assignment algorithm

AN EXAMPLE

To illustrate the mechanics of the algorithms presented in this paper, the following code sequence will be used. It is assumed that a life time analysis of the code has been performed, and that the minimum number of registers are being used. The objective is therefore to allocate multiport memories and minimize the amount of interconnect that will be needed in order to connect the memories to the functional units.

R3 = R1+R2(FU1)	R12=R1	
R5=R3-R4(FU2)	R7=R3*R6(FU1)	R13=R3
R8=R3+R5(FU2)	R9=R1+R7(FU1)	
	R11=R10/R5(FU3)	
R14=R11.R8(FU2)	R15=R12orR9(FU1)	
R1=R14	R2=R15	

From this scheduled sequence, the conflict graph is generated, one for the read set and the other for the write set.

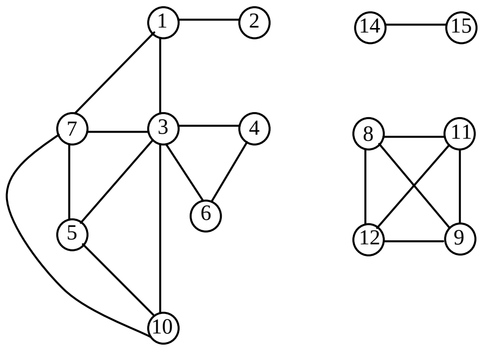


Figure g. Conflict graph for the Read Set

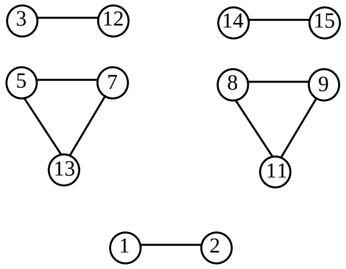


Figure h. Conflict graph for the Write Set

For the two conflict graphs generated, we can then create node_stacks, one for the read nodes and one for the write nodes. In the stack shown below, it is assumed that the top of the stack is to the right. The first set is for read nodes and the second is for the write nodes.

2	14	15	1	4	6	3	7	5	10	8	9	11	12

3	12	14	15	1	2	13	5	7	8	11	9

Figure i: Node stacks

The next step is to color the read nodes followed by the write nodes. From the register transfer sequence specified by the schedule, the minimum number of memory modules that are required is:

$$\text{Min}_{\text{MPMs}} = \text{CS}(i = 1....3)\ \left\{\max \left\lceil \frac{5}{2} \right\rceil, \left\lceil \frac{3}{1} \right\rceil \right\} = 3$$

We see that in both control step two and three, five registers are accessed simultaneously and three of them written simultaneously. Since our assumption in this example is that we have multiport memories with one read port and one read/write port, the minimum number of memories that are needed evaluates to 3. For notational convenience, we will refer to these memories as M1, M2 and M3.

Using the algorithms presented earlier in this paper, the registers are partitioned across the three memory modules as follows:

1. Memory M1 – Register Set {5, 11, 12, 14}
2. Memory M2 – Register Set {2, 3, 6, 7, 9}
3. Memory M3 – Register Set {1, 4, 8, 10, 13, 15}

The following table shows that such a partition is indeed valid. All the access requirements of the five control steps are met by this partition. At no time during the execution, more than three registers are written into and no more than six registers are read from.

	M1 {5,11,12,14}	M2 {2,3,6,7,9}	M3 {1,4,8,10,13,15}
CS1R		2	1
CS1W	12	3	
CS2 R		3,6	4
CS2W	5	7	13
CS3 R	5	3,7	1,10
CS3W	11	9	8
CS4R	11,12	9	8
CS4W		2	1
CS5 R		2	1
CS5W	14		15

The next step in the solution is the creation of the ALU busy tables and the ALU conflict graphs. Following the construction of the conflict graphs, we need to color the special registers. These are registers which must be connected to terminal one of the ALUs. We will assume that these will be colored red – corresponding to the first input of the ALU. Red (R) and Green (G) will be the two colors used. Using the same coloring procedure used earlier to partition the register sets into memory modules, we arrive at the following coloring for the three conflict graphs (Figure j).

The next step is to derive the terminal sets for each ALU. The terminal sets are:

1. ALU 1
   a. Term1.set = {1, 3, 9}
   b. Term2.set = {2, 6, 7, 12}
2. ALU 2
   a. Term1.set = {3, 8}
   b. Term2.set = {4, 5, 11}
3. ALU 3
   a. Term1.set = {10}
   b. Term2.set = {5}
   c.

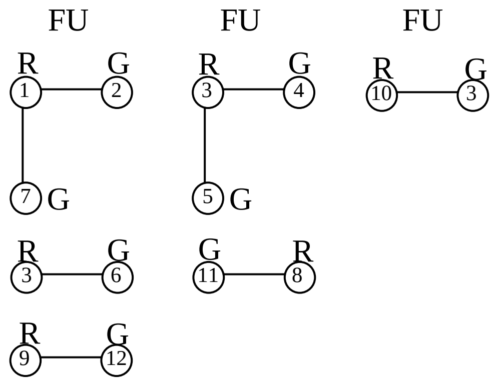


Figure j: Coloring of the Conflict Graphs

From these terminal sets, we can classify the registers according to the memory modules. From this classification, the registers are picked one by one and assigned to the ports of the memories in which they reside. The following are the memory sets for each of the terminals of the three ALUs.

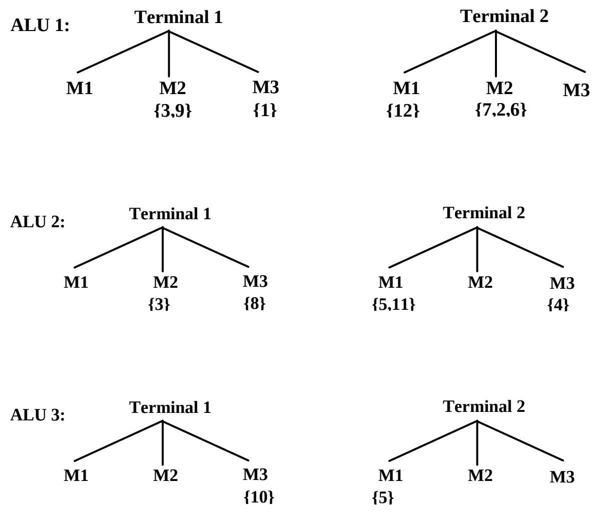


Figure k: Memory sets for ALU terminals

Following the register to port assignment and multiplexor creation procedure, we arrive at the following data path for the initial schedule. The synthesized data path has three allocated memories, each with two read ports and one write port. In addition to the coarse allocation of the memories, the algorithms define the interconnect pattern between the ALUs and the allocated multiport memories.

Furthermore, all necessary multiplexers are specified. From the synthesized datapath we can see that instead of having a uniformly large sized multiplexer at each input port, the synthesized datapath has multiplexers of just the right size for the number of inputs that it needs to process.

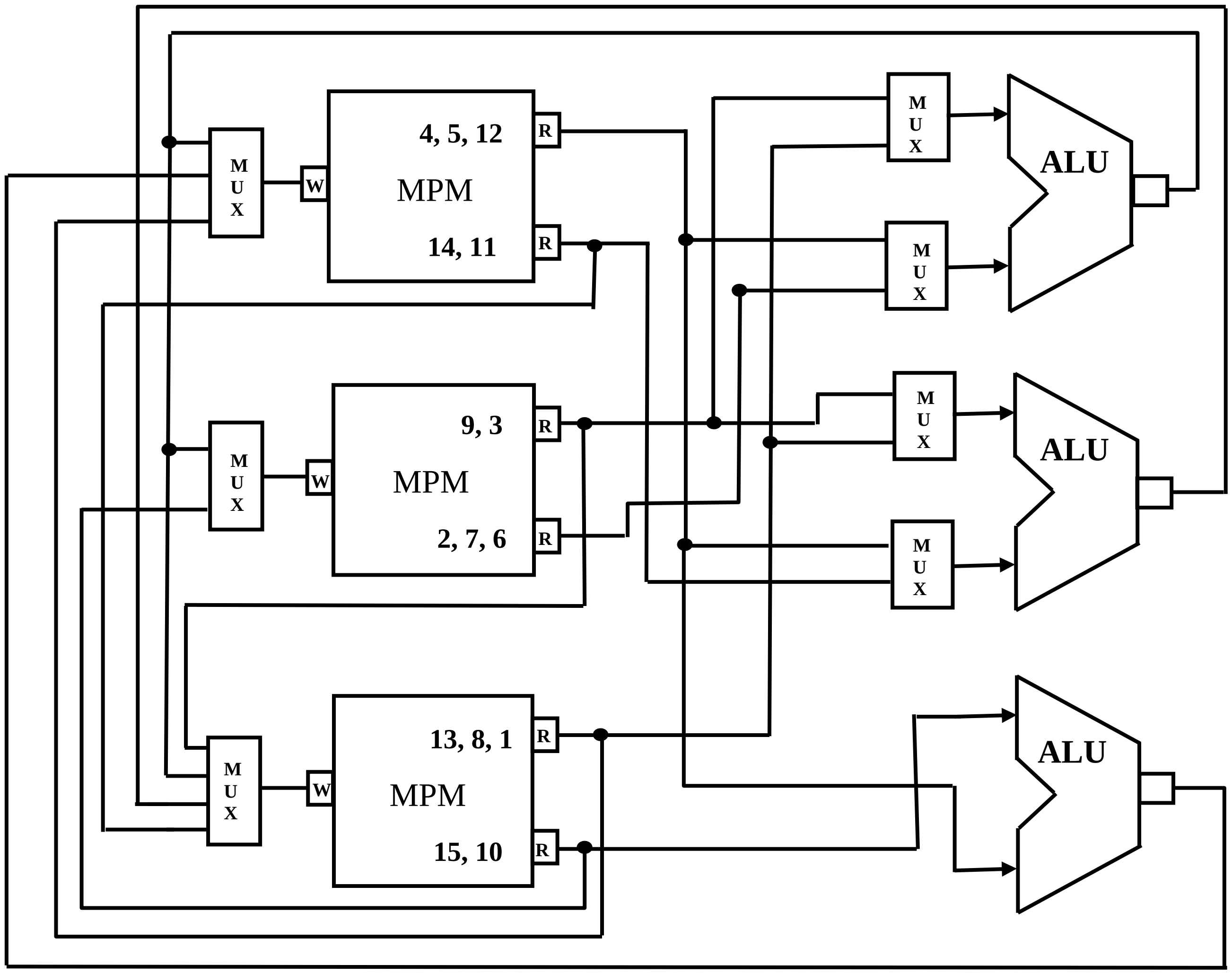


Figure l: Synthesized Data Path

CONCLUSION

In this paper, we presented algorithms for the allocation of memory modules, and interconnect during the data path synthesis phase in a synthesis based design methodology. The algorithms optimize for number of memory elements, and also for interconnect between the memory elements and functional units. The algorithms can be customized for memories with different number of input and output ports for writing and reading

REFERENCES

1. W. Shiue, "Memory Synthesis for Low Power ASIC Design", Proceedings of the 2002 IEEE Asia Pacific Conference on ASICs, Session 6B, August 2002

2. M. Balakrishnan, A. Majmudar, D. Banerji, J. Linders, and J. Majithia, "Allocation of Multiport Memories in Data Path Synthesis," IEEE Transactions on CAD, vol. 7, no. 4, pp. 536-540, April 1988.

3. T. Kim and C. Liu, “Utlization of Multiport Memories in Data Path Synthesis,” DAC, pp. 298- 302, June 1993.

4. 4. H.D. Lee and S.Y. Hwang, “A Scheduling Algorithm for Multiport Memory Minimization in Datapath Synthesis,” DAC, June1995.

5. 5. P. Lippens, J. van Meerbergen, W. Verhaegh, and A. van der Werf, “Allocation of Multiport Memories for Hierarchical Data Streams,”ICCAD, Nov. 1993.

6. 6. S. Wuytack, F. Catthoor, G. de Jong, and Hugo De Man, “Minimizing the Required Memory Bandwidth in VLSI System Realizations,” IEEE Transactions on VLSI Systems, Vol. 7, No. 4, Dec. 1999.

7. 7. P. R. Panda, “Memory Bank Customization and Assignment in Behavioral Synthesis,” ICCAD,1999.

8. Chien-in Henry Chen and Gerald Sobleman, “Single Port/Multiport Memory Synthesis in Data Path Design”, in Proceedings of the IEEE International Symposium on Circuits and Systems, 1990, pp 1110-1114.

9. G.J. Chaitin et al., “Register Allocation using Coloring”, Computer Languages, 45-57, 1981.

10. M. Garey and David Johnson, Computers and Intractability, W.H. Freeman and Co, 1979.

BIOGRAPHICAL INFORMATION

R. Nunna is a Professor in the Department of Electrical and Computer Engineering at California State University Fresno. He received his M.S. and Ph.D. degrees in Computer Engineering from the University of Louisiana at Lafayette. His research interests are in VLSI Circuits and Systems and CAD Methodologies.