Partitioning-Based Algorithm for Pipelined Scheduling and Module Assignment

Allen C-H. Wu
Joseph Lis
Daniel D. Gajski

Technical Report #91-32
April 9, 1991

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

Abstract

We propose partitioning-based algorithms for pipeline scheduling, module assignment, and interconnect sharing. A novel hypergraph model is used to perform module assignment which facilitates the identification of sharable resources and the calculation of interconnect costs. The algorithms use clustering and interchange improvement techniques to maximize interconnect sharing. The results show significant improvement over other published results.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

The data path synthesis of digital systems have been active research topics in recent years [1,3,5,6,8,9]. Two datapath models can be identified. One assumes that original description is a sequence of *assignment* statements, *if* statements, and *loop* statements. The problem is to map this description on connection of functional and storage units. Some or all of the functional units can be pipelined. The goal of this mapping is to minimize total execution time or number of functional units, storage units, and busses. The minimal execution time is achieved by exploiting maximal parallelism in the program such that minimal cost is obtained by maximal sharing of hardware units. The second model assumes that the given description is a single loop using a infinite stream of data. In this model, loop iterations form the pipeline and units are not shared inside a single iteration. We describe the solution for the second problem in this paper.

The key parameter in determining the performance of a pipeline is the **latency** which is the number of time steps separating two task initiations. This parameter influences the overall computation time as well as hardware cost associated with the pipeline design. Consider a task that is partitioned into 5 subtasks. The pipeline schedules of such a task with latency=1 and latency=3 are shown in Figure 1. Let **c** be the clock rate and **t** be the number of control steps of a task. The total computation time for **n** tasks is: **tc** + (**n**–1)**lc**. For example, if **l**=1, **n**=100, and **t**=5, the total computation time is 104**c**. On the other hand, if **l**=3 then the computation time of 302**c** is required to execute 100 tasks. Thus, the lower the latency, the faster the computation time.

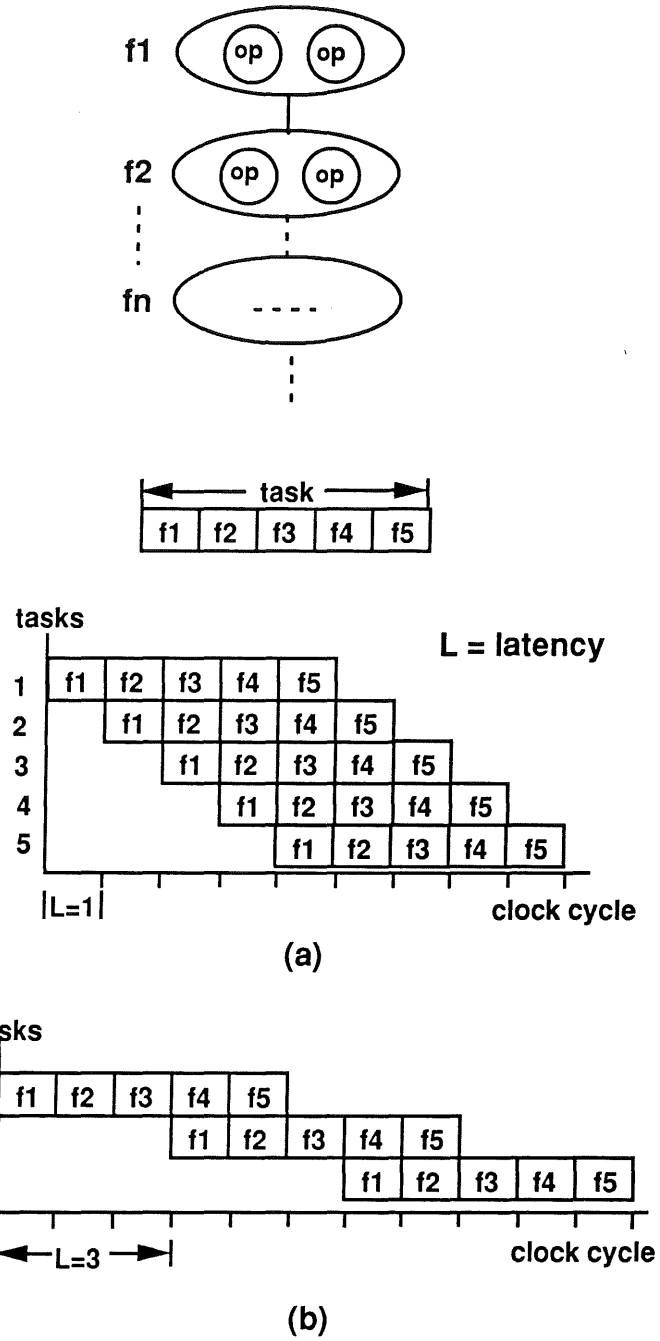


Figure 1. Pipeline example (a) Latency=1 and (b) Latency=3.

An adverse effect of a small latency is a correspondingly higher hardware cost due to the increased number of parallel operations in each state. Consider the example in Figure 1(a). Five subtasks are executed in the same clock cycle; therefore, minimally five function units are required to carry out the pipeline design with the latency=1. However, in Figure 1(b), only two function units are needed to carry out the pipeline design with the latency=3.

The pipelining technique has been incorporated into the scheduling, allocation, and resource binding tasks of high-level synthesis. Sehwa [8] first introduced a set of techniques for the synthesis of pipelined data paths. Park and Kurdahi [7] use a constrained clique partitioning approach with the goal of maximizing interconnect sharing to perform module assignment for pipelined data path designs. Hwang and Casavant [4] have developed a scheduling and hardware sharing algorithm using the force-directed scheduling approach [6] to synthesize both pipelined and non-pipelined designs.

In this paper, we present a partition-based formulation for pipeline scheduling, function unit allocation, and interconnect sharing. A novel hypergraph model is used to perform module assignment which facilitates the identification of sharable resources and the calculation of interconnect costs. Clustering and interchange improvement techniques are used to maximize interconnect sharing. Results have demonstrated significant improvement over other published results.

The remainder of this paper is organized as follows: Section 2 defines the notation and terms used throughout the paper. Section 3 describes the partition-based scheduling technique. Section 4 discusses the function unit assignment and

design is Max($\lceil n_{op}/r_{op} \rceil$). Furthermore, if a fixed latency l is used, a function unit can be used at most in l clocks whose indices modulo-l are (0,1,..,l-1). Based on these facts, we can state the following fact:

Fact 1: For a fixed latency l, there exists a set of disjoint partitions $S=\{s_k \mid k=0..l-1\}$ such that no two partitions are executed in the same clock cycle.

Consequently, we can formulate the pipeline scheduling as a partitioning problem in which operation nodes are assigned to disjoint partitions while maximizing the function unit sharing. The scheduling algorithm first determines the necessary and sufficient number of function units $R=\{r_{op}\}=\lceil n_{op}/l \rceil$ for a given latency l, or determines the minimal latency l can be performed for the given resources. For example, consider a data flow graph consisting of 15 addition operation nodes and 8 multiply operation nodes. To satisfy the performance requirement of latency=3, 5 adders ($\lceil 15/3 \rceil$) and 3 multipliers ($\lceil 8/3 \rceil$) are necessary and sufficient to carry out the pipeline design. As a result, there are three disjoint partitions, and initially, 5 adders and 3 multipliers are allocated to each partition.

3.2. Partitioning-based scheduling

The partitioning-based scheduling task consists of three steps: (i) Freedom calculation, (ii) Disjoint set partitioning, and (iii) Time step assignment. Algorithm I shows the pseudo code for this procedure.

Freedom calculation. Based on ASAP (as soon as possible) and ALAP (as late as possible) scheduling, the algorithm first calculates the freedom of each node. For a node v_i, $\mathbf{freedom}(v_i)=f_{ALAP}(v_i)-f_{ASAP}(v_i)$. Furthermore, a chaining strategy [5] is implemented for the freedom calculation. The freedom calculations for the FIR filter

are shown in Figure 2(a) where the delay for an adder is 40ns, a multiplier is 80ns, and the clock cycle is 100ns.

Disjoint set partitioning. In the second step, the algorithm assigns nodes into **l** disjoint partitions such that the resource sharing is maximized. The algorithm first assigns nodes with zero freedom to their corresponding partitions. Then, the algorithm assigns the rest of nodes to partitions in increasing order of freedom. For the node $\mathbf{v}_i$ with freedom interval from $\mathbf{f}_{ASAP}(\mathbf{v}_i)$ to $\mathbf{f}_{ALAP}(\mathbf{v}_i)$, the algorithm can assign $\mathbf{v}_i$ to partition $\mathbf{s}_k$ between $\mathbf{k}=(\mathbf{f}_{ASAP}(\mathbf{v}_i)$ **mod l**) and $\mathbf{k}=(\mathbf{f}_{ALAP}(\mathbf{v}_i)$ **mod l**). The node assignment of $\mathbf{v}_i$ is performed using the following steps:

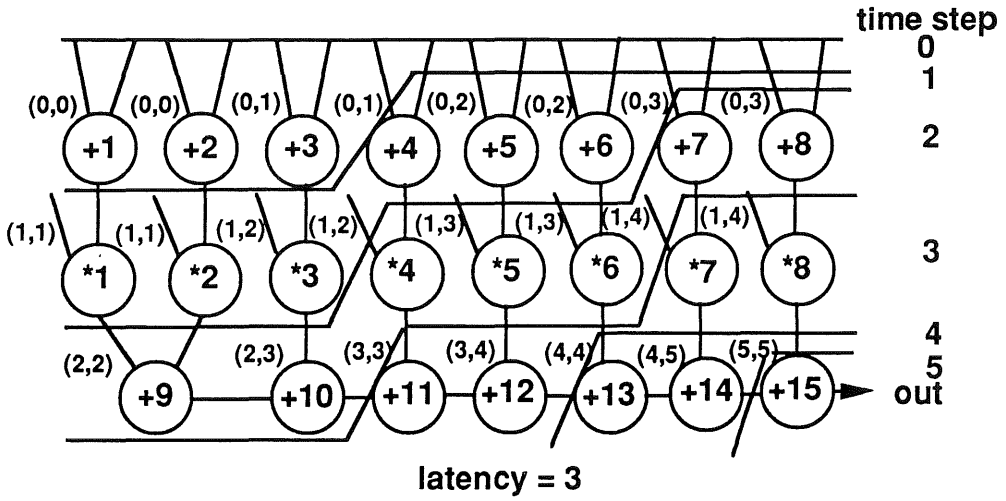
(1) From $\mathbf{k}=(\mathbf{f}_{ASAP}(\mathbf{v}_i)$ **mod l**) to $(\mathbf{f}_{ALAP}(\mathbf{v}_i)$ **mod l**), the algorithm locates the first partition $\mathbf{s}_k$ with the available function unit for node $\mathbf{v}_i$.

(2) If $\mathbf{s}_k$ contains a predecessor node $\mathbf{v}_j$ of node $\mathbf{v}_i$ and $\mathbf{v}_i$ cannot be chained with $\mathbf{v}_j$, then locate the next available partition set; otherwise, assign node $\mathbf{v}_i$ to $\mathbf{s}_k$.

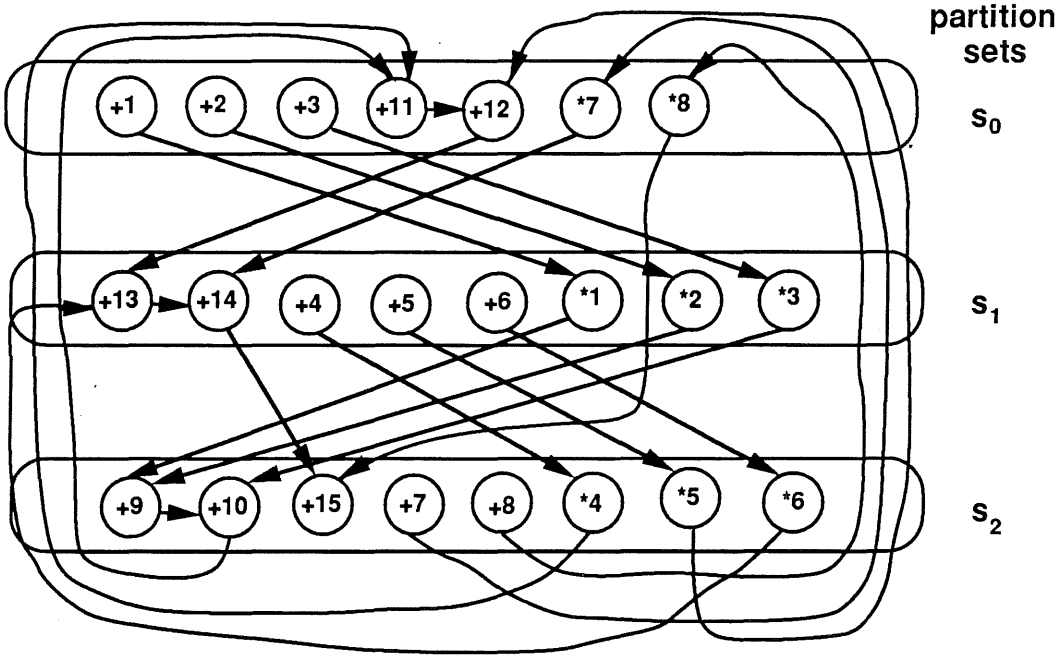
(3) If $\mathbf{s}_k$ contains a successor node $\mathbf{v}_j$ of node $\mathbf{v}_i$ and $\mathbf{v}_j$ can be chained with $\mathbf{v}_i$, then assign node $\mathbf{v}_i$ to $\mathbf{s}_k$; otherwise, assign node $\mathbf{v}_i$ to $\mathbf{s}_k$ and propagate node $\mathbf{v}_j$ to the next available partition set.

The final partition configuration of the FIR filter with fixed latency=3 is shown in Figure 2(b). Since in the first scheduling phase, the algorithm allocates only the necessary resources for each disjoint partition. The algorithm adds more resources when needs to satisfy the required performance.

Control step assignment. Finally, the algorithm assigns a control step to each operation node from inputs to outputs based on the data flow and its dependency. From disjoint set $\mathbf{s}_0$ to $\mathbf{s}_{l-1}$, the algorithm assigns the control step to the nodes without



(a)



(b)

Figure 2. (a). Freedom calculations and schedule and (b). Disjoint sets.

any data dependency edges and then deletes the nodes and their data dependency edges to the nodes' successors. For example, in Figure 2(b), in s_0 there are no data dependency edges for nodes +1, +2,and +3. The algorithm will assign step 1 to these three nodes and delete the nodes +1, +2, and +3. It will then delete the arcs between nodes +1, +2, +3 and nodes *1, *2, *3. The algorithm executes repeatedly until all of the nodes are assigned to a time step. The final scheduling of a FIR filter with fixed latency=3 is shown in Figure 2(a).

Algorithm I Partitioning-Based Scheduling

```
Partitioning_Based_Scheduling(G){
   determine_partition_set(G,R,l);
   freedom_calculation(G);
   sort_freedom_list(G);
   /**assign operation nodes to partitions**/
   for (i=1 to n){
      done = FALSE;
      step = f_ASAP(v_i);
      while (done == FALSE && step ≤ f_ALAP(v_i)){
         k = step mod l;
         if (op(i) ∈ R(s_k)){
            if ({v_j ∈ s_k | v_j is v_i's predecessor}){
               if (v_i can be chained with v_j){
                  s_k = s_k ∪ {v_i};
                  R(s_k) = R(s_k) - op(i);
                  done = TRUE;
               }
               else
                  step = step + 1;
            }
            else if ({v_j ∈ s_k | v_j is v_i's successor}){
               if (v_j can be chained with v_i){
                  s_k = s_k ∪ {v_i};
                  R(s_k) = R(s_k) - op(i);
                  done = TRUE;
               }
               else{
                  s_k = s_k ∪ {v_i};
                  R(s_k) = R(s_k) - op(i);
                  step = step + 1;
                  i = j;
               }
```

```
            }
            else{
                s_k = s_k ∪ {v_i};
                R(s_k) = R(s_k) - op(i);
                done = TRUE;
            }
        }
        if (step > f_ALAP(v_i) && done == FALSE){
            for (k=1 to l)
                R(s_k) = R(s_k) ∪ op(i);
            step = f_ASAP(v_i);
        }
    }
    /**control step assignment**/
    p = 0;
    k = 0;
    while (S≠φ ){
        k = k mod l;
        t_p = t_p ∪ {v_i ∈ s_k | v_i has no dependency edges};
        delete_node_edge(s_k,{v_i ∈ s_k | v_i has no dependency edges});
        k = k + 1;
        p = p + 1;
    }
}
```

4. Function unit assignment and interconnect sharing

The objective of function unit assignment and interconnect sharing is to assign operation nodes into function units such that the interconnect cost is minimized. We formulate the function unit assignment and interconnect sharing problem in terms of a hypergraph merging. In this section, we first describe the hypergraph formation and interconnect cost in the hypergraph model. Then, we describe the function unit assignment algorithm that minimizes the interconnect cost by clustering the operation nodes into function units exploiting data dependency similarity.

4.1. Formulation

4.1.1. The hypergraph

The algorithm first transforms the data flow graph **G** to a hypergraph **H** in which there are two types of hypernodes: (i) input/output and (ii) operation. Input/output hypernodes denote input/output ports. Each operation hypernode denotes a particular single-function unit such as adder, multiplier, or shifter. Each operation hypernode contains a set of nodes in the data flow graph. Hypernodes are connected with one or more hyperedges. Each hyperedge denotes the physical connections between two function units; the weight of a hyperedge is the number of dependency edges assigned to it.

At the time of hypergraph formation, each dependency edge is transformed into one hyperedge. After that, our algorithm performs hyperedge merging to reduce cost of interconnections. Before merging, each hyperedge is labeled as a right-data-input or a left-data-input. Two hyperedges can be merged if and only if:

(1) They have the same source and destination hypernodes.

(2) The operation nodes connected by the hyperedges have the same data dependency elapse time.

(3) They have the same label (right or left).

An example of case (2) is shown in Figure 3(a). The data dependency elapse time between two operation nodes $\mathbf{v}_1$ and $\mathbf{v}_3$ is defined as $\mathbf{t}(\mathbf{v}_3)$-$\mathbf{t}(\mathbf{v}_1)$. In case when $\mathbf{t}(\mathbf{v}_3)$-$\mathbf{t}(\mathbf{v}_1)$=2 and $\mathbf{t}(\mathbf{v}_4)$-$\mathbf{t}(\mathbf{v}_2)$=1, because $e_{a1,b3}$ needs one latch while $e_{a2,b4}$ needs two latches one extra multiplexer input is required for each hyperedge ($e_{a1,b3}$ and $e_{a2,b4}$) as shown in

Figure 3(a). On the other hand, two hyperedges $e_{a1,b3}$ and $e_{a2,b4}$ in Figure 3(b) can be merged since their elapse times are the same $\mathbf{t}(\mathbf{v}_3)$-$\mathbf{t}(\mathbf{v}_1)$=$\mathbf{t}(\mathbf{v}_4)$-$\mathbf{t}(\mathbf{v}_2)$=1. Furthermore, consider case in Figure 3(c), since $e_{a1,b3}$ enters the left input of v_b and $e_{a2,b4}$ enters the right input of v_b, and both inputs are not commutable. Therefore, $e_{a1,b3}$ and $e_{a2,b4}$ can not be merged since one multiplexer input is required for each hyperedge. However, if two inputs $e_{a1,b3}$ and $e_{a2,b4}$ are commutable, they can be commuted first and then merged into one hyperedge as shown in Figure 3(d). Another example is shown in Figure 3(e). If two hyperedges are connected to the different inputs of a function unit from the same source, they can not be merged.

Figure 4(b) shows an example of hypergraph formulation from the **DFG** shown in Figure 4(a). There are a set of input and output hypernodes V_{io}={v_a,v_b,...,v_p}. In addition, there are 4 hypernodes V_{op}={v_1,v_2,v_3,v_4}, where **type**(v_1) and **type**(v_2) represent adders and **type**(v_3) and **type**(v_4) represent multipliers. Each hypernode contains two operation nodes: v_1={**v**(+1),**v**(+3)}, v_2={**v**(+2),**v**(+4)}, v_3={**v**(*1),**v**(*3)}, v_4={**v**(*2),**v**(*4)}; therefore, **q**(v_1)=2, **q**(v_2)=2, **q**(v_3)=2, and **q**(v_4)=2. Since an adder and a multiplier have 2 inputs and 1 output each, **p**(v_1)=**p**(v_2)=**p**(v_3)= **p**(v_1)=2. There are two dependency edges between v_1 to v_3, thus **w**(e_{13})=2. The hyperedge direction is based on the flow of data between the hypernodes. For example, e_{13} is connected from the output of v_1 to the input of v_3; therefore, e_{13} is viewed as the outgoing hyperedge to the v_1 and the incoming hyperedge to the v_3.

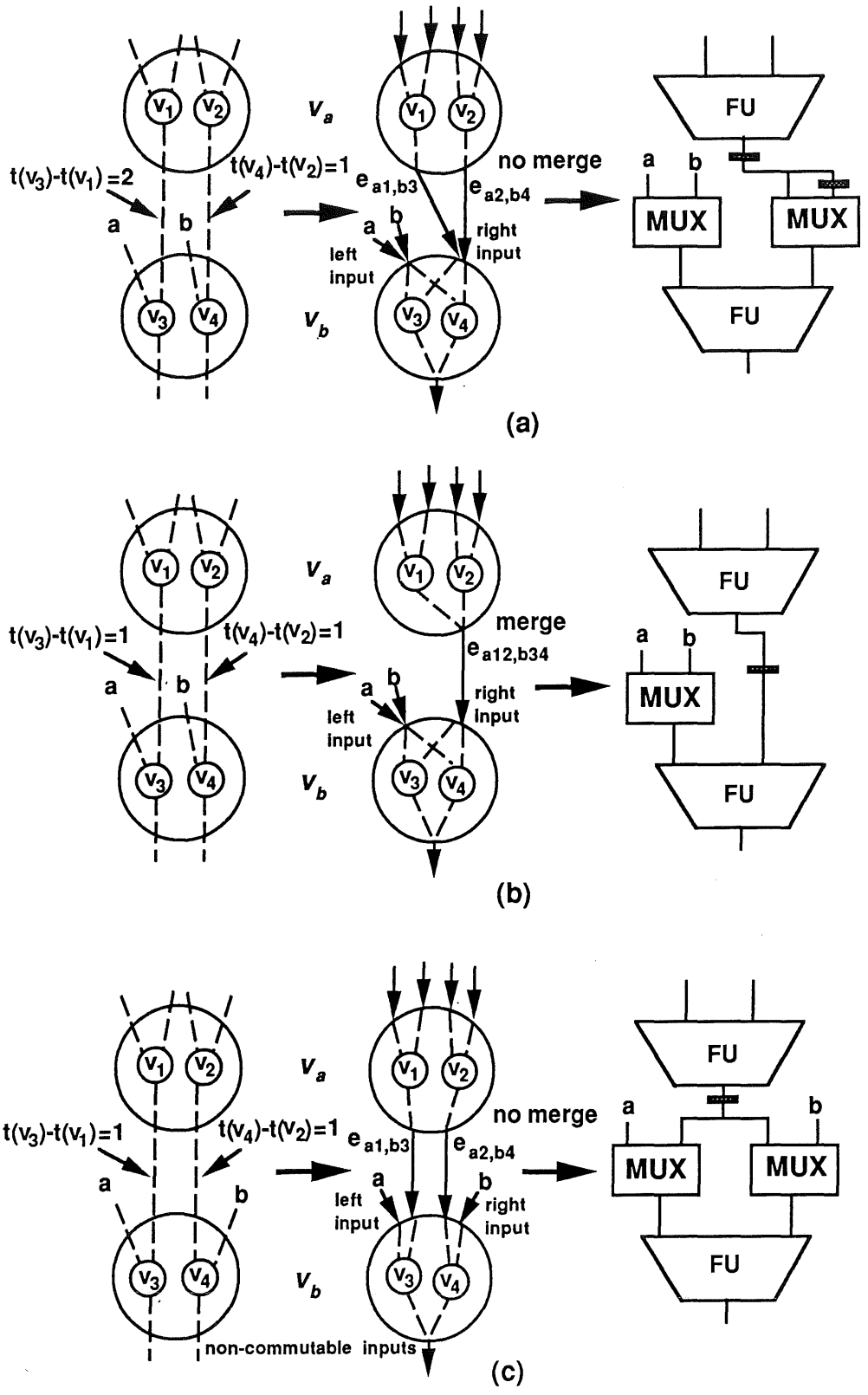
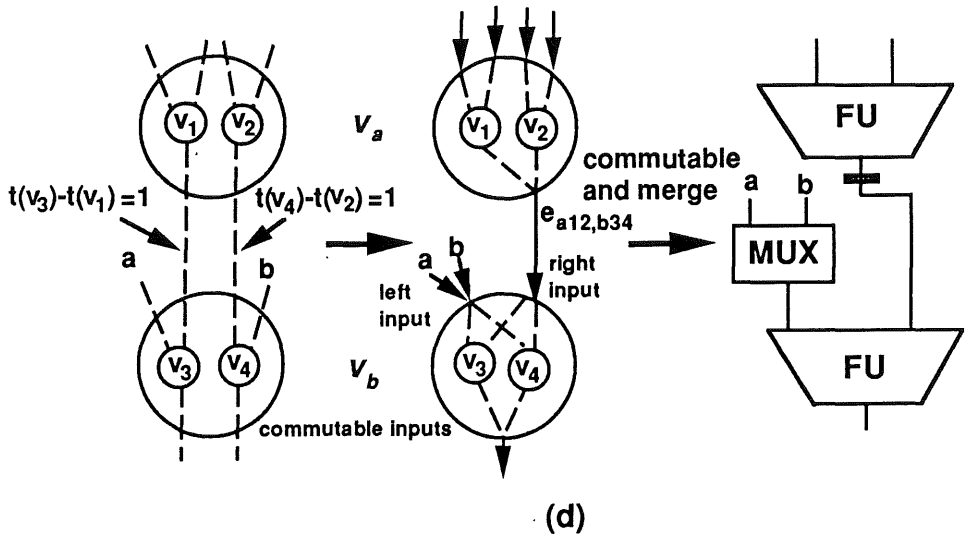


Figure 3. Hyperedge merging and interconnect sharing.



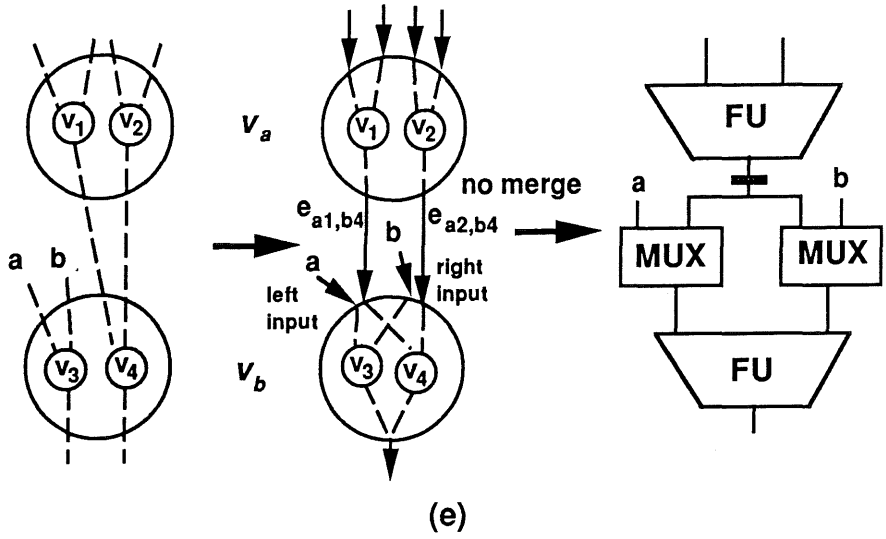
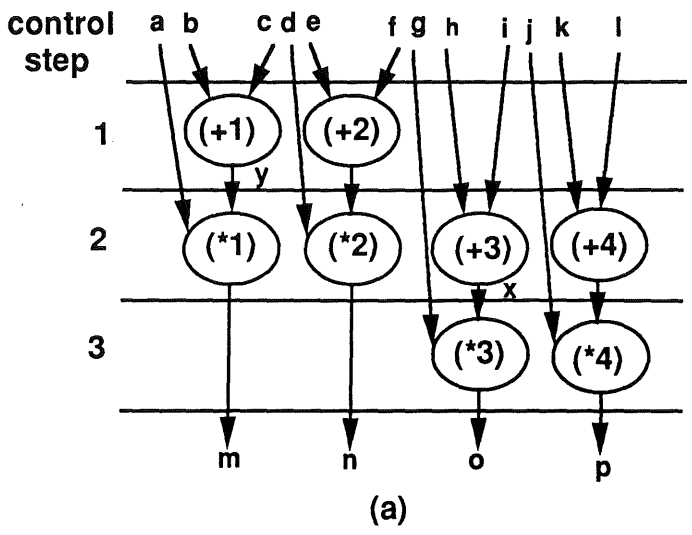
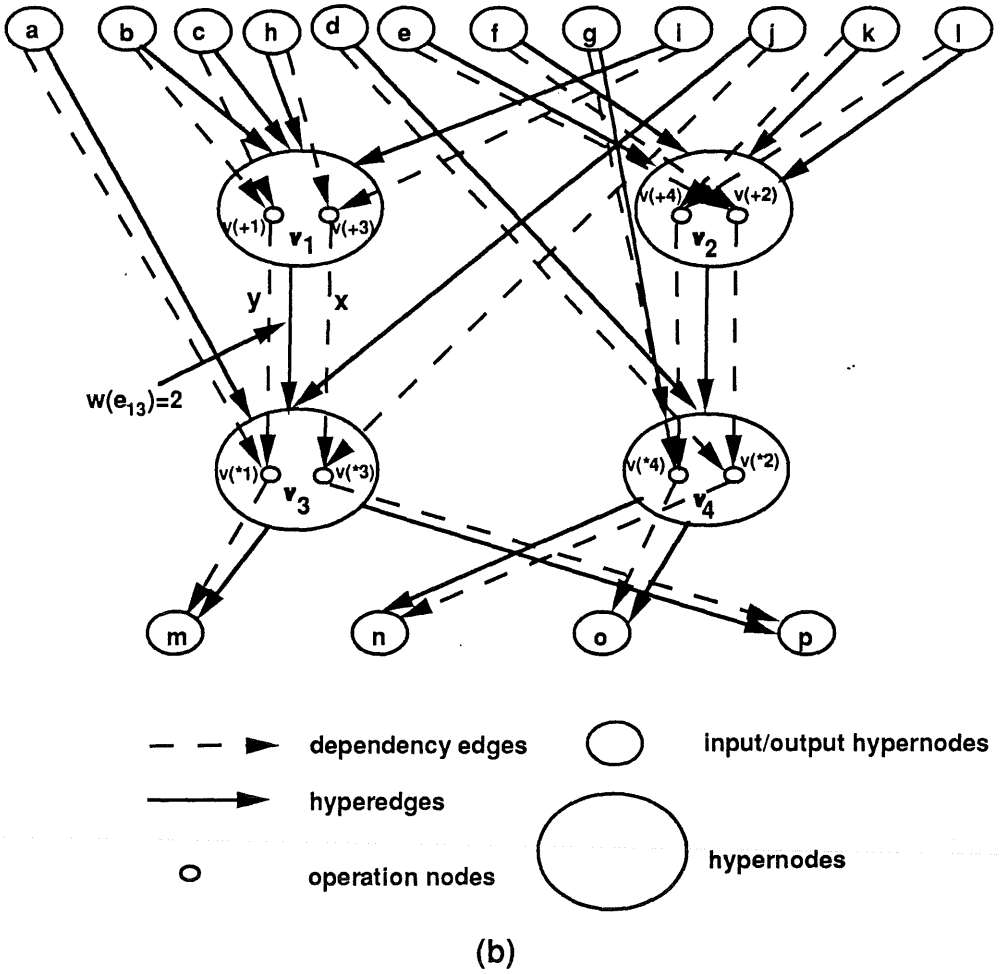


Figure 3. (cont.)



(a)



(b)

Figure 4. (a). A data flow graph example and (b). Hypergraph of (a).

assignment step, the algorithm assigns the operation nodes into hypernodes based on the closeness of data dependency elapse time among the operation nodes. In the interchanging improvement step, the algorithm takes into account the data dependency similarities between hypernodes, and maximizes the interconnect sharing by interchanging the operation nodes in the different hypernodes.

4.2.1. Initial assignment

In the scheduling step, the algorithm allocates the function units and partitions the operation nodes into disjoint sets $S=\{s_k \mid k=0..l-1\}$. To satisfy the latency requirement, the function units only can be shared by the operation nodes in the different sets. The task of the initial assignment is to cluster the operation nodes from different sets into hypernodes (function units) such that the closeness of data dependency elapse time in each cluster is maximized.

The data dependency elapse time of each operation node is calculated according to the final schedule. Each operation node includes two sets of data dependency elapse times: (i) the input elapse times between the node and it's predecessor nodes and (ii) the output elapse times between the node and it's successor nodes. For the example in Figure 2, consider operation node +9 which is scheduled at time step 2 ($t(+9)=2$). node +9 has two predecessor nodes, *1 and *2, which are scheduled at time step 1 ($t(*1)=t(*2)=1$), and one successor node +10 which is scheduled at time step 2 ($t(+10)=2$). Thus, $t_{in\_elapse}(+9)=\{1,1\}$ such that the input elapse times of node +9 are $t(+9)-t(*1)=1$ and $t(+9)-t(*2)=1$, and $t_{out\_elapse}(+9)=\{0\}$ such that the output elapse time of node +9 is $t(+9)-t(+10)=0$.

The algorithm first calculates the elapse times for each operation node. During the assignment process, the algorithm then calculates the closeness of operation nodes and available function units when selecting the best suited unit for each operation node. For an operation node $\mathbf{v}_i$, if $\mathbf{v}_i$ can be performed in a unit v_c, then the **Closeness**($\mathbf{v}_i$,v_c) is calculated as follows:

for ($\mathbf{v}_j \in v_c$){
  if ($\mathbf{t} \in \mathbf{t}_{in\_elapse}(\mathbf{v}_i)$ **and** $\mathbf{t} \in \mathbf{t}_{in\_elapse}(\mathbf{v}_j)$)
    Closeness($\mathbf{v}_i$,v_c) = **Closeness**($\mathbf{v}_i$,v_c) + 1;
  if ($\mathbf{t} \in \mathbf{t}_{out\_elapse}(\mathbf{v}_i)$ **and** $\mathbf{t} \in \mathbf{t}_{out\_elapse}(\mathbf{v}_j)$)
    Closeness($\mathbf{v}_i$,v_c) = **Closeness**($\mathbf{v}_i$,v_c) + 1;
}
Closeness($\mathbf{v}_i$,v_c) = **Closeness**($\mathbf{v}_i$,v_c) / $\mathbf{q}(v_c)$;

Since the function units can not be shared by the operation nodes in the same set, the algorithm will assign the operation nodes to the function units one set at a time. The algorithm calculates the closeness between each the operation node and available units, and assigns each operation node to the unit with the maximum closeness.

4.2.2. Improvement by interchanging

In the interchanging improvement step, the algorithm takes into account the data similarity among function units. The algorithm minimizes the multiplexer cost by merging the hyperedges. We first describe how to find a **feasible merging solution** that allows two hyperedges to be merged. Finding a **feasible merging solution** consists of two parts:

Finding a pair of feasible hyperedges. For any hypernode, there are two possible ways to merge the hyperedges: (i) merging of the incoming hyperedges, and (ii) merging of the outgoing hyperedges. An example of case (i) is shown in Figure 5(a). The hypernode v_c has two incoming hyperedges $e_{a1,c5}$ and $e_{b3,c6}$ from v_a and v_b respectively.

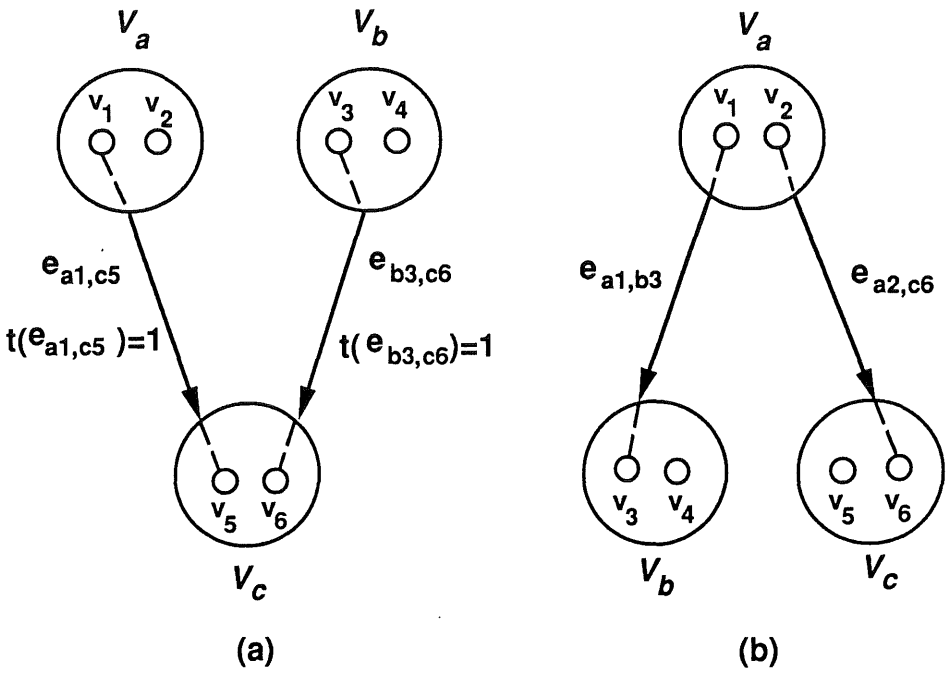


Figure 5. Feasible hyperedges.

An example of case (ii) is shown in Figure 5(b). The hypernode v_a has two outgoing hyperedges $e_{a1,b3}$ and $e_{a2,c6}$ entering v_b and v_c respectively. In both cases, if two hyperedges have (1). the same elapse time and (2). exited or entered the same type of hypernodes (**type**(v_a)=**type**(v_b) in case (i) and **type**(v_b)=**type**(v_c) in case (ii)), then they can possibly be merged by rearranging the operation nodes in v_a and v_b (Figure 5(a)) or in v_b and v_c (Figure 5(b)). Therefore, a pair of feasible hyperedges can be defined as: (i) hypernodes which have the same elapse time and (ii) they either exit from hypernodes of the same type and enter the same destination hypernode, or they are two

hyperedges exiting from the same hypernode and entering destination hypernodes of the same type.

Finding a feasible rearrangement of the operation nodes in a pair of hypernodes. After locating a pair of feasible hyperedges, a pair of feasible hypernodes can be located. For example, in Figure 5(a), $e_{a1,c5}$ and $e_{b3,c6}$ are the feasible hyperedges; v_a and v_b are the feasible hypernodes. There are two possible ways to merge $e_{a1,c5}$ and $e_{b3,c6}$: (i) relocating $\mathbf{v}_1$ from v_a to v_b or (ii) relocating $\mathbf{v}_3$ from v_b to v_a. The rearrangement of

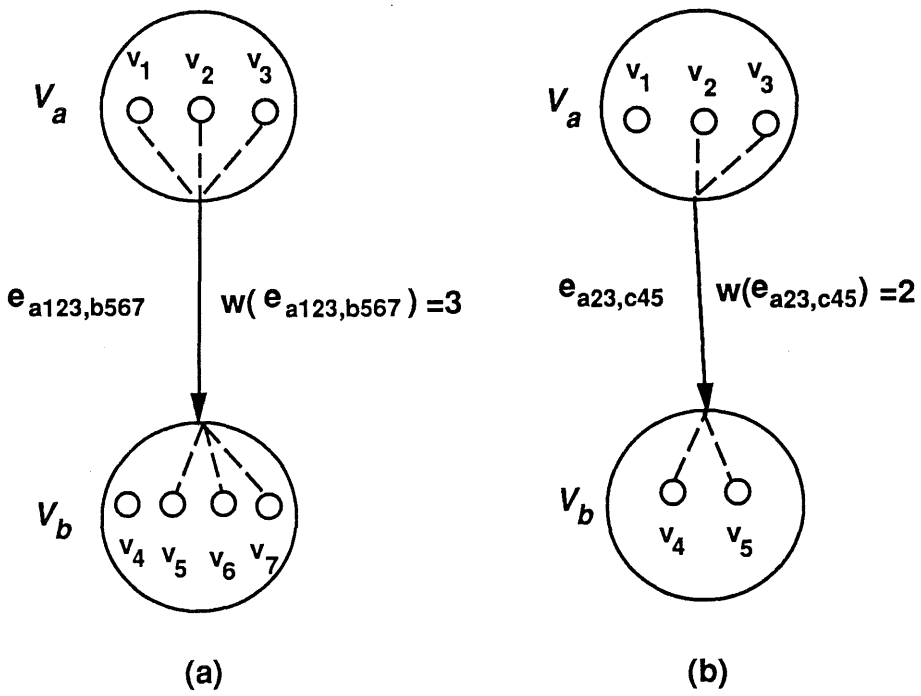


Figure 6. "Lock" hyperedges.

operation nodes must not violate the function unit sharing rule as described in the previous section. For example, in case (i), if $\mathbf{v}_3$ and $\mathbf{v}_1$ can not share the same function unit ($\mathbf{v}_3$ and $\mathbf{v}_2$ are in the same disjoint set), then the algorithm has to interchange $\mathbf{v}_3$ and $\mathbf{v}_2$ rather than moving $\mathbf{v}_3$ to v_a. However, if $\mathbf{v}_3$ and $\mathbf{v}_1$ can not be assigned to the same operation unit, then the feasible rearrangement of operation nodes does not exist since $e_{a1,c5}$ and $e_{b3,c6}$ can not be merged by interchanging $\mathbf{v}_3$ and $\mathbf{v}_1$.

Using a bucket structure [2], the algorithm first sorts hypernodes in terms of the number of feasible hyperedges by ordering a list in decreasing order. After finding a feasible merging solution, the algorithm calculates the total multiplexer cost. If a smaller multiplexer cost is obtained, then the algorithm merges the hyperedges; otherwise, the algorithm continues to find the next feasible merging solution. After merging, if the number of operation nodes in a hypernode is equal to the weight of it's incoming or outgoing hyperedge, then this hypernode achieves the maximum interconnect sharing. Hence, the algorithm will *lock* this hypernode, i.e. no more interchange for this hypernode is possible. For example, in Figure 6(a), $\mathbf{q}(v_a)$=$\mathbf{w}(e_{a123,b567})$=3. In Figure 6(b), $\mathbf{q}(v_b)$=$\mathbf{w}(e_{a23,b45})$=2. Both hypernodes will be *locked*. The algorithm runs repeatedly until no more hyperedges can be merged.

Algorithm II Function Unit Assignment

```
Let F be a set of feasible merging solution;
Function_Unit_Assignment(G,S,T,R){
   /*initial assignment*/
   calculate_data_dependency_elapse_time(G,T);
   V = build_hypernode(R);
   for (k=0 to l-1){
      for (v_i ∈ s_k)
         closeness_calculation(v_i,V);
      function_unit_assignment(s_k,V);
   }
   H = build_hypergraph(V,G);
```

```
    /*interchanging improvement*/
    mux_cost = mux_cost_calculation(H,φ );
    no_more_merge = FALSE;
    while (no_more_merge == FALSE){
       F = find_feasible_solution(H);
       if (F == φ )
          no_more_merge = TRUE;
       else{
          mux_cost_merge = mux_cost_calculation(H,F);
          if (mux_cost_merge < mux_cost){
             merge_hyperedge(H,F);
             mux_cost = mux_cost_merge;
          }
       }
    }
}
```

5. Experimental results

The algorithms are written in the C language, and the prototype implementation currently runs on SUN4 workstations under the UNIX operating system.

We have applied our algorithms to two examples: a FIR filter [8] (Figure 2(a)) and an elliptic filter [6] (Figure 10). For the FIR filter example, we have tested the example with the latency from 1 to 6. The examples of schedule and structural netlist with latency=2, 3, and 4 are shown in Figure 7, 8, and 9 respectively. Table 1 shows the comparison of our results with the results in [4] which is the only published paper documenting a complete set of results for the FIR filter and the elliptic filter examples. The results show that the number of multiplexer inputs was reduced up to 34% and the number of latches was reduced up to 19% using our algorithms.

For the elliptic filter example, we have tested the example with the latency from 1 to 9. The examples of schedule, operation assignment, and structural netlist with latency=8 and 9 are shown in Figure 11 and 12 respectively. The results in Table 2

show that the number of multiplexer inputs and the number of latches were reduced up to 30% compared to the results in [4]. However, in the case of latency=4, our algorithm used 8 adders but [4] used 7 adders.

6. Conclusions

We presented partitioning-based algorithms for pipeline scheduling, module assignment, and interconnect sharing. Based on a hypergraph model, the algorithms use clustering and interchange improvement techniques to maximize interconnect sharing. The results have shown significant improvement over other published results. This research has demonstrated that the hypergraph model facilitates the identification of sharable resources and calculation of interconnect costs. Furthermore, this approach produces very good results in very short time.

7. Acknowledgements

This work was supported by NSF grant #MIP-8922851, California MICRO grant #90-046, and contributions from Rockwell International, Western Digital, and Silicon Systems Inc. We are grateful for their support. The authors also like to thank Tedd Hadley, L. Ramachandran, Viraphol Chaiyakul, and Nels Vander Zanden for their useful discussions.

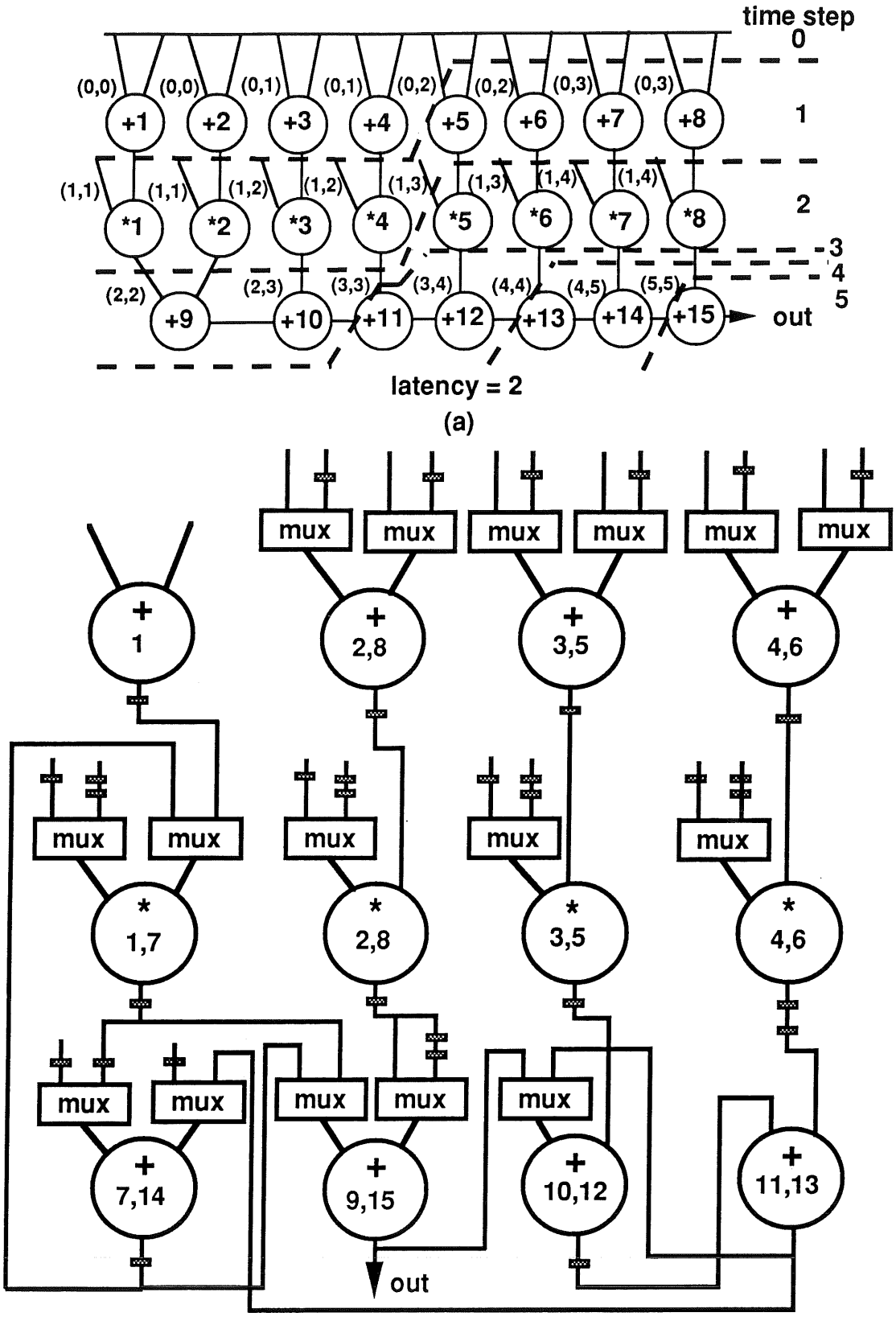
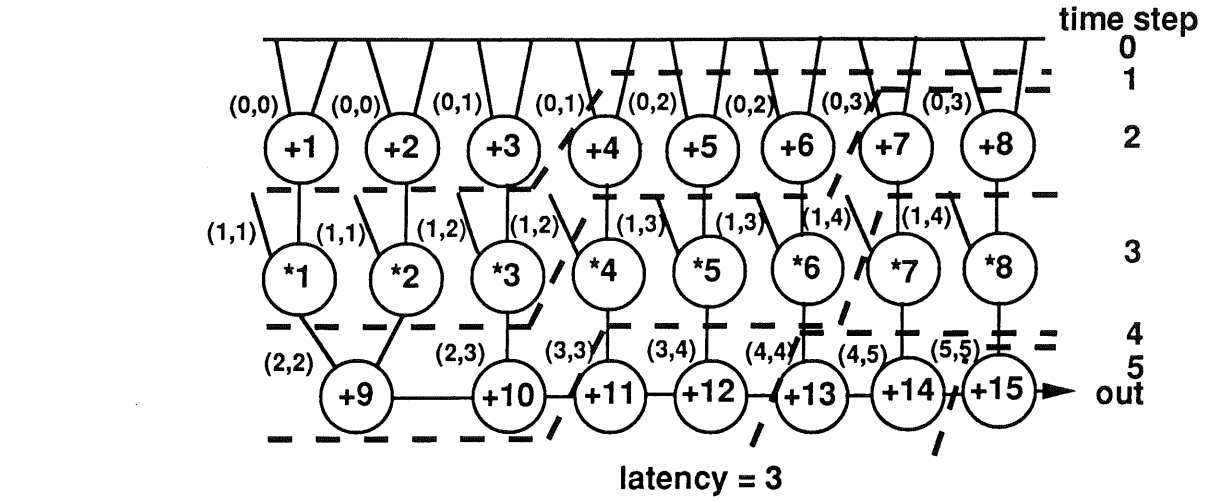
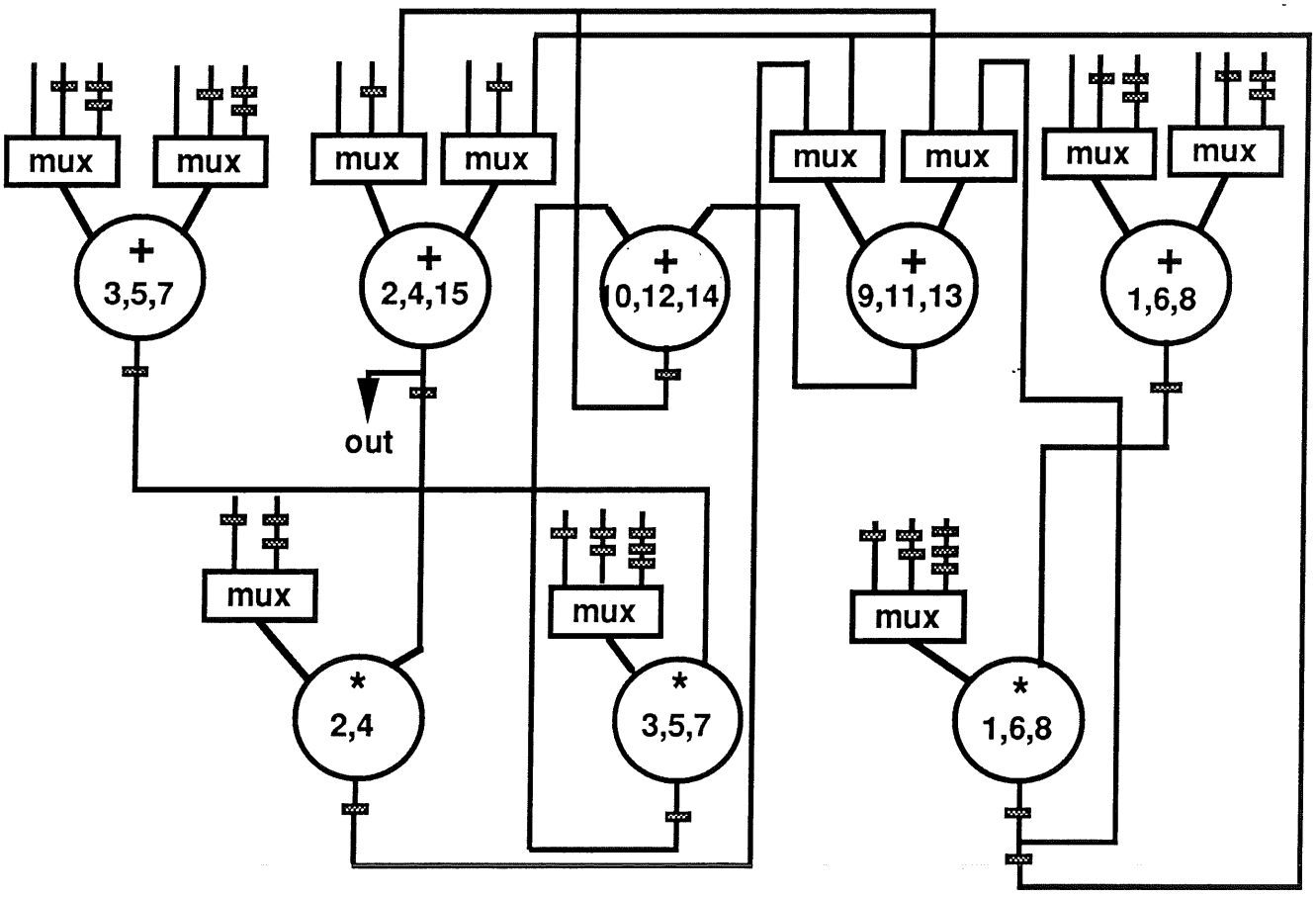


Figure 7. The schedule and structural netlist of a FIR filter with latency=2.

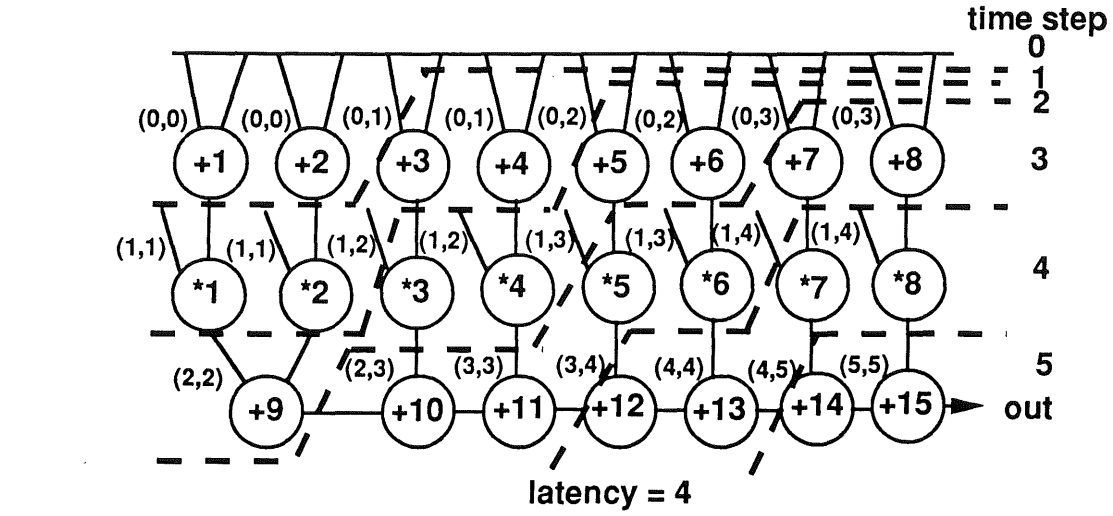


(a)

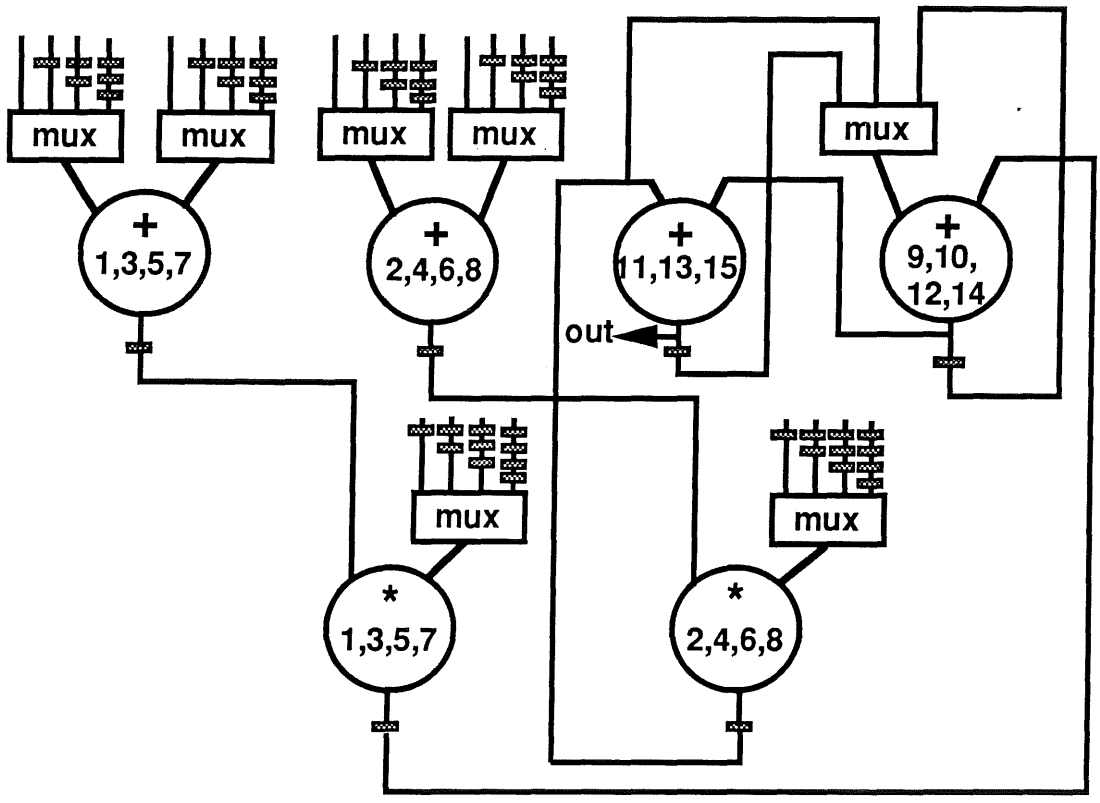


(b)

Figure 8. The schedule and structural netlist of a FIR filter with latency=3.



(a)



(b)

Figure 9. The schedule and structural netlist of a FIR filter with latency=4.

Latency / Resources	1			2			3			4			5			6		
	*	**	%	*	**	%	*	**	%	*	**	%	*	**	%	*	**	%
Number of *'s	8	8	0	4	4	0	3	3	0	2	2	0	2	2	0	2	2	0
Number of +'s	15	15	0	8	8	0	5	5	0	4	4	0	4	4	0	3	3	0
Size of multiplexers	0	0	0	32	40	-20.0	30	42	-28.6	27	41	-34.1	33	39	-15.4	34	37	-8.1
Number of registers	52	57	-8.9	34	42	-19.0	37	43	-14.0	50	50	0	43	46	-6.5	37	43	-14.0
CPU time (sec)	0.3	9	–	0.5	138	–	0.3	144	–	0.3	142	–	0.4	143	–	0.3	147	–

*: our results.

**: The results in [4].

Table 1. Results and Comparisons of a FIR filter example.

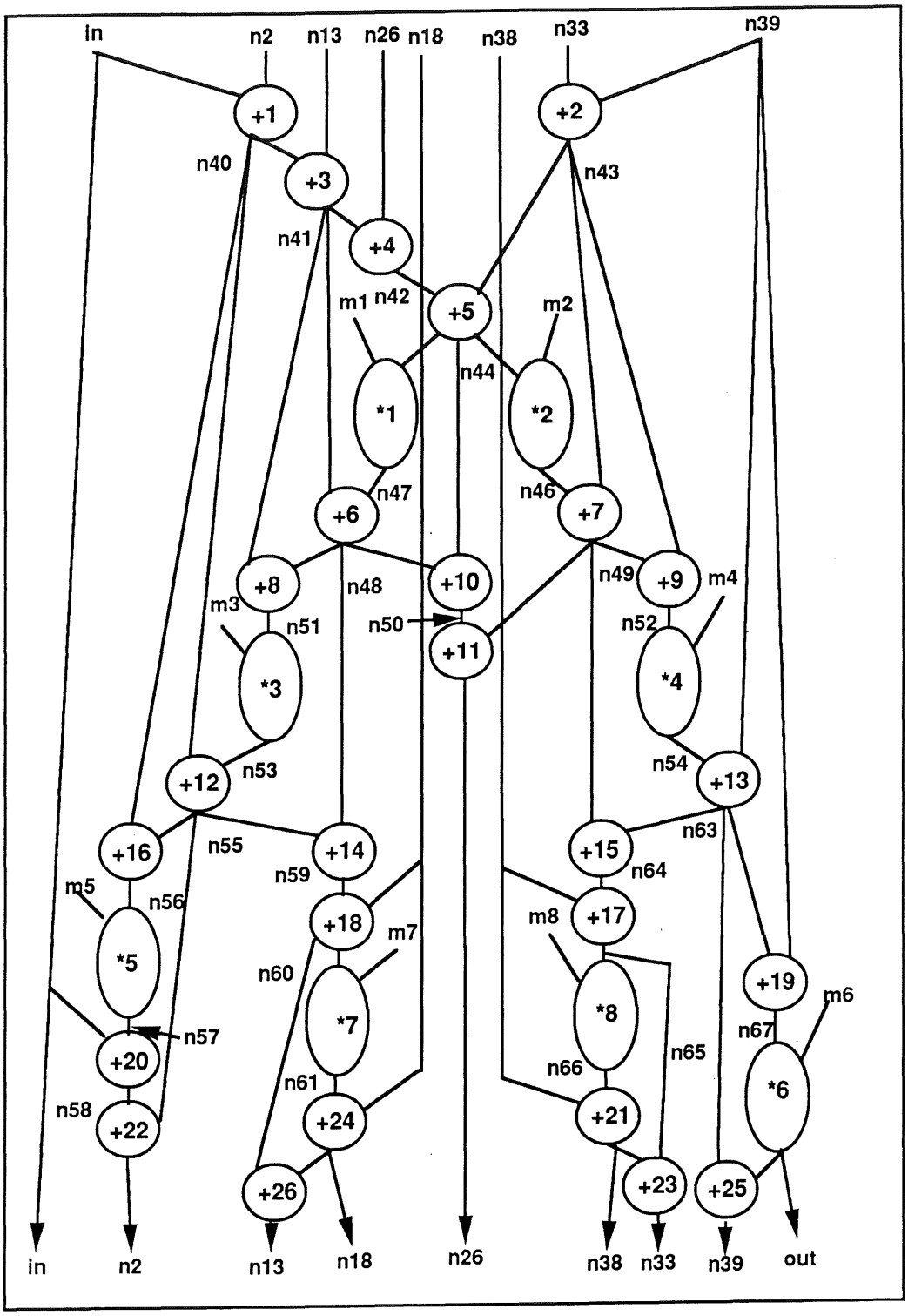


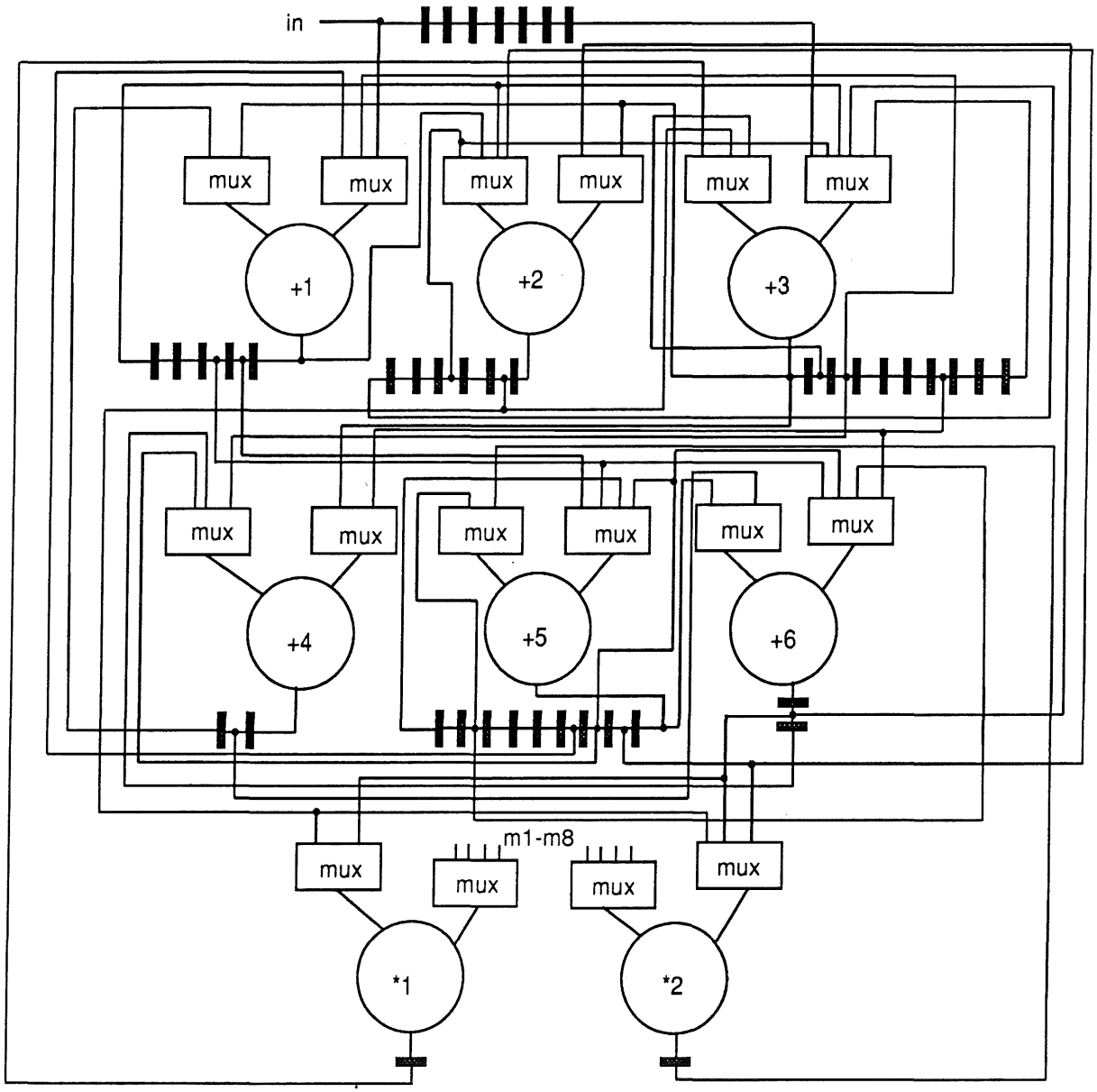
Figure 10. The data flow graph of an elliptic filter example.

step 1: +1,+3
step 2: +2,+4,+5
step 3: *1,*2
step 4: +6,+7,+8,+9
step 5: +10,+11,*3,*4
step 6: +12,+13,+14,+15,+16,+19
step 7: +17,+18,*5,*6
step 8: +20,+22,+25,*7,*8
step 9: +21,+23,+24,+26

(a)

FU	operation assignment
*1	*1,*3,*5,*8
*2	*2,*4,*6,*7
+1	+1,+2,+14
+2	+3,+5,+8,+11,+16
+3	+4,+6,+10,+12,+20,+21
+4	+15,+22,+23
+5	+7,+13,+18,+24,+25
+6	+9,+17,+19,+26

(b)



(c)

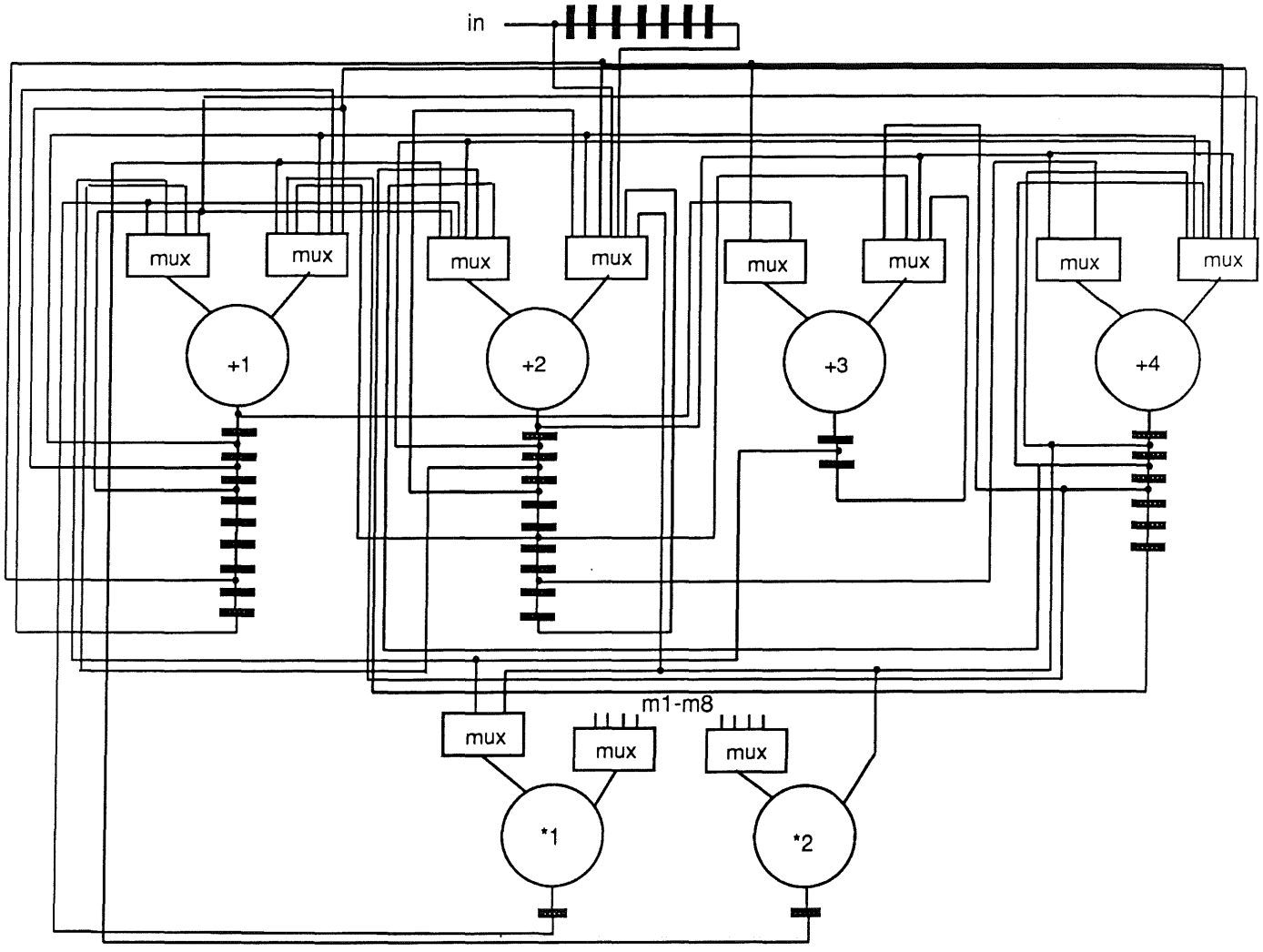
Figure 11. The (a) schedule, (b) operation assignment, and (c) structural netlist of an elliptic filter with latency=8.

step 1: +1,+2,+3
step 2: +4,+5
step 3: *1,*2
step 4: +6,+7,+8,+9
step 5: +10,+11,*3,*4
step 6: +12,+13,+16,+19
step 7: +14,+15,+17,+18,*5,*6
step 8: +20,+22,+25,*7,*8
step 9: +21,+23,+24,+26

(a)

FU	operation assignment
*1	*1,*3,*5,*8
*2	*2,*4,*6,*7
+1	+2,+6,+12,+14,+21,+25
+2	+1,+4,+7,+10,+13,+15,+20,+24
+3	+8,+16,+17,+23
+4	+3,+5,+9,+11,+18,+19,+22,+26

(b)



(c)

Figure 12. The (a) schedule, (b) operation assignment, and (c) structural netlist of an elliptic filter with latency=9.