steps in the schedule.

21. *Assume that you have a library that contains multiple implementations of functional units with different sizes and speeds. Devise an algorithm that would perform scheduling in combination with unit selection. Hint: An intuitive strategy for scheduling is to use the fast components only for the operations that are critical to the performance of the overall design while implementing the non-critical operations with the slower components.

22. Compute the cost of implementing the control logic for the schedules given in Figure 7.12(b), Figure 7.13(a) and Figure 7.13(b) assuming a datapath with 3, 4 and 6 functional units and a shared register file with 6, 8 and 12 read ports and 3, 4 and 6 write ports. Assume that the register file has 256 words.

Chapter 8

Allocation

8.1 Problem Definition

As described in the previous chapter, scheduling assigns operations to control steps and thus converts a behavioral description into a set of register transfers that can be described by a state table. A target architecture for such a description is the FSMD given in Chapter 2. We derive the control unit for such a FSMD from the control-step sequence and the conditions used to determine the next control step in the sequence. The datapath is derived from the register transfers assigned to each control step; this task is called datapath synthesis or datapath allocation.

A datapath in the FSMD model is a netlist composed of three types of register transfer (RT) components or units: functional, storage and interconnection. Functional units, such as adders, shifters, ALUs and multipliers, execute the operations specified in the behavioral description. Storage units, such as registers, register files, RAMs and ROMs, hold the values of variables generated and consumed during the execution of the behavior. Interconnection units, such as buses and multiplexers, transport data between the functional and storage units.

Datapath allocation consists of two essential tasks: unit selection and unit binding. Unit selection determines the number and types of RT components to be used in the design. Unit binding involves the mapping of the variables and operations in the scheduled CDFG into the functional, storage and interconnection units, while ensuring that the

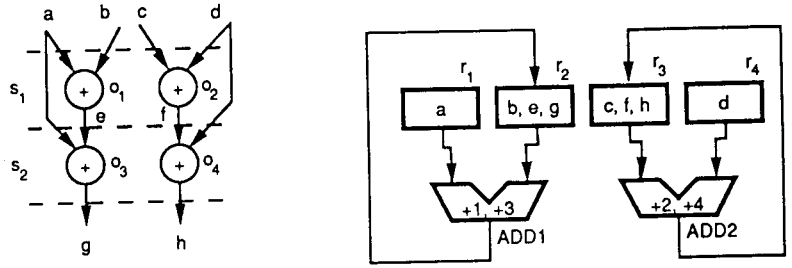

Figure 8.1: Mapping of behavioral objects into RT components.

design behavior operates correctly on the selected set of components. For every operation in the CDFG, we need a functional unit that is capable of executing the operation. For every variable that is used across several control steps in the scheduled CDFG, we need a storage unit to hold the data values during the variable's lifetime. Finally, for every data transfer in the CDFG, we need a set of interconnection units to effect the transfer. Besides the design constraints imposed on the original behavior and represented in the CDFG, additional constraints on the binding process are imposed by the type of hardware units selected. For example, a functional unit can execute only one operation in any given control step. Similarly, the number of multiple accesses to a storage unit during a control step is limited by the number of parallel ports on the unit.

We illustrate the mapping of variables and operations in the DFG of Figure 8.1 into RT components. Let us assume that we select two adders, *ADD1* and *ADD2*, and four registers, r_1, r_2, r_3 and r_4. Operations o_1 and o_2 cannot be mapped into the same adder because they must be performed in the same control step s_1. On the other hand, operation o_1 can share an adder with operation o_3 because they are carried out during different control steps. Thus, operations o_1 and o_3 are both mapped into *ADD1*. Variables a and c must be stored separately because their values are needed concurrently in control step s_2. Registers r_1 and r_2, where

variables a and e reside, must be connected to the input ports of *ADD1*; otherwise, operation o_3 will not be able to execute in *ADD1*. Similarly, operations o_2 and o_4 are mapped to *ADD2*. Note that there are several different ways of performing the binding. For example, we can map o_2 and o_3 to *ADD1* and o_1 and o_4 to *ADD2*.

Besides implementing the correct behavior, the allocated datapath must meet the overall design constraints in terms of the metrics defined in Chapter 3 (e.g., area, delay, power dissipation, etc.). To simplify the allocation problem, we use two quality measures for datapath allocation: the total size (i.e., the silicon area) of the design and the worst-case register-to-register delay (i.e., the clock cycle) of the design.

We can solve the allocation problem in three ways: greedy approaches, which progressively construct a design while traversing the CDFG; decomposition approaches, which decompose the allocation problem into its constituent parts and solve each of them separately; and iterative methods, which try to combine and interleave the solution of the allocation subproblems.

We begin this chapter with a brief discussion of typical datapath architectural features and their effects on the datapath-allocation problem. Using a simple design model, we then outline the three techniques for datapath allocation: the greedy constructive approach, the decomposition approach and the iterative refinement approach. Finally, we conclude this chapter with a brief discussion of future trends.

8.2 Datapath Architectures

In Chapter 2, we discussed basic target architectures and showed how pipelined datapaths are used for performance improvements with negligible increase in cost. In Chapter 3, we presented formulas for calculating the clock cycle for such architectures. In this section we will review some basic features of real datapaths and relate them to the formulation of the datapath-allocation problem.

A datapath architecture defines the characteristics of the datapath units and the interconnection topology. A simple target architecture may greatly reduce the complexity of the synthesis problems since the number of alternative designs is greatly reduced. On the other hand,

a less constrained architecture, although more difficult to synthesize, may result in higher quality designs. While an oversimplified datapath architecture leads to elegant synthesis algorithms, it also usually results in unacceptable designs.

The interconnection topology that supports data transfers between the storage and functional units is one of the factors that has a significant influence on the datapath performance. The complexity of the interconnection topology is defined by the maximum number of interconnection units between any two ports of functional or storage units. Each interconnection unit can be implemented with a multiplexer or a bus. For example, Figure 8.2 shows two datapaths, using multiplexer and bus interconnection units respectively, which implement the following five register transfers:

$$s_1\colon r_3 \Leftarrow ALU1\,(r_1, r_2);\quad r_1 \Leftarrow ALU2\,(r_3, r_4);$$
$$s_2\colon r_1 \Leftarrow ALU1\,(r_5, r_6);\quad r_6 \Leftarrow ALU2\,(r_2, r_5);$$
$$s_3\colon r_3 \Leftarrow ALU1\,(r_1, r_6);\ .$$

We call the interconnection topology "point-to-point" if there is only one interconnection unit between any two ports of the functional and/or storage units. The point-to-point topology is most popular in high-level synthesis since it simplifies the allocation algorithms. In this topology, we create a connection between any two functional or storage units as needed. If more than one connection is assigned to the input of a unit, a multiplexer or a bus is used. In order to minimize the number of interconnections, we can combine registers into register files with multiple ports. Each port may support read and/or write accesses to the data. Some register files allow simultaneous read and write accesses through different ports. Although register files reduce the cost of interconnection units, each port requires dedicated decoder circuitry inside the register file, which increases the storage cost and the propagation delay.

To simplify the binding problem, in this section we assume that all register transfers go through functional units and that direct interconnections of two functional units are not allowed. Therefore, we only need interconnection units to connect the output ports of storage units

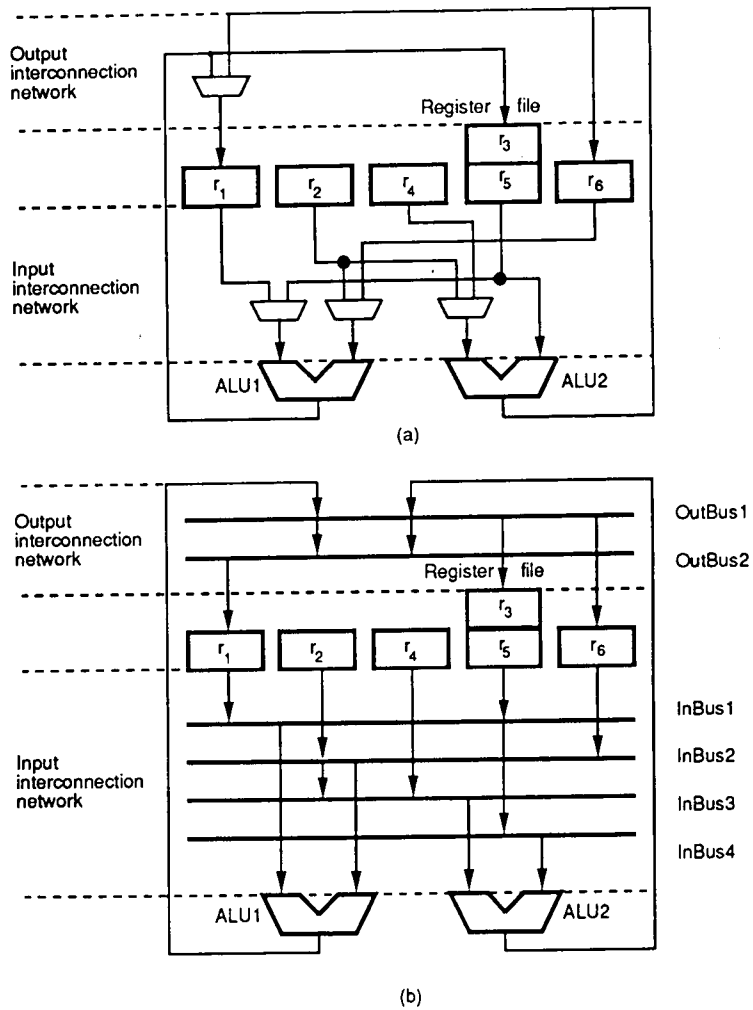

Figure 8.2: Datapath interconnections: (a) a multiplexer-oriented datapath, (b) a bus-oriented datapath.

to the input ports of functional units (i.e., the input interconnection network) and the output ports of functional units to the input ports of storage units (i.e., the output interconnection network).

The complexity of the input and output interconnection networks need not be the same. One can be simplified at the expense of the other. For example, selectors may not be allowed in front of the input ports of storage units. This results in a more complicated input interconnection network, and, hence, an imbalance between the read and write times of the storage units. In addition to affecting the allocation algorithms, such an architecture increases testability [GrDe90]. Furthermore, the number of buses that each register file drives can be constrained. If a unique bus is allocated for each register-file output, tri-state bus drivers are not needed between the registers and the buses [GeEl90]. This restriction on register-file outputs produces more multiplexers at the input ports of functional units. Moreover, some variables may need to be duplicated across different register files in order to simplify the selector circuits between the buses and the functional units.

Another interconnection scheme commonly used in processors has the buses partitioned into segments so that each bus segment is used by one functional unit at a time [Ewer90]. The functional and storage units are arranged in a single row with bus segments on either side, and so looks like a 2-bus architecture. A data transfer between the segments is achieved through switches between bus segments. Thus, we accomplish interconnection allocation by controlling the switches between the bus segments to allow or disallow data transfers.

Let us analyze the delays involved in register-to-register transfers for the five-transfer example in Figure 8.2. The relative timing of read, execute and write micro-operations in the first two clock cycles of the example are shown in Figure 8.3. Let t_r be the time delay involved for reading the data out of the registers and then propagating through the input interconnection network; t_e, the propagation delay through a functional unit; and t_w, the delay for data to propagate from the functional units through the output interconnection network and be written to the registers. In the target architectures described so far, all the components along the path from the registers' output ports back to the registers' input ports (i.e., the input interconnection network, the ALUs and the output interconnection network) are combinational. Thus, the clock

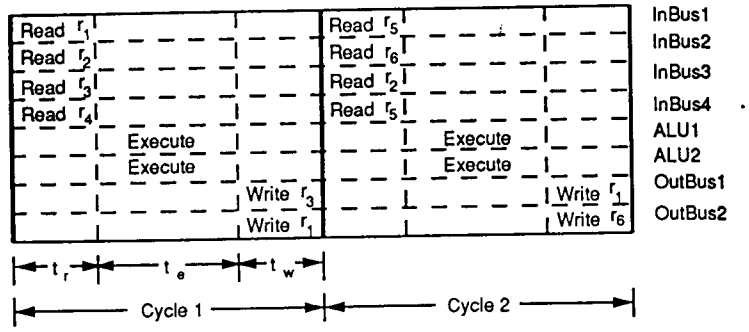

Figure 8.3: Sequential execution of three micro-operations in the same clock period.

cycle will be equal to or greater than $t_r + t_e + t_w$.

As described in Chapter 2, latches may be inserted at the input and/or output ports of functional units to improve the datapath performance. When latches are inserted only at the outputs of the functional units (Figure 8.4), the read accesses and functional-unit execution for operations scheduled into the current control step can be performed at the same time as the write accesses of operations scheduled into the previous control step (Figure 8.5). The cycle period is reduced to $\max(t_r + t_e, t_w)$. But the register transfers are not well balanced: reading of the registers and execution of an ALU operation are performed in the first cycle, while only writing of the result back into the registers is performed during the second cycle. Similarly, if only the inputs of the functional units are latched, the read accesses for operations scheduled into the next control step can be performed at the same time as the functional unit-execution and the write accesses for operations scheduled into the current control step. The cycle period in that case will be $\max(t_r, t_e + t_w)$. In either case, the register files and latches are controlled by a single-phase clock.

Operation execution and the reading/writing of data can take place

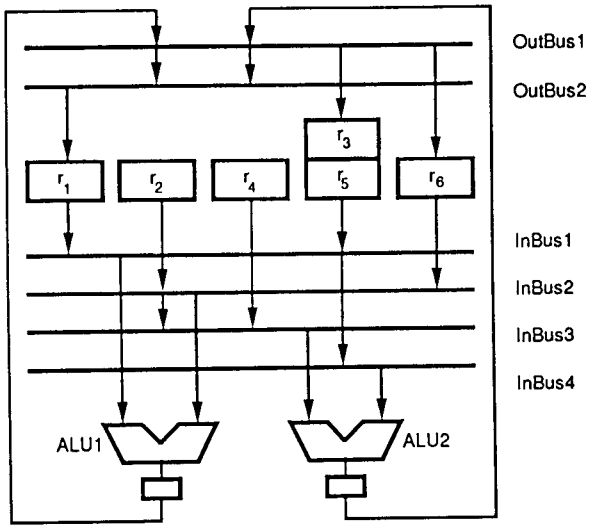

Figure 8.4: Insertion of latches at the output ports of the functional units.

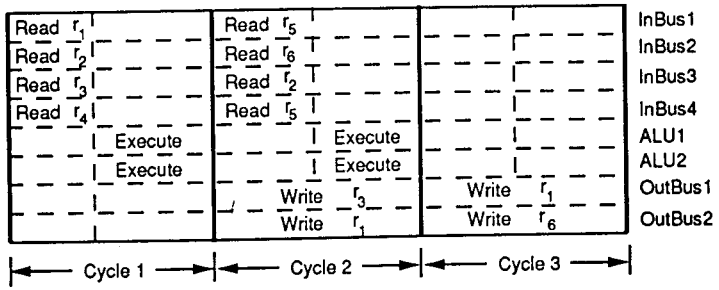

Figure 8.5: Overlapping read and write data transfers.

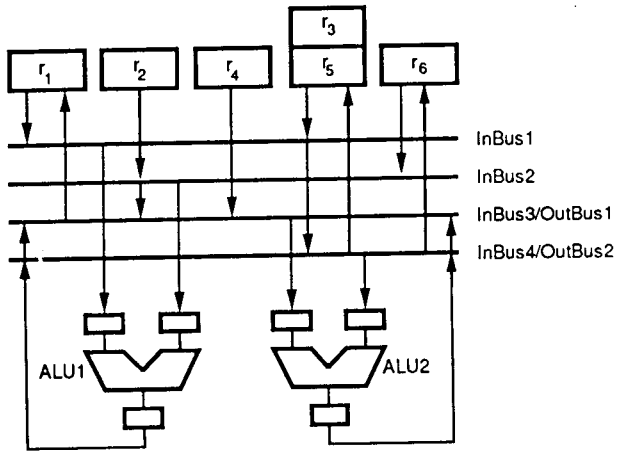

Figure 8.6: Insertion of latches at both the input and output ports of the functional units.

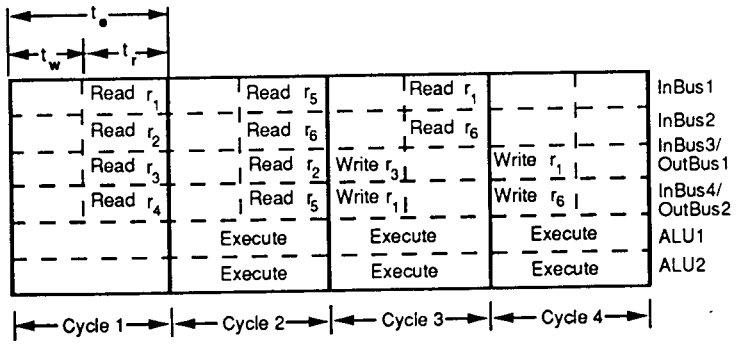

Figure 8.7: Overlapping data transfer with functional-unit execution.

concurrently when both the inputs and outputs of a functional unit are latched (Figure 8.6). The three combinational components, namely, the input-interconnection units, the functional units and the output-interconnection units, can all be active concurrently. Figure 8.7 shows how this pipelining scheme works. The clock cycle consists of two minor cycles. The execution of an operation is spread across three consecutive clock cycles. The input operands for an operation are transferred from the register files to the input latches of the functional units during the second minor cycle of the first cycle. During the second cycle, the functional unit executes the operation and writes the result to the output latch by the end of the cycle. The result is transferred to the final destination, the register file, during the first minor cycle of the third cycle. A two-phase non-overlapping clocking scheme is needed. Both the input and output latches are controlled by one phase since the end of the read access and the end of operation execution occur simultaneously. The other phase is used to control the write accesses to the register files.

By overlapping the execution of operations in successive control steps, we can greatly increase the hardware utilization. The cycle period is reduced to $\max(t_e, t_r + t_w)$. Moreover, the input and output networks can share some interconnection units. For example, *OutBus1* is merged with *InBus3* and *OutBus2* with *InBus4* in Figure 8.6.

Thus, inserting input and output latches makes available more interconnection units for merging, which may simplify the datapath design. By breaking the register-to-register transfers into micro-operations executed in different clock cycles, we achieve a better utilization of hardware resources. However, this scheme requires binding algorithms to search through a larger number of design alternatives.

Operator chaining was introduced in Section 7.3.1 as the execution of two or more operations in series during the same control step. To support operation chaining, links are needed from the output ports of some functional units directly to the input ports of other functional units. In an architecture with a shared bus (e.g., Figure 8.6), this linking can be accomplished easily by using the path from a functional unit's output port through one of the buses to some other functional unit's input port. Since such a path must be combinational, bypass circuits have to be added around all the latches along the path of chaining.

8.3 Allocation Tasks

Datapath synthesis consists of four different yet interdependent tasks: module selection, functional-unit allocation, storage allocation and interconnection allocation. In this section, we define each task and discuss the nature of their interdependence.

8.3.1 Unit Selection

A simple design model may assume that we have only one particular type of functional unit for each behavioral operation. However, a real RT component library contains multiple types of functional units, each with different characteristics (e.g., functionality, size, delay and power dissipation) and each implementing one or several different operations in the register-transfer description. For example, an addition can be carried out by either a small but slow ripple adder or by a large but fast carry look-ahead adder. Furthermore, we can use several different component types, such as an adder, an adder/subtracter or an entire ALU, to perform an addition operation. Thus, unit selection selects the number and types of different functional and storage units from the component library. A basic requirement for unit selection is that the number of units performing a certain type of operation must be equal to or greater than the maximum number of operations of that type to be performed in any control step. Unit selection is frequently combined with binding into one task called allocation.

8.3.2 Functional-Unit Binding

After all the functional units have been selected, operations in the behavioral description must be mapped into the the set of selected functional units. Whenever we have operations that can be mapped into more than one functional unit, we need a functional-unit binding algorithm to determine the exact mapping of the operations into the functional units. For example, operations o_1 and o_3 in Figure 8.1 have been mapped into adder *ADD1*, while the operations o_2 and o_4 have been mapped into adder *ADD2*.

8.3.3 Storage Binding

Storage binding maps data carriers (e.g., constants, variables and data structures like arrays) in the behavioral description to storage elements (e.g., ROMs, registers and memory units) in the datapath. Constants, such as coefficients in a DSP algorithm, are usually stored in a read-only memory (ROM). Variables are stored in registers or memories. Variables whose lifetime intervals do not overlap with each other may share the same register or memory location. The lifetime of a variable is the time interval between its first value assignment (the first variable appearance on the left-hand side of an assignment statement) and its last use (the last variable appearance on the right-hand side of an assignment statement). After variables have been assigned to registers, the registers can be merged into a register file with a single access port if the registers in the file are not accessed simultaneously. Similarly, registers can be merged into a multiport register file as long as the number of registers accessed in each control step does not exceed the number of ports.

8.3.4 Interconnection Binding

Every data transfer (i.e., a read or write) needs an interconnection path from its source to its sink. Two data transfers can share all or part of the interconnection path if they do not take place simultaneously. For example, in Figure 8.1, the reading of variable b in control step s_1 and variable e in control step s_2 can be achieved by using the same interconnection unit. However, writing to variables e and f, which occurs simultaneously in control step s_1, must be accomplished using disjoint paths. The objective of interconnection binding is to maximize the sharing of interconnection units and thus minimize the interconnection cost, while still supporting the conflict-free data transfers required by the register-transfer description.

8.3.5 Interdependence and Ordering

All the datapath synthesis tasks (i.e., scheduling, unit selection, functional unit binding, storage binding and interconnection binding) depend on each other. In particular, functional-unit, storage and inter-

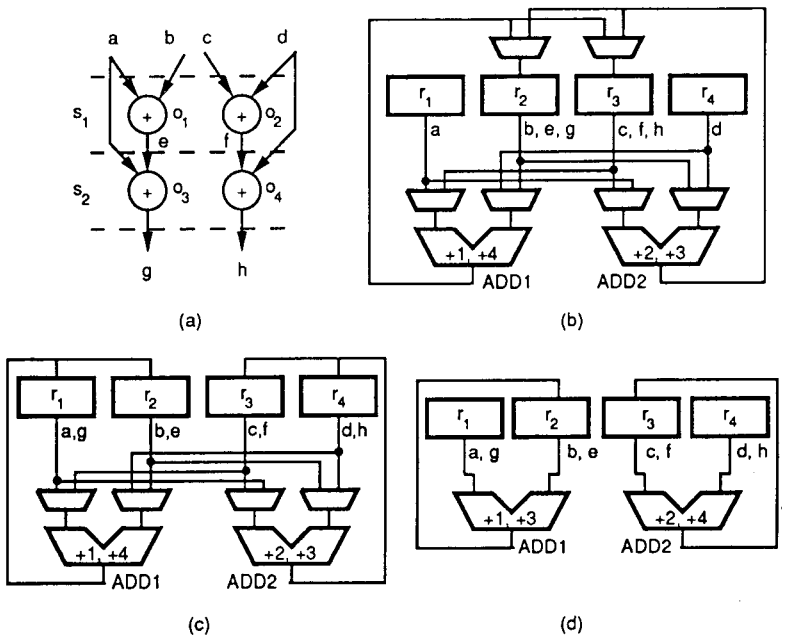

Figure 8.8: Interdependence of functional-unit and storage binding: (a) a scheduled DFG, (b) a functional-unit binding requiring six multiplexers, (c) improved design with two fewer multiplexers, obtained by register reallocation, (d) optimal design with no multiplexers, obtained by modifying functional-unit binding.

connection binding are tightly related to each other. For example, Figure 8.8 shows how both functional unit and storage binding affect interconnection allocation. Suppose the eight variables (a through g) in the DFG of Figure 8.8(a) have been partitioned into four registers as follows: $r_1 \leftarrow \{a\}$, $r_2 \leftarrow \{b, e, g\}$, $r_3 \leftarrow \{c, f, h\}$, $r_4 \leftarrow \{d\}$. Given two adders, *ADD1* and *ADD2*, there are two ways of grouping the four addition operations, o_1, o_2, o_3, and o_4, in Figure 8.8(a) so that each group is assigned to one adder:

(1) *ADD1* $\leftarrow \{o_1, o_4\}$, *ADD2*$\leftarrow \{o_2, o_3\}$, or

(2) *ADD1* $\leftarrow \{o_1, o_3\}$, *ADD2*$\leftarrow \{o_2, o_4\}$.

For the given register binding, we need six 2-to-1 multiplexers for unit interconnection in case (1) (Figure 8.8(b)). However, we can eliminate two 2-to-1 multiplexers (Figure 8.8(c)), by modifying the register binding to be : $r_1 \leftarrow \{a, g\}$, $r_2 \leftarrow \{b, e\}$, $r_3 \leftarrow \{c, f\}$, $r_4 \leftarrow \{d, h\}$. If we then modify the functional-unit binding to case (2) above, no multiplexers are needed (Figure 8.8(d)). Thus, this design is optimal if the interconnection cost is measured by the number of multiplexers needed. Clearly, both functional-unit and storage binding potentially affect the optimization achievable by interconnection allocation.

The previous example also raises the issue of ordering among the allocation tasks. The requirements on interconnection become clear after both functional-unit and storage allocation have been performed. Furthermore, functional-unit allocation can make correct decisions if storage allocation is done beforehand, and vice versa. To break this deadlock situation, we choose one task ahead of the other. Unfortunately, in such a ordering, the first task chosen cannot use the information from the second task, which would have been available had the second task been performed first.

8.4 Greedy Constructive Approaches

We can construct a datapath using a greedy approach in which RT components are assigned to operations in a step-by-step fashion [KuPa90].

A constructive algorithm starts with an empty datapath and builds the datapath gradually by adding functional, storage and interconnection units as necessary. For each operation, it tries to find a functional unit on the partially designed datapath that is capable of executing the operation and is idle during the control step in which the operation must be executed. In case there are two or more functional units that meet these conditions, we choose the one which results in a minimal increase in the interconnection cost. On the other hand, if none of the functional units on the partially designed datapath meet the conditions, we add a new functional unit from the component library that is capable of carrying out the operation. Similarly, we can assign a variable to an available register only if its lifetime interval does not overlap with those of variables already assigned to that register. A new register is allocated only when no allocated register meets the above condition. Again, when multiple alternatives exist for assignment of a variable to a register, we select the one that minimally increases the datapath cost.

Figures 8.9(b)-(g) show several modified datapaths obtained by adding interconnection and/or functional units to the partial design of Figure 8.9(a). We discuss each case below:

1. In Figure 8.9(b) we add a new connection from each of r_3 and *Bus1* to the left input of *ALU1*.
2. In Figure 8.9(c) we add a connection from r_3 to the right input of *ALU2* through *Bus1*. Since the connection from *Bus1* to the right input of *ALU2* already exists, we add only a tri-state buffer.
3. Figure 8.9(d) is similar to Figure 8.9(c), except that we add the multiplexer *Mux2* instead of a tri-state buffer.
4. In Figure 8.9(e) we add a new functional unit, *ALU3*, and a new connection from each of r_3 and *Bus1* to the right input of *ALU3*.
5. Figure 8.9(f) is similar to Figure 8.9(e), except that we add the multiplexer *Mux2* instead of a tri-state buffer.
6. In Figure 8.9(g) we merge *Mux1* and *Bus1* into a single bus, *Bus1*.

Interconnection allocation follows immediately after both the source and sink of a data transfer have been bound. For example, the partial

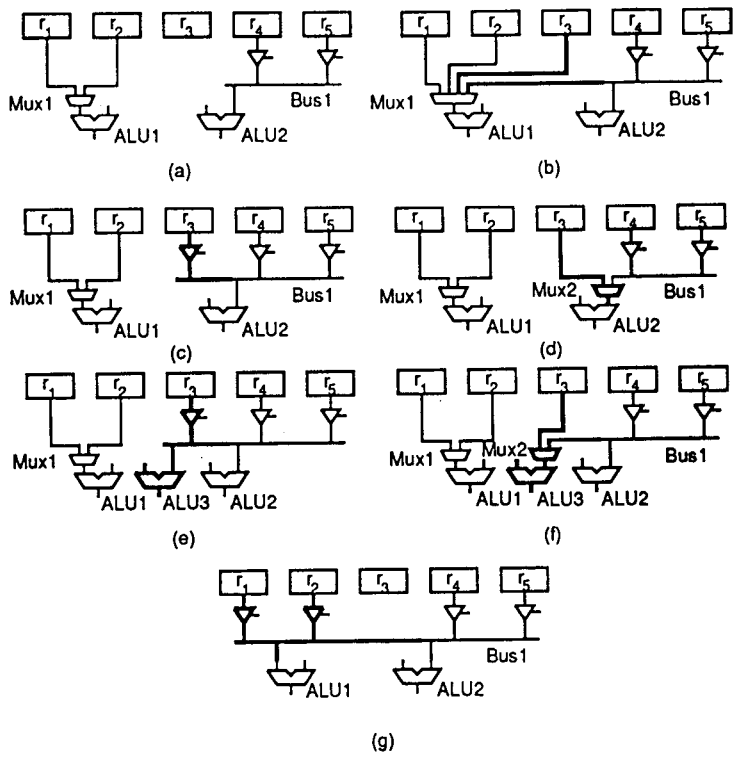

Figure 8.9: Datapath construction: (a) an initial partial design, (b) addition of two more inputs to a multiplexer, (c) addition of a tri-state buffer to a bus, (d) addition of a multiplexer to the input of a functional unit, (e) addition of a functional unit and a tri-state buffer to a bus, (f) addition of a functional unit and a multiplexer, (g) conversion of a multiplexer to a shared bus.

datapath in Figure 8.9(a) does not have a link between register r_3 and *ALU2*. Suppose a variable that is one of the inputs to an operation that has been assigned to *ALU2* is just assigned to register r_3. Then, an interconnection link has to be established as shown by the bold wires in Figure 8.9(c). Each interconnection unit that is required by a data transfer in the behavioral description contributes to the datapath cost. For example, in Figure 8.9(b), two more connections are made to the left input port of *ALU1*, where a two-input multiplexer already exists. Therefore, the cost of this modification is the difference between the cost of a two-input multiplexer and that of a four-input multiplexer. However, at least two additional data transfers are now supported with this modification.

Algorithm 8.1 describes the greedy constructive allocation method. Let UBE be the set of unallocated behavioral entities and $DP_{current}$ be the partially designed datapath. The behavioral entities being considered could be variables that have to be mapped into registers, operations that have to be mapped into functional units, or data transfers that have to be mapped into interconnection units. $DP_{current}$ is initially empty. The procedure ADD(DP, *ube*) structurally modifies the datapath DP by adding to it the components necessary to support the behavioral entity *ube*. The function COST(DP) evaluates the area/performance cost of a partially designed datapath DP. DP_{work} is a temporary datapath which is created in order to evaluate the cost c_{work} of performing each modification to $DP_{current}$.

Starting with the set UBE, the inner *for* loop determines which unallocated behavioral entity, *BestEntity*, requires the minimal increase in the cost when added to the datapath. This is accomplished by adding each of the unallocated behavioral entities in UBE to $DP_{current}$ individually and then evaluating the resulting cost. The procedure ADD then modifies $DP_{current}$ by incorporating *BestEntity* into the datapath. *BestEntity* is deleted from the set of unallocated behavioral entities. The algorithm iterates in the outer *while* loop until all behavioral entities have been allocated (i.e., $UBE = \phi$).

In order to use the greedy constructive approach, we have to address two basic issues: the cost-function calculation and the order in which the unallocated behavioral entities are mapped into the datapath. The costs can be computed as explained in Chapter 3. For example, the cost

Algorithm 8.1: Constructive Allocation.

```
DP_current = φ;
while UBE ≠ φ do
   LowestCost = ∞;

   for all ube ∈ UBE do
      DP_work = ADD(DP_current, ube);
      c_work = COST(DP_work);
      if c_work < LowestCost then
         LowestCost = c_work;
         BestEntity = ube;
      endif
   endfor

   DP_current = ADD(DP_current, BestEntity);
   UBE = UBE − BestEntity;
endwhile
```

of converting the datapath in Figure 8.9(a) to the one in Figure 8.9(g) depends on the difference between the cost of one 2-to-1 multiplexer and that of three tri-state buffers. Since the buses of Figure 8.9(a) and Figure 8.9(g) are of different length, the two datapaths may also have different wiring costs.

The order in which unallocated entities are mapped into the datapath can be determined either statically or dynamically. In a static approach, the objects are ordered before the datapath construction begins. The ordering is not changed during the construction process. By contrast, in a dynamic approach no ordering is done beforehand. To select an operation or variable for binding to the datapath, we evaluate every unallocated behavioral entity in terms of the cost involved in modifying the partial datapath, and the entity that requires the least expensive modification is chosen. After each binding, we revaluate the costs associated with the remaining unbound entities. Algorithm 8.1 uses the dynamic strategy.

In hardware sharing, a previously expensive binding may become inexpensive after some other bindings are done. Therefore, a good strategy incorporates a look-ahead factor into the cost function. That is, the cost of a modification to the datapath should be lower if it decreases

the cost of other bindings done in the future.

8.5 Decomposition Approaches

The intuitive approach taken by the constructive method falls into the category of greedy algorithms. Although greedy algorithms are simple, the solutions they find can be far from optimal. In order to improve the quality of the results, some researchers have proposed a decomposition approach, where the allocation process is divided into a sequence of independent tasks; each task is transformed into a well-defined problem in graph theory and then solved with a proven technique.

While a greedy constructive approach like the one described in Algorithm 8.1 might interleave the storage, functional-unit, and interconnection allocation steps, decomposition methods will complete one task before performing another. For example, all variable-to-register assignments might be completed before any operation-to-functional-unit assignments are performed, and vice versa. Because of interdependencies among these tasks, no optimal solution is guaranteed even if all the tasks are solved optimally. For example, a design using three adders may need one fewer multiplexer than the one using only two adders. Therefore, an allocation strategy that minimizes the usage of adders is justified only when an adder costs more than a multiplexer.

In this section we describe allocation techniques based on three graph-theoretical methods: clique partitioning, left-edge algorithm and the weighted bipartite matching algorithm. For the sake of simplicity, we illustrate these allocation techniques by applying them to behavioral descriptions consisting of straight-line code without any conditional branches.

8.5.1 Clique Partitioning

The three tasks of storage, functional-unit and interconnection allocation can be solved independently by mapping each task to the well known problem of graph clique-partitioning [TsSi86].

We begin by defining the clique-partitioning problem. Let $G = (V, E)$

denote a graph, where V is the set of vertices and E the set of edges. Each edge $e_{i,j} \in E$ links two different vertices v_i and $v_j \in V$. A subgraph SG of G is defined as (SV, SE), where $SV \subseteq V$ and $SE = \{e_{i,j} \mid e_{i,j} \in E, v_i, v_j \in SV\}$. A graph is complete if and only if for every pair of its vertices there exists an edge linking them. A clique of G is a complete subgraph of G. The problem of partitioning a graph into a minimal number of cliques such that each node belongs to exactly one clique is called clique partitioning. The clique-partitioning problem is a classic NP-complete problem for which heuristic procedures are usually used.

Algorithm 8.2 describes a heuristic proposed by Tseng and Siewiorek [TsSi86] to solve the clique-partitioning problem. A super-graph $G'(S, E')$ is derived from the original graph $G(V, E)$. Each node $s_i \in S$ is a super-node that can contain a set of one or more vertices $v_i \in V$. E' is identical to E except that the edges in E' now link super-nodes in S. A super-node $s_i \in S$ is a common neighbor of the two super-nodes s_j and $s_k \in S$ if there exist edges $e_{i,j}$ and $e_{i,k} \in E'$. The function COMMON_NEIGHBOR(G', s_i, s_j) returns the set of super-nodes that are common neighbors of s_i and s_j in G'. The procedure DELETE_EDGE(E', s_i) deletes all edges in E' which have s_i as their end super-node.

Initially, each vertex $v_i \in V$ of G is placed in a separate super-node $s_i \in S$ of G'. At each step, the algorithm finds the super-nodes s_{Index1} and s_{Index2} in S such that they are connected by an edge and have the maximum number of common neighbors. These two super-nodes are merged into a single super-node, $s_{Index1Index2}$, which contains all the vertices of s_{Index1} and s_{Index2}. The set $CommonSet$ contains all the common neighbors of s_{Index1} and s_{Index2}. All edges originating from s_{Index1} or s_{Index2} in G' are deleted. New edges are added from $s_{Index1Index2}$ to all the super-nodes in $CommonSet$. The above steps are repeated until there are no edges left in the graph. The vertices contained in each super-node $s_i \in S$ form a clique of the graph G.

Figure 8.10 illustrates the above algorithm. In the graph of Figure 8.10(a), $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{e_{1,3}, e_{1,4}, e_{2,3}, e_{2,5}, e_{3,4}, e_{4,5}\}$. Initially, each vertex is placed in a separate super-node (labeled s_1 through s_5 in Figure 8.10(b)). The three edges, $e'_{1,3}$, $e'_{1,4}$ and $e'_{3,4}$, of the super-graph G' have the maximum number of common neighbors among all edges (Figure 8.10(b)). The first edge, $e'_{1,3}$, is selected and the

Algorithm 8.2: Clique Partitioning.

```
/* create a super graph G'(S, E') */
S = φ;    E' = φ;
for each v_i ∈ V do   s_i = {v_i}; S = S ∪ {s_i};   endfor
for each e_{i,j} ∈ E do   E' = E' ∪ {e'_{i,j}};   endfor

while E' ≠ φ do

   /* find s_Index1, s_Index2 having most common neighbors */
   MostCommons = −1;
   for each e'_{i,j} ∈ E' do
      c_{i,j} = | COMMON_NEIGHBOR(G', s_i, s_j) |;
      if c_{i,j} > MostCommons then
         MostCommons = c_{i,j};
         Index1 = i;    Index2 = j;
      endif
   endfor

   CommonSet = COMMON_NEIGHBOR(G', s_Index1, s_Index2);

   /* delete all edges linking s_Index1 or s_Index2 */
   E' = DELETE_EDGE(E', s_Index1);
   E' = DELETE_EDGE(E', s_Index2);

   /* merge s_Index1 and s_Index2 into s_Index1Index2 */
   s_Index1Index2 = s_Index1 ∪ s_Index2;
   S = S − s_Index1 − s_Index2;
   S = S ∪ {s_Index1Index2};

   /* add edge from s_Index1Index2 to super-nodes in CommonSet */
   for each s_i ∈ CommonSet do
      E' = E' ∪ {e'_{i,Index1Index2}};
   endfor

endwhile
```

following steps are carried out to yield the graph of Figure 8.10(c).

(1) s_4, the only common neighbor of s_1 and s_3 is put in *CommonSet*.

(2) All edges are deleted that link either super-nodes s_1 or s_3 (i.e., $e'_{1,3}$, $e'_{1,4}$, $e'_{2,3}$ and $e'_{3,4}$).

(3) Super-nodes s_1 and s_3 are combined into a new super-node s_{13}.

(4) An edge is added between s_{13} and each super-node in *CommonSet*; i.e., the edge $e_{13,4}$ is added.

On the next iteration, s_4 is merged into s_{13} to yield the super-node s_{134} (Figure 8.10(d)). Finally, s_2 and s_5 are merged into the super-node s_{25} (Figure 8.10(e)). The cliques are $s_{134} = \{v_1, v_3, v_4\}$ and $s_{25} = \{v_2, v_5\}$ (Figure 8.10(f)).

In order to apply the clique partitioning technique to the allocation problem, we have to first derive the graph model from the input description. Consider register allocation as an example. The primary goal of register allocation is to minimize the register cost by maximizing the sharing of common registers among variables. To solve the register allocation problem, we construct a graph $G = (V, E)$, in which every vertex $v_i \in V$ uniquely represents a variable v_i and there exists an edge $e_{i,j} \in E$ if and only if variables v_i and v_j can be stored in the same register (i.e., their lifetime intervals do not overlap). All the variables whose representative vertices are in a clique of G can be stored in a single register. A clique partitioning of G provides a solution for the datapath storage-allocation problem that requires a minimal number of registers. Figure 8.11 shows a solution of the register-allocation problem using the clique-partitioning algorithm.

Both functional-unit allocation and interconnection allocation can be formulated as a clique-partitioning problem. For functional-unit allocation, each graph vertex represents an operation. An edge exists between two vertices if two conditions are satisfied:

(1) the two operations are scheduled into different control steps, and

(2) there exists a functional unit that is capable of carrying out both operations.

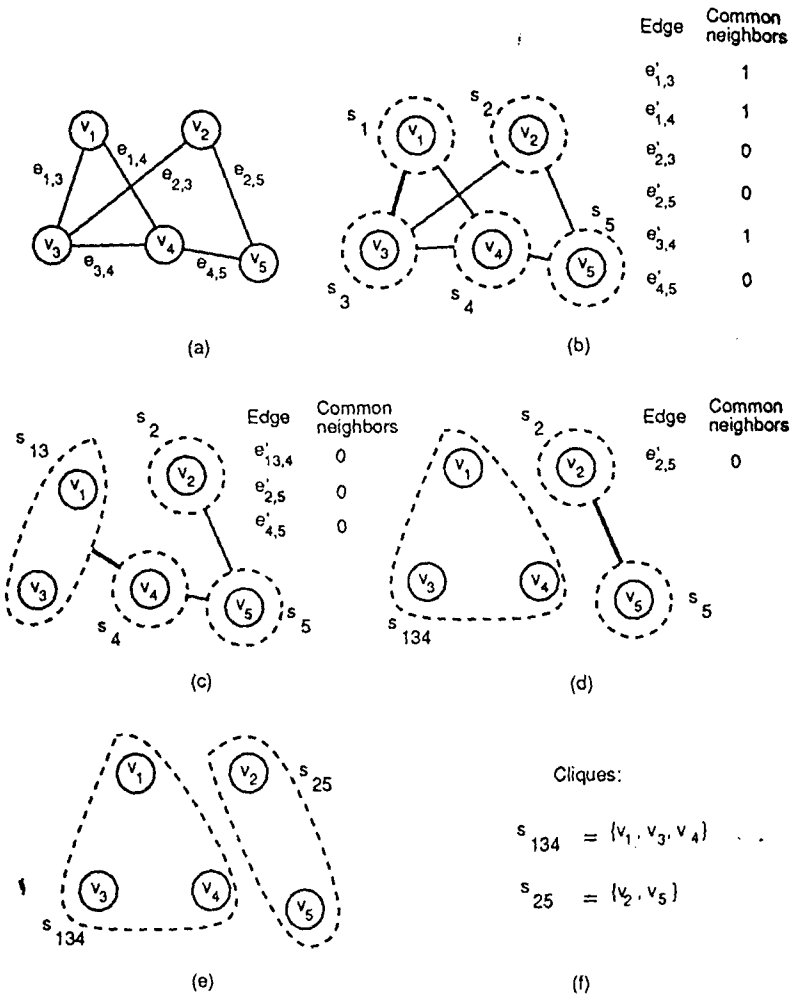

Figure 8.10: Clique partitioning: (a) given graph G, (b) calculating the common neighbors for the edges of graph G', (c) super-node s_{13} formed by considering edge $e'_{1,3}$, (d) super-node s_{134} formed by considering edge

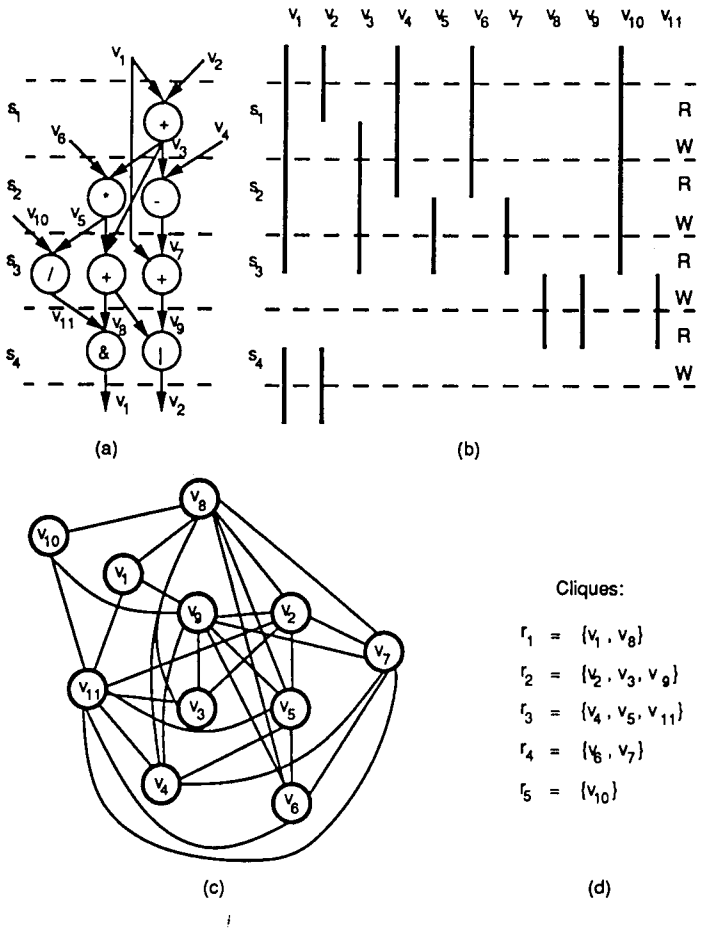

Figure 8.11: Register allocation using clique partitioning: (a) a scheduled DFG, (b) lifetime intervals of variables, (c) the graph model for register allocation, (c) a clique-partitioning solution.

A clique-partitioning solution of this graph would yield a solution for the functional-unit allocation problem. Since a functional unit is assigned to each clique, all operations whose representative vertices are in a clique are executed in the same functional unit.

For interconnection-unit allocation, each vertex corresponds to a connection between two units, whereas an edge links two vertices if the two corresponding connections are not used concurrently in any control step. A clique-partitioning solution of such a graph implies partitioning of connections into buses or multiplexers. In other words, all connections whose representative vertices are in the same clique use the same bus or multiplexer.

Although the clique-partitioning method when applied to storage allocation can minimize the storage requirements, it totally ignores the interdependence between storage and interconnection allocation. Paulin and Knight [PaKn89] extend the previous method by augmenting the graph edges with weights that reflect the impact on interconnection complexity due to register sharing among variables. An edge is given a higher weight if sharing of a register by the two variables corresponding to the edge's two end vertices reduces the interconnection cost. On the other hand, an edge is given a lower weight if the sharing causes an increase in the interconnection cost. The modified algorithm prefers cliques with heavier edges. Hence, variables that share a common register are more likely to reduce the interconnection cost.

8.5.2 Left-Edge Algorithm

The left-edge algorithm [HaSt71] is well known for its application in channel-routing tools for physical-design automation. The goal of the channel routing problem is to minimize the number of tracks used to connect points on the channel boundary. Two points on the channel boundary are connected with one horizontal (i.e., parallel to the channel) and two vertical (i.e., orthogonal to the channel) wire segments. Since the channel width depends on the number of horizontal tracks used, the channel-routing algorithms try to pack the horizontal segments into as few tracks as possible. Kurdahi and Parker [KuPa87] apply the left-edge algorithm to solve the register-allocation problem, in which variable lifetime intervals correspond to horizontal wire segments and registers to

wiring tracks.

The input to the left-edge algorithm is a list of variables, L. A lifetime interval is associated with each variable. The algorithm makes several passes over the list of variables until all variables have been assigned to registers. Essentially, the algorithm tries to pack the total lifetime of a new register allocated in each pass with as many variables whose lifetimes do not overlap, by using the channel-routing analogy of packing horizontal segments into as few tracks as possible.

Algorithm 8.3 describes register allocation using the left-edge algorithm. If there are n variables in the behavioral description, we define L to be the list of all variables v_i, $1 \leq i \leq n$. Let the 2-tuple $<Start(v), End(v)>$ represent the lifetime interval of a variable v where $Start(v)$ and $End(v)$ are respectively the start and end times of its lifetime interval. The procedure SORT(L) sorts the variables in L in ascending order with their start times, $Start(v)$, as the primary key and in descending order with their end times, $End(v)$, as the secondary key. The procedure DELETE (L, v) deletes the variable v from list L, FIRST(L) returns the first variable in the sorted list L and NEXT(L, v) returns the variable following v in list L. The array MAP keeps track of the registers assigned to each variable. The value of *reg_index* represents the index of the register being allocated in each pass. The end time of the interval of the most recently assigned variable in that pass is contained in *last*.

Initially, the variables are not assigned to any of the registers. During each pass over L, variables are assigned to a new register, r_{reg_index}. The first variable from the sorted list whose lifetime does not overlap with the lifetime of any other variables assigned to r_{reg_index} is assigned to the same register. When a variable is assigned to a register, the register is entered in the array MAP for that variable. On termination of the algorithm, the array MAP contains the registers assigned to all the variables and *reg_index* represents the total number of registers allocated.

Figure 8.12(a) depicts the sorted list of the lifetime intervals of the variables of the DFG in Figure 8.11(a). Note that variables v_1 and v_2 in Figure 8.11(a) are divided into two variables each (v_1, v_1' and v_2, v_2') in order to obtain a better packing density. Figure 8.12(b) illustrates how the left-edge algorithm works on the example. Starting with a new empty register r_1, the first variable in the sorted list, v_1, is put into r_1. Traveling

Algorithm 8.3: Register Allocation using Left-Edge Algorithm.

```
for all v ∈ L do  MAP[v] = 0;  endfor
SORT(L);

reg_index = 0;
while L ≠ φ do
   reg_index = reg_index + 1;
   curr_var = FIRST(L);
   last = 0;

   while curr_var ≠ null do
      if Start(curr_var) ≥ last then
         MAP[curr_var] = r_reg_index;
         last = End(curr_var);

         temp_var = curr_var;
         curr_var = NEXT(L, curr_var);
         DELETE(L, temp_var);
      else
         curr_var = NEXT(L, curr_var);
      endif
   endwhile

endwhile
```

down the list, no variables can be packed into r_1 before v_8 is encountered. After packing v_8 into r_1, the next packable variable down the list is v_1'. No more variables can be assigned to r_1 without overlapping variable lifetimes. Hence the algorithm allocates a new register (r_2) and starts from the beginning of the list again. The sorted list now has three fewer variables than it had in the beginning (i.e., v_1, v_8 and v_1' have been removed). The list becomes empty after five registers have been allocated.

Unlike the clique-partitioning problem, which is NP-complete, the left-edge algorithm has a polynomial time complexity. Moreover, this algorithm allocates the minimum number of registers [KuPa87]. However, it cannot take into account the impact of register allocation on the interconnection cost, as can the weighted version of the clique-partitioning

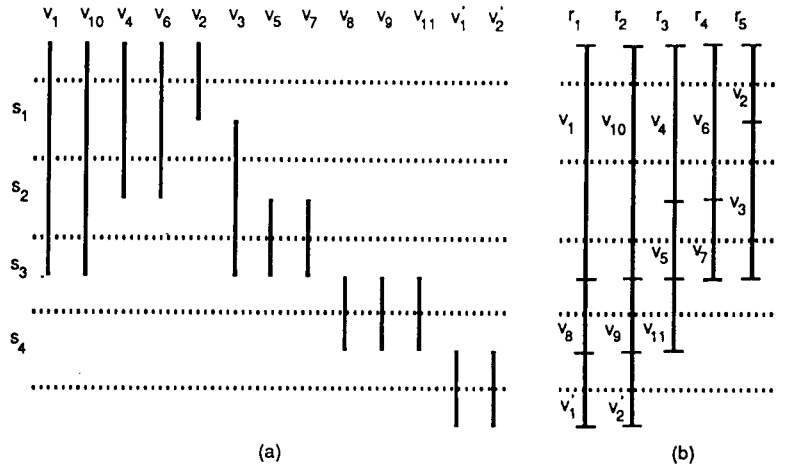

Figure 8.12: Register allocation using the left-edge algorithm: (a) sorted variable lifetime intervals, (b) five-register allocation result.

algorithm.

8.5.3 Weighted Bipartite-Matching Algorithm

Both the register and functional-unit allocation problems can be transformed into a weighted bipartite-matching algorithm [HCLH90]. Unlike the left-edge algorithm, which binds variables sequentially to one register at a time, the bipartite-matching algorithm binds multiple variables to multiple registers simultaneously. Moreover, it takes interconnection cost into account during allocation of registers and functional units.

Algorithm 8.4 describes the weighted bipartite-matching method. Let L be the list of all variables as defined in the description of the left-edge algorithm. The number of required registers, ***num_reg***, is the maximum density of the lifetime intervals (i.e., the maximum number of overlapping variable lifetimes). Let R be the set of nodes representing

the registers $\{r_1, r_2, ..., r_{num\_reg}\}$ into which the set of variables, V, will be mapped. The set of variables is partitioned into clusters of variables with mutually overlapping lifetimes. The function OVERLAP($Cluster_i$, v) returns the value true if either $Cluster_i$ is empty or if the lifetime of variable v overlaps with the lifetimes of all the variables already assigned to $Cluster_i$. The function BUILD_GRAPH (R,V) returns the set of edges E containing edges between nodes of the two sets R, representing registers, and V, representing variables in a particular cluster. An edge $e_{i,j} \in E$ will represents feasible assignment of variable v_j to a register r_i if and only if the lifetime of v_j does not overlap with the lifetime of any variable already assigned to r_i. The procedure MATCHING($G(R \cup V, E)$) finds the subset $E' \subseteq E$, which represents a bipartite-matching solution of the graph G. Each edge $e_{i,j}$ in E' represents the assignment of a variable $v_j \in V$ to register $r_i \in R$. No two edges in E' share a common end-node (i.e., no two variables in the same cluster can be assigned to the same register). The array MAP, procedures SORT and DELETE, and the function FIRST are identical to that of the left-edge algorithm.

After sorting the list of variables according to their lifetime intervals in the same way as the left-edge algorithm, this algorithm divides them into clusters of variables. The lifetime of each variable in a cluster overlaps with the lifetimes of all the other variables in the same cluster (Figure 8.13(a)). In the figure, the maximum number of overlapping lifetimes is five. Thus, the set of registers, R, will have five registers (r_1, r_2, r_3, r_4 and r_5) into which all the variables will be mapped. A cluster of variables is assigned to the set of registers simultaneously. For example, the first cluster of five variables v_1, v_{10}, v_4, v_6 and v_2, shown in Figure 8.13(b), have been assigned to the registers r_1, r_2, r_3, r_4 and r_5 respectively. The algorithm then tries to assign the second cluster of three variables, v_3, v_5 and v_7, to the registers.

A variable can be assigned to a register only if its lifetime does not overlap with the lifetimes of all the variables already assigned to that register. In Figure 8.13(b), each graph edge represents a possible variable-to-register assignment. As in the clique-partitioning algorithm, weights can be associated with the edges. An edge $e_{i,j}$ is given a higher weight if the assignment of v_j to r_i results in a reduced interconnection cost. For example, let variable v_m be bound to register r_n. If another variable

Algorithm 8.4: Register Allocation using Bipartite Matching.

```
for all v ∈ L do  MAP[v] = 0;  endfor
SORT(L);
/* divide variables into clusters */
clus_num = 0;
while L ≠ φ do
   clus_num = clus_num + 1;
   Cluster_clus_num = φ ;
   while (L ≠ φ) and OVERLAP(Cluster_clus_num, FIRST(L)) do
      Cluster_clus_num = Cluster_clus_num ∪ {FIRST(L)};
      L = DELETE(L, FIRST(L));
   endwhile
endwhile
/* allocate registers for one cluster of variables at a time */
for k = 1 to clus_num do
   V = Cluster_k;
   E = BUILD_GRAPH(R, V);
   E' = MATCHING(G(R ∪ V, E));
   for each e_{i,j} ∈ E', where v_j ∈ V and r_i ∈ R do
      MAP[v_j] = r_i;
   endfor
endfor
```

v_k is to be used by the same functional units that also use variable v_m, then since the two variables can share the same interconnections, it is desirable that v_k also be assigned to r_n.

In a bipartite graph, the node set is partitioned into two disjoint subsets and every edge connects two nodes in different subsets. The graph depicted in Figure 8.13(b) is bipartite. It has two sets, the set of registers, $R = \{r_1, r_2, r_3, r_4, r_5\}$ and the set of variables, $V = \{v_3, v_5, v_7\}$. The graph also has a set of edges $E = \{e_{3,5}, e_{3,7}, e_{4,5}, e_{4,7}, e_{5,3}, e_{5,5}, e_{5,7}\}$ returned by the function BUILD_GRAPH. The problem of matching each variable to a register is equivalent to the classic job-assignment problem. The largest subset of edges that do not have any common end-nodes is

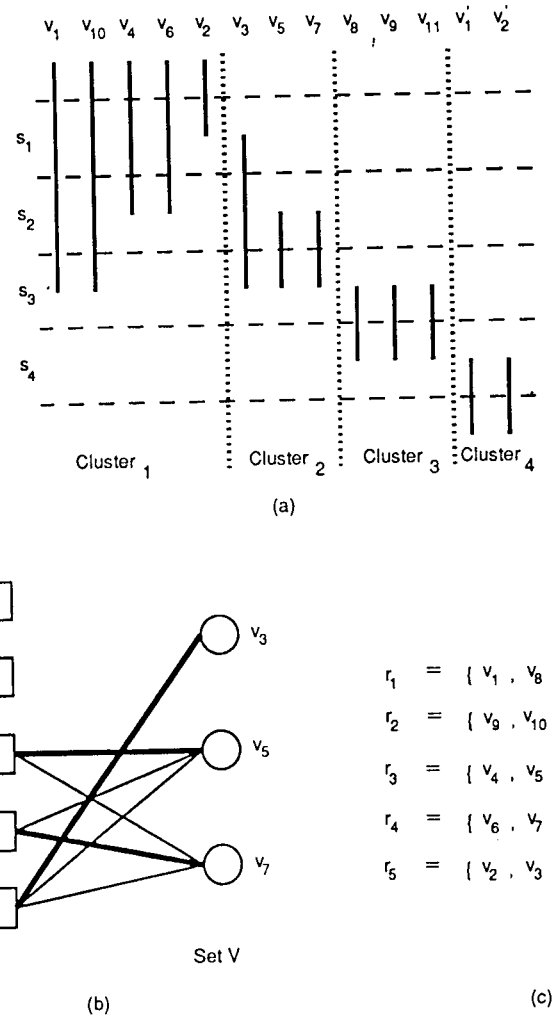

Figure 8.13: Weighted bipartite-matching for register allocation: (a) sorted lifetime intervals with clusters, (b) bipartite graph for binding of variables in $Cluster_2$ after $Cluster_1$ has been assigned to the registers,

defined as the maximal matching of the graph. The maximal edge-weight matching of a graph is the maximal matching with the largest sum of the weights of its edges. A polynominal time algorithm for obtaining the maximum weight matching is presented in [PaSt82].

The set E' indicates the assignments of variables to registers as determined by the maximum matching algorithm in function MATCHING. $E' = \{e_{5,3}, e_{3,5}, e_{4,7}\}$ is indicated in the graph of Figure 8.13(b) with bold lines. After binding the second cluster of variables to the registers according to the matching solution, namely, v_3 to r_5, v_5 to r_3 and v_7 to r_4, the algorithm proceeds to allocate the third cluster of variables, v_8, v_9 and v_{11}, and so on. The final allocation of variables to registers is given in Figure 8.13(c).

The matching algorithm, like the left-edge algorithm, allocates a minimum number of registers. It also takes partially into consideration the impact of register allocation on interconnection allocation since it can associate weights with the edges.

8.6 Iterative Refinement Approach

Given a datapath synthesized by constructive or decomposition methods, its quality can be improved by reallocation. As an example, consider functional-unit reallocation. It is possible to reduce the interconnection cost by just swapping the functional-unit assignments for a pair of operations. For instance, if we start with the datapath shown in Figure 8.8(c), swapping the functional unit assignments for operations o_3 and o_4 will reduce the interconnection cost by four 2-to-1 multiplexers (Figure 8.8(d)).

Changing some variable-to-register assignments can be beneficial too. In Figure 8.8(b), if we move variable g from register r_2 to register r_1 and h from r_3 to r_4, we get an improved datapath with two fewer multiplexers (Figure 8.8(c)).

The main issues in the iterative refinement approach are the types of modifications to be applied to a datapath, the selection of a modification type during an iteration and the termination criteria for the refinement process.

The most straightforward approach could be a simple assignment

exchange. In this approach, the modification to a datapath is limited to a swapping of two assignments (i.e., variable pairs or operation pairs). Assume that only operation swapping is used for the iterative refinement. The pairwise exchange algorithm performs a series of modifications to the datapath in order to decrease the datapath cost. First, all possible swappings of operation assignments scheduled into the same control step are evaluated in terms of the gain in the datapath cost due to a change in the interconnections. Then, the swapping that results in the largest gain is chosen and the datapath is updated to reflect the swapped operations. This process is repeated until no amount of swapping results in a positive gain (i.e., a further reduction in the datapath cost).

Algorithm 8.5 describes the pairwise exchange method. Let $DP_{current}$ represent the current datapath structure and DP_{work} represent a temporary datapath created to evaluate the cost of each operation assignment swap. The function COST(DP) evaluates the cost of the datapath DP. The datapath costs of $DP_{current}$ and DP_{work} are represented by $c_{current}$ and c_{work}. The procedure SWAP(DP, o_i, o_j) exchanges the assignments for operations o_i and o_j of the same type and updates the datapath DP accordingly. In each iteration of the innermost loop, *CurrentGain* represents the reduction in datapath cost due to the swapping of operations in that iteration. *BestGain* keeps track of the largest reduction in the cost attainable by any single swapping of operations evaluated so far in the current iteration.

This approach has two weaknesses. First, using a simple pairwise swapping alone may be inadequate for exploring all possible refinement opportunities. For instance, no amount of variable-swapping can change the datapath of Figure 8.8(b) to that of Figure 8.8(d). Second, the greedy strategy of always going for the most profitable modification is likely to lead the refinement process into a local minimum. While a probabilistic method, such as simulated annealing [KiGV83], can be used to cope with the second problem at the expense of more computation time, the first one requires a more sophisticated solution, such as swapping several assignments at the same time.

Suppose operation o_i has been assigned to functional unit fu_j and one of its input variables has been bound to register r_k. The removal of o_i from fu_j will not eliminate the interconnection from r_k to fu_j unless no other operation that has been previously assigned to fu_j has its input

Algorithm 8.5: Pairwise Exchange.

```
repeat
   BestGain = −∞;
   c_current = COST(DP_current) ;
   for all control steps, s do
      for each o_i, o_j of the same type scheduled into s, i ≠ j do
         DP_work = SWAP(DP_current, o_i, o_j);
         c_work = COST(DP_work);
         CurrentGain = c_current − c_work;
         if CurrentGain > BestGain then
            BestGain = CurrentGain;
            BestOp1 = o_i; BestOp2 = o_j;
         endif
      endfor
   endfor
   if BestGain > 0 then
      DP_current = SWAP(DP_current, BestOp1, BestOp2);
   endif
until BestGain ≤ 0
```

variables assigned to r_k. Clearly, the iterative refinement process has to approach the problem at a coarser level by considering multiple objects simultaneously. We must take into account the relationship between entities of different types. For example, the gain obtained in operation reallocation may be much higher if its input variables are also reallocated simultaneously. The strategy of reallocating a group of different types of entities can be as simple as a greedy constructive algorithm (Section 8.4) or as sophisticated as a branch-and-bound search (e.g., STAR [TsHs90]).

8.7 Summary and Future Directions

In this chapter, we described the datapath allocation problem, which consists of four basic subtasks; unit selection, functional-unit binding, storage binding and interconnection binding. We outlined the features of

some realistic architectures and their impact on the interconnection complexity for allocation. We described the basic techniques for allocation using a simple model that assumes only a straight-line code behavioral description and a simple point-to-point interconnection topology. We discussed the interdependencies among the subtasks that can be performed in an interleaved manner using a greedy constructive approach, or sequentially, using a decomposition approach. We applied three graph-theoretical algorithms to the datapath allocation problem: clique partitioning, left-edge algorithm and weighted bipartite-matching. We also showed how to iteratively refine a datapath by a selective, controlled reallocation process.

The greedy constructive approach (Algorithm 8.1) is the simplest amongst all the approaches. It is both easy to implement and computationally inexpensive. Unfortunately, it is liable to produce inferior designs. The three graph theoretical approaches solve the allocation tasks separately. The clique-partitioning approach (Algorithm 8.2) is applicable to storage, functional and interconnection unit allocation. The left-edge algorithm (Algorithm 8.3) is well suited for storage allocation. The bipartite matching approach (Algorithm 8.4) is applicable to storage and functional unit allocation. Although they all run in polynomial time, only the left-edge and the bipartite matching algorithms guarantee optimal usage of registers, while only the clique-partitioning and the bipartite-matching algorithms are able to take into account the impact on interconnection during storage allocation. The iterative refinement approach (Algorithm 8.5) achieves a high quality design at the expense of excessive computation time. The selection and reallocation of a set of behavioral entities have a significant impact on the algorithm's convergence as well as the datapath cost.

Future work in datapath allocation will need to focus on improving the allocation algorithms in several directions. First, the allocation algorithms can be integrated with the scheduler in order to take advantage of the coupling between scheduling and allocation. As we have pointed out, the number of control steps and the required number of functional units cannot accurately reflect the design quality. A fast allocator can quickly provide the scheduler with more information (e.g., storage sharing between operands of different operations, interconnection cost and data-transfer delays) than just these two numbers. Consequently, the

scheduler will be able to make more accurate decisions. Second, the algorithms must use cost functions based on physical design characteristics. This will give us more realistic cost functions that closely match the actual design. Finally, allocation of more complex datapath structures must be incorporated. For example, the variables and arrays in the behavioral description could be partitioned into memories, a task that is complicated by the fact that memory accesses may take several clock cycles.

8.8 Exercises

1. Using components from a standard datapath library, compare the delay times for the following datapath components:
   (a) a 32-bit 4-to-1 multiplexer,
   (b) a 32-bit 2-to-1 multiplexer,
   (c) a 32-bit ALU performing addition,
   (d) a 32-bit ALU performing a logic OR,
   (e) a 32-bit floating-point multiplier,
   (f) a 32-bit fixed-point multiplier, and
   (g) a 32-bit 16-word register file performing a read.

2. Suppose the result of operation o_i is needed by operation o_j and the two operations have been scheduled into two consecutive control steps. In a shared-bus architecture such as that of Figure 8.6, o_j will be reading its input operands before o_i has written its output operand. What changes are required in the target architecture to prevent o_j from getting the wrong values?

3. Extend the design in Figure 8.7 to support chaining of functional units. Discuss the implications of this change on allocation algorithms.

4. *Extend the left-edge algorithm [KuPa87] to handle inputs with conditional branches. Does it still allocate a minimal number of registers?

5. Prove or disprove that the bipartite-matching method [HCLH90] uses the same number of registers as the left-edge algorithm does in a straight-line code description.

6. Extend the weighted bipartite-matching algorithm to handle inputs with conditional branches. Does it still allocate a minimal number of registers?

7. Show that every multiplexer can be replaced by a bus and vice versa.

8. In a bus-based interconnection unit, assume that there is one level of tri-state buffers from the inputs to the buses and one level of multiplexers from the buses to the outputs. Redesign the interconnection unit of Figure 8.6 (which, by coincidence, uses no multiplexers at all), using the same number of buses so that the data transfer capability of our five-transfer example is preserved and the number of tri-state buffers is minimized (at the expense of more multiplexers, of course).

9. *Some functional units, such as adders, allow their two input operands to be swapped (e.g., $a + b = b + a$). Suppose both the functional-unit and storage allocation have been done. Design an interconnection-allocation algorithm that takes advantage of this property.

10. Using a component library and a timing simulator, measure the impact of bus loading on data-transfer delay. Draw a curve showing the delay as a function of the number of components attached to the bus.

11. Show an example similar to that of Figure 8.8, where swapping of a pair of variables reduces the worst case delay for the data transfer. Assume that latches exist at both the input and output ports of all functional units.

12. Given a partially allocated datapath in which both the functional-unit allocation and the register allocation have been done, design an interconnection-allocation algorithm that minimizes the maximal bus load for a non-shared bus-based architecture.

13. *Design an interconnection-first algorithm for datapath allocation targeted towards a bus-based architecture. That is, assign data transfers to buses before register and functional-unit allocation. Compare this algorithm with the interconnection-last algorithms. Does this approach simplify or complicate the other two tasks?