3

High-Level Synthesis

3.1 Introduction

In this book, the term *synthesis* is used to denote the process of transforming a digital system from a behavioral specification into an implementation structure. Generally speaking, the specification includes some form of abstractions, i.e., some of the design decisions are not bound. The implementation, on the other hand, has to describe in detail the complete design at a given level of abstraction. Thus, synthesis can be seen as a process of creating implementation details which are left out of the specification [Pen87]. For example, a purely behavioral specification of a microprocessor may specify only what should be done in a typical instruction cycle and leaves it to the synthesis procedure to decide whether a centralized bus should be used, which technique should be employed to implement the control function, and how much parallelism should be supported.

Due to the complexity of digital systems, especially those implemented in VLSI technology, the synthesis process is usually divided into several steps. These steps usually include system-level synthesis, high-level synthesis, logic synthesis and physical design.

System-level synthesis deals with the formulation of the basic architecture of the implementation. The input to this synthesis step is a system-level specification which describes the behavior of the entire system in terms of a set of interacting processes. Such a system can be implemented by a set of cooper-

ating processors, such as ASICs, dedicated controllers, FPGAs, and DSP processors. The allocation of the set of physical processors and the mapping of processes in the behavioral specification onto these processors are the most critical design decisions to be made during the system-level synthesis step. An important feature of the system level is that the synthesis techniques and design requirements are highly application dependent. For example, system-level synthesis techniques used in the real-time embedded-controller application area will be very different from those used in DSP systems. This feature results in also the different definitions of what design tasks constitute a system-level synthesis step. In this book, we will address a few fundamental issues related to system-level synthesis which can be considered as the common denominator for a system-level synthesis technique. These issues are system partitioning, hardware/software co-design, and interconnection-structure design. The output of the system-level synthesis step is a set of processes with well-defined interfaces. Each of the processes is specified by a behavioral specification.

High-level synthesis will then translate the behavioral specification of a process into a structural description that is still technology independent. This structural description is usually given in terms of a netlist at the register-transfer level.

System-level synthesis and high-level synthesis form the front-end of a synthesis approach to digital system design, and are together called *system synthesis*. This chapter will present the main issues related to high-level synthesis, while the next chapter will address the system-level synthesis problems. The issue of controller synthesis which is related to both high-level synthesis and system-level synthesis will be presented at the end of this chapter.

After system-level synthesis and high-level synthesis have been performed, logic synthesis and physical design are used to map the structural implementation at register-transfer level into a layout description which is the final implementation. Logic synthesis and physical design form the back-end of the synthesis approach to digital system design.

One important reason for the separation of the synthesis process into front-end (system) synthesis and back-end synthesis can be attributed to the good general property possessed by front-end synthesis and the short-lived nature of the target semiconductor process associated with the back-end synthesis. As front-end synthesis is not bound to a particular technology, it can be used in several different design environments or adopted quickly to new technologies as needed. The back-end, however, has a very short life cycle, because technologies change.

In recent years, there has been a clear trend toward automating the system-synthesis process and there are several reasons for this:

- Shorter design cycle. The use of automation in the synthesis process reduces the design time, and provides better chances for a company to hit the market window for the products. Automation reduces also the cost of the products significantly since in many cases the design cost dominates the product development cost.
- The ability to explore a much larger design space. An efficient synthesis technique can produce several designs from the same specification in a short period of time. This allows the designer to explore different trade-offs between cost, performance, power consumption, testability, etc.
- Support for design verification. One prerequisite for automating the hardware/software co-design process, for example, is to start the synthesis process with a joint specification of both the hardware and software. This makes it possible to verify the complete design consisting of both hardware components and software procedures.
- Fewer errors. The reduction of manual design activities means that the number of human errors will be decreased. If the synthesis algorithms can be validated, we can also be more confident that the final design will correctly implement the given specification.
- Increased availability of IC technology. As more design knowledge is captured in the synthesis algorithms, it is much easier for people who are not IC-technology experts to design chips.

3.1.1 The High-Level Synthesis Tasks

The input to the high-level synthesis process is given in an algorithmic-level specification, such as behavioral VHDL. This type of specification gives the required mapping from sequences of inputs to sequences of outputs. The specification should constrain the internal structure of the system to be designed as little as possible. From the input specification, a synthesis system produces a description of a data path, that is, a network of registers, functional units, multiplexers, and buses. A control part should also be produced if it is not integrated into the data path. In a synchronous design, the control part can be given as microcode, PLA profiles or random logic.

The basic components of the data path will eventually be implemented by some physical modules available in a given technology. The technological parameters, such as silicon area, operation delay and power consumption of the physical modules are usually stored in a module library and made available to the high-level synthesis algorithms. In this way, the same high-level synthesis algorithm can be used to synthesize design based on different technologies by using different module libraries.

The basic tasks to perform in high-level synthesis are behavioral analysis, design-style selection, operation scheduling, data-path allocation, control allocation, module binding, and optimization.

3.1.2 Basic Synthesis Techniques

To carry out a synthesis task means to make design choices. There is usually a set of alternative structures that can be used to realize a given behavior. For example, an addition operation in a given behavioral specification can be implemented either by a dedicated adder or share an ALU with several other operations.

The function of a synthesis algorithm is to analyze all or a subset of these alternatives and to choose the best structure which meets given design constraints, such as limitations on cycle time, area, or power, while minimizing a cost function. The difficulty of synthesis is that trade-offs of different design aspects are highly dependent on each other. Thus, when making design decisions for a synthesis task, other tasks have to be taken into account in order to optimize some design criterion, for example, the implemetation cost. Consequently, each of the synthesis tasks cannot be carried out independently without reducing the possibility of global optimization of the design.

Another difficult factor of synthesis is the immense gap between the input specifications and the implementation results. Many design decisions are not bound in the specification and are left for the synthesis algorithms to decide. For example the synthesizer has to consider whether to multiplex a set of operations onto a single ALU or to implement several dedicated operators and the consequences of such a decision in terms of device timing, power consumption, chip size, pin-out count, and other low-level parameters.

The final source of difficulty of automated system synthesis is the parallelism inherent in a digital system. To organize the available hardware resources to efficiently and reliably perform the desired operation, the synthesis algorithms have to automatically generate parallel structures as well as their synchronization/communication schemes. They are also responsible for the scheduling of operations so as to ensure sufficient parallelism in the implementation.

It is obvious that system synthesis is a very complex problem. Many of the synthesis tasks have also been proved to be NP-complete [Gajo75, DeMi93]. Therefore, it is often necessary to divide a design into several modules and apply synthesis algorithms to one module at a time. To further reduce the complexity, a synthesis approach either partitions the synthesis task into several sub-tasks and perform one sub-task at a time, or partition the task into a sequence of transformation steps each of which makes a small change to the intermediate result of the earlier steps. The later approach is called the transformational approach to synthesis. The basic idea of the transformational approach can be illustrated with Figure 3.1, where the traditional approach to high-level synthesis divides the main synthesis task into three sub-tasks: allocation, scheduling, and binding. The transformational approach first moves the design to a structural implementation in one single step by a relatively naive

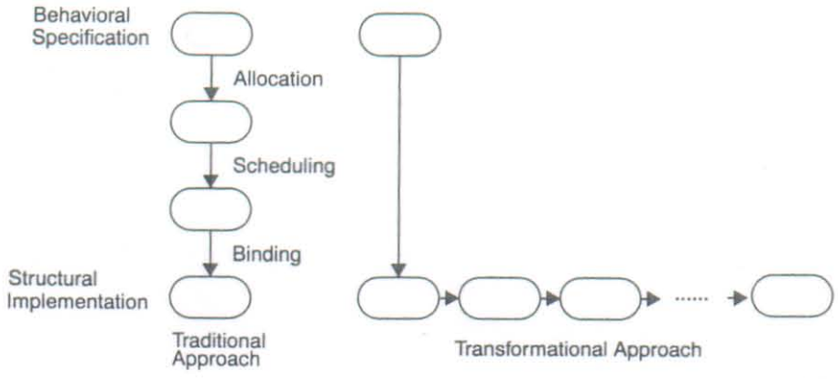


Figure 3.1: Comparison between the traditional and the transformational approaches to high-level synthesis.

mapping. The implementation produced by this step is then transformed using the iterative application of semantic-preserving transformations. These transformations do not change the level of abstraction, rather they explore the implementation space, looking for good solutions.

The use of the transformational approach has two main advantages. The first one is that it facilitates the correctness-by-construction method. Since the synthesis process is divided into a sequence of small transformations, if the transformations can be proved to be semantics-preserving, the synthesis will produce a design which correctly implements the specification [Pen88].

The second advantage of the transformational approach is that it makes it more efficient to employ optimization heuristics in the synthesis process. A transformational approach can be viewed as a neighborhood-search optimization method. The synthesis process starts with an initial implementation which is generated by a naive mapping and similar to an initial solution of an optimization problem. The synthesis process then makes small changes to transform the current solution to a neighborhood solution, which is the same as the neighborhood moves. The objective of the transformations is to reach the optimal design, which is the same as finding the optimal solution in the solution space. Therefore, we can employ existing neighborhood search techniques in the transformational approach. Techniques such as simulated annealing, tabu search, and genetic algorithms have been reported to perform well in different synthesis processes [EPKD97, HaPe96].

An important component of the transformational approach is the design representation which is used to capture the intermediate results of the synthesis process. The representation model must be able to represent the design with different degrees of completion. That is, it should be able to describe a very abstract design with a lot of unspecified information, for example, a purely

behavioral specification. At the same time, it should also be able to describe a very detailed implementation with physical parameters which is, for example, produced by the synthesis process. Thus, it is not necessary or always possible to have just one design representation model; a lot of synthesis systems actually use several different representation models during different stages of the synthesis process.

In most design environments, however, it is very useful to have a unified design representation which can be used to represent the design at different levels of abstraction. The main application of a design representation is to explicitly capture the intermediate result of a design so as to allow the design algorithm to make *appropriate* design decisions. This is very important in the transformational approach to synthesis.

3.2 Scheduling

Operation scheduling, or in short scheduling, deals with the assignment of each operation to a time slot corresponding to a clock cycle or time interval. Typically, the input to this task consists of a control and data flow graph (CDFG), a set of available hardware resources and a performance constraint. A schedule will be generated such that the data/control dependency defined by the CDFG will not be violated and the performance constraint is satisfied.

Since scheduling determines which operations can be assigned to the same time slot, it affects the degree of concurrency of the resulting design and thus its performance. Further, the maximum number of concurrent operations of a given type in a schedule is a lower bound on the number of required hardware resources for that operation. Therefore, the choice of a schedule affects the cost of the implementation and consequently scheduling plays an important role in high-level synthesis.

The scheduling problem can be formulated in several ways depending on the basic assumptions made. One straightforward way is to assume that the behavioral descriptions do not contain conditional or loop constructs, that each operation takes exactly one control step to execute, and that each type of operation can be performed by one and only one type of functional unit [GDWL92]. In this case, we have the following problem:

Given: a set O of operations with a partial ordering which determines the precedence relations, a set K of functional unit types, a type function, τ: $O \rightarrow K$, to map the operations into the functional unit types, and resource constraints m_k for each functional unit type.

Find: a (optimal) schedule for the set of operations that obeys the partial ordering and utilizes only the available functional units.

The above stated problem is called resource-constrained scheduling, since a set of constraints regarding the number of functional units (hardware resources) for each type are explicitly given. For example, in a design where only two

multipliers can fit within the available chip area, it is necessary to impose a resource constraint to limit the number of multipliers to two. The other scheduling problems are time-constrained or time- and resource-constrained, which will be discussed later in this section.

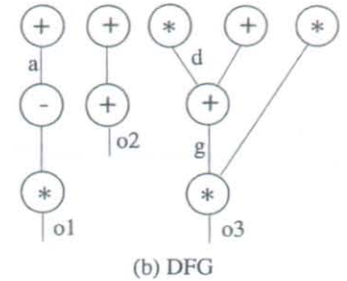
Example 3.1: Figure 3.2 gives a simple behavioral specification and its corresponding data flow graph (DFG). The DFG is generated by an algorithm which analyzes the data dependency relation of the different operations. The data dependency analysis is performed based on the following principle: Let O be the set of all operations in the DFG. If the result of operation $o_i \in O$ is used by operation $o_j \in O$, then operation o_i must finish its execution before operation o_j can begin, and we say that there is a data dependency between these two operations. This data dependency is represented during the synthesis process as a precedence constraint (partial ordering) between the operations, which must be satisfied by the schedule. Figure 3.2b illustrates the data flow graph which is generated from the VHDL code shown in Figure 3.2a. We have assumed that all data flows downwards and have therefore drawn the directed edges only as edges without arrows to indicate directions. □

The simplest scheduling technique is a greedy heuristics based on the "as soon as possible" (ASAP) principle. To schedule the DFG given in Figure 3.2b, using the ASAP algorithm, the operations are first sorted topologically according to their partial ordering; that is, if there is a partial order from o_i to o_j, o_i will be sorted before o_j. The algorithm then schedules operations one by one in the topologically sorted order by placing them in the earliest possible control step [MPC90].

For the DFG example given in Figure 3.2b, the topologically sorted order is illustrated in Figure 3.3a by the number used to label the operations. Let us assume that the available functional units include one adder and one multiplier. Both the addition and substraction operations are mapped into the adder, and

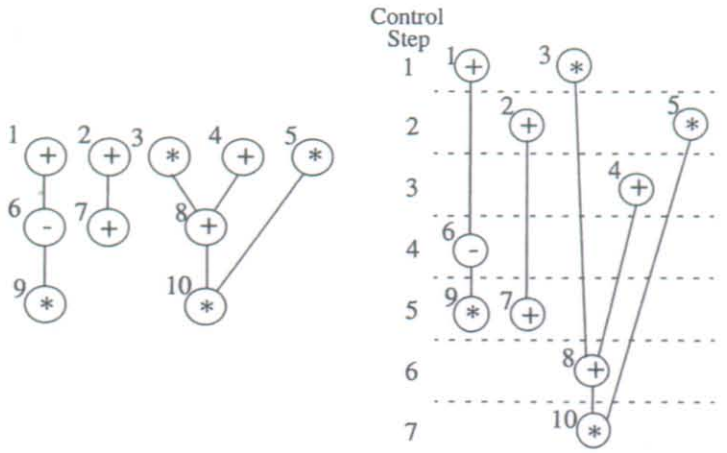
```
a := i1 + i2;
o1 := (a - i3) * 3;
o2 := i4 + i5 + i6;
d := i7 * i8;
g := d + i9 + i10;
o3 := i11 * 7 * g;
```

(a) Behaviroal specification



(b) DFG

Figure 3.2: A behavioral specification and its data flow graph.



(a) Sorted DFG (b) ASAP schedule

Figure 3.3: An ASAP scheduling example.

the multiplication operations are mapped into the multiplier. When operation 1 (the operation labeled with 1) is considered for scheduling, it is scheduled in control step 1, since the addition operation is mapped to the adder and there is an adder available. When operation 2 is considered, however, since the adder is already occupied, it cannot be scheduled in control step 1. Operation 2 will be scheduled in control step 2 instead. The algorithm will then examine operation 3. Since the multiplication operation is mapped to the multiplier and the multiplier is available, it is scheduled in control step 1. This process will continue until each operation is assigned to a control step.

Example 3.2: Figure 3.3b illustrates the result of applying the ASAP scheduling algorithm to the DFG graph of Example 3.1, which is shown in Figure 3.3a. In this case, seven control steps are needed to execute the given behavioral specification, provided that one adder and one multiplier are used.

□

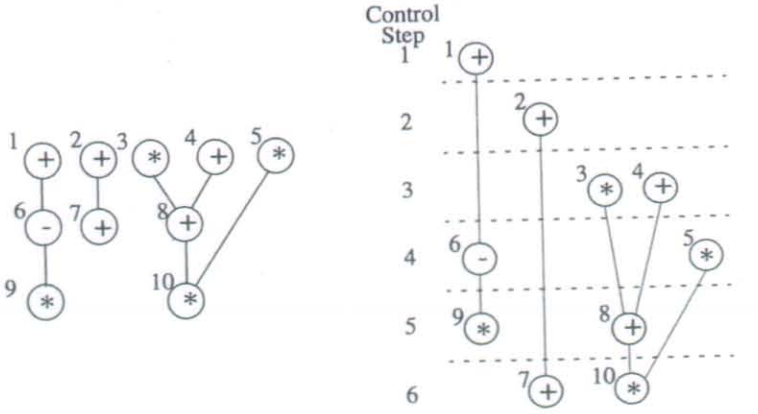
A similar approach to the simple scheduling problem is to use the "as late as possible" (ALAP) principle. With an ALAP algorithm, all the operations are also first sorted topologically according to their data/control dependencies, as in the case of ASAP. However, the operations will be scheduled backwards by placing them in the latest possible control step. To schedule the DFG graph of Example 3.1, operation 10 is considered first. Since there is a multiplier available, operation 10 is scheduled in the last control step, which is illustrated in Figure 3.4. The next operation to be considered is operation 9, another multiplication. It cannot be scheduled in the last control step, since there is no more

multiplier available. Operation 9 will therefore be scheduled in the last but one control step. When operation 8, an addition, is considered, even though there is an adder available in the last control step, it cannot be scheduled there, since there is a data dependence between operation 8 and 10. Operation 8 must be completed before operation 10 can start. Since operation 10 has been scheduled in the last control step. The latest possible control step to perform operation 8 is the last but one control step. The algorithm will then exam operation 7, which has no data dependence with any operation already scheduled. Operation 7 can therefore be scheduled in the latest possible control step where the needed hardware resource to perform it is available, which is the last control step.

Example 3.3: Figure 3.4b illustrates the results of applying the ALAP algorithm to the DFG graph of Example 3.1, which is also depicted in Figure 3.4a. The ALAP algorithm generates a schedule which consists of six control steps, which is one step less than the schedule generated by the ASAP algorithm. Since the two designs need to have the same amount of functional units, the ALAP algorithm generates a better schedule in this particular case.

□

Usually, we would like the generated schedule to be an optimal one, i.e., it takes the minimal number of control steps to execute the specified behavior. The general scheduling problem is however NP-complete [GaJo75], and heuristics that do not guarantee optimal results are widely used to generate satisfactory solutions. In the following sections, a few of the most widely used algorithms will be described. For a complete treatment of scheduling techniques, please refer to [DeMi93].



(a) Sorted DFG (b) ALAP schedule

Figure 3.4: An ALAP scheduling example.

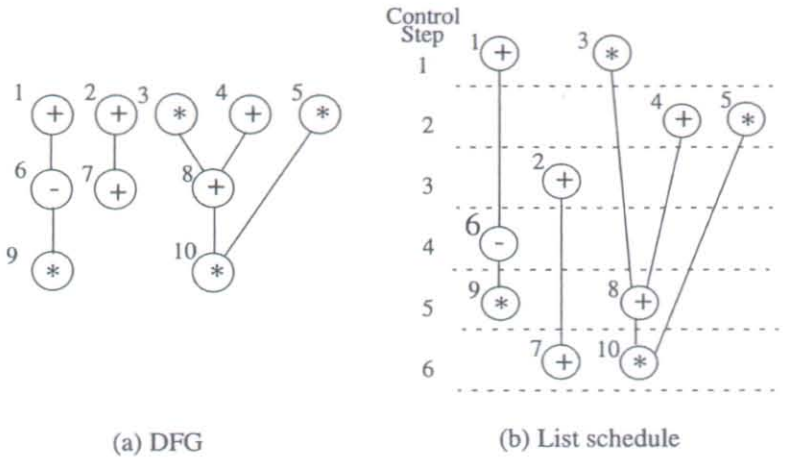
3.2.1 List Scheduling

The ASAP and ALAP algorithms are examples of *constructive* techniques for solving the scheduling problem. In such techniques, a schedule is constructed step by step until all operations are scheduled.

Another constructive approach is *list scheduling*, which proceeds from control step to control step. In each control step, the operations that are available to be scheduled are kept in a list, ordered by some priority function. Each operation on the list is then scheduled if the resources needed are free; otherwise it is deferred to the next control step.

An important decision for a list-scheduling approach is therefore to select the priority function, since it has a strong impact on the final results. Some systems give higher priority to operations with low mobility; here mobility of a operation is defined as the number of control steps from the earliest up to the latest feasible control step in which the operation can be scheduled. Others give higher priority to operations with more immediate successors, arguing that scheduling them in the current control step would make the largest number of operations ready, thereby allowing the earliest possible consideration of each operation. Unfortunately, there is no agreement on which priority function is best, and the selection of such a function usually depends on the application.

Let us consider the DFG of Example 3.1, which is illustrated in Figure 3.5a. Assume that the priority function is defined as the length of the path from the operation to the end of the block; an operation associated with a high number has high priority. In the first step of the list scheduling algorithm, operation 1, 2, 3, 4, and 5 are ready to be scheduled since none of them depend on any other operation to be completed. These ready operations are ordered by the priority



(a) DFG (b) List schedule

Figure 3.5: A list-scheduling example.

function as follows:

Priority(o_1) = 2;
Priority(o_3) = 2;
Priority(o_4) = 2;
Priority(o_2) = 1;
Priority(o_5) = 1.

These operation will be scheduled one by one in the above order. Assume again that there are only one adder and one multiplier available. Operation 1 (o_1) and 3 will be scheduled in the first control step. The algorithm will then proceed to the second control step. Now we have operation 2, 4, 5, and 6 ready to be scheduled, since operation 2, 4, and 5 do not have any predecessors and the only predecessor of operation 6 has been scheduled in the previous step. Based on the priority function, we have the following order:

Priority(o_4) = 2;
Priority(o_2) = 1;
Priority(o_5) = 1;
Priority(o_6) = 1;

Therefore operation 4 and 5 will be scheduled in control step 2 and the algorithm proceed to control step 3. Operation 2, 6, and 8 are now ready to be scheduled and they will be ordered in the following way:

Priority(o_2) = 1;
Priority(o_6) = 1;
Priority(o_8) = 1.

Since all the three ready operations are of type addition and there is only one adder available, only one of them can be scheduled. The three operations have also the same priority value; this provides room for a secondary priority function to be used to select one of them to be scheduled. We assume however that only one priority scheme is used. Therefore, the selection is based on the topological order and operation 2 is scheduled. This process continues until all operations are scheduled. The final schedule for this example is illustrated in Figure 3.5b. This gives a schedule with six control steps, which is the optimal solution for this example.

3.2.2 Force-Directed Scheduling

The scheduling problems discussed so far are called resource-constrained scheduling (RCS). A RCS algorithm tries to find a schedule with the smallest number of control steps under some resource constraints. The resource constraints are typically given in the form of the number of functional units of each type, as in the case of the previously discussed examples. In the more general form, the resource constraints can be given in terms of a complex cost function such as the estimated area of the silicon implementation of the whole

design.

There is another type of scheduling problems which are called time-constrained scheduling (TCS). A TCS algorithm tries to minimize the resources required to meet a specified global time constraint. The time constraint is typically given in terms of the number of control steps allowed for the execution of the specified behavior. The TCS problems occur very often in digital signal processing applications in which the system throughput is fixed and the silicon area must be minimized.

The force-directed scheduling algorithm [PaKn89] is one of the most widely used techniques for the TCS problem. The basic strategy is to place similar operations in different control steps so as to balance the concurrency of the operations assigned to the functional units without increasing the total execution time. By balancing the concurrency of operations, it is ensured that each functional unit has a high utilization and therefore the total number of units required is decreased.

The forced-directed scheduling algorithm consists of three main steps: determine the time frame of each operation, create a distribution graph, and calculate the force associated with each assignment.

The first step of the force-directed scheduling algorithm is to determine the time frame of each operation. A good approximation of the time frame can be determined by constructing the ASAP and ALAP schedules without any resource constraints and then combining the result of both schedules.

Example 3.4: Let us consider the example given in Figure 3.2 again. Assume that the time constraint indicates that four control steps are allowed for this example. Figure 3.6 illustrates the ASAP and ALAP schedule of the DFG given in Figure 3.2b, assuming no resource constraints. From the two schedules, it is easy to identify the time frame

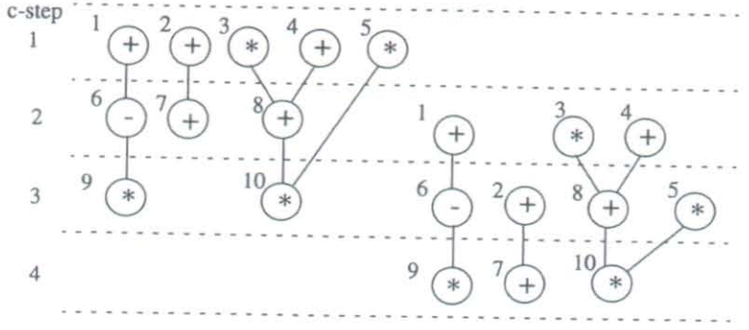


Figure 3.6: ASAP and ALAP schedules of Example 3.1.

for each operation. For example, operation 1 can be either scheduled in control step (c-step) 1 or 2 and therefore its time frame spans from c-step 1 to c-step 2. Operation 2, on the other hand, has a time frame from c-step 1 to c-step 3. The question that remains to be answered is in which c-step of the time frame the operation should be scheduled in order to reduce the number of functional units needed. It can be observed that if the ASAP schedule is directly used, 3 adders and two multipliers are needed, which is definitely not optimal for this example. □

The time frames for all the operations can be collected in a graph, such as the one given in Figure 3.7 for Example 3.1. The width of the box in Figure 3.7 containing an operation in a c-step represents the probability that the operation will be eventually placed in that c-step (time slot). It is assumed that the probability distribution for each operation is uniform. The width of an operation box therefore equals 1 divided by the number of c-steps of its time frame.

The next step of the force-directed scheduling algorithm is to add the probabilities of each type of operations for each c-step and build a *distribution graph* for it. The distribution graph shows, for each c-step, how heavily loaded that step is, provided that all possible schedules are equally likely. If an operation could be done in any of the k steps in a time frame, $1/k$ is added to each of the c-steps in the graph. Using the information captured in Figure 3.7, we can calculate the value of the distribution graph, or DG, for the multiplication operations. The results are $DG_{mult}(1) = 1/2 + 1/3 = 0.833$, $DG_{mult}(2) = 1/2 + 1/3 = 0.833$, $DG_{mult}(3) = 1/2 + 1/2 + 1/3 = 1.333$, and $DG_{mult}(4) = 1/2 + 1/2 = 1$, as illustrated in Figure 3.8. The results for the addition/subtraction operation DG are $DG_{a/s}(1) = 1/2 + 1/3 + 1/2 = 1.333$, $DG_{a/s}(2) = 1/2 + 1/2 + 1/3 + 1/3 + 1/2 + 1/2 = 2.667$, $DG_{a/s}(3) = 1/2 + 1/3 + 1/3 + 1/2 = 1.667$, and $DG_{a/s}(4) = 1/3 = 0.333$..

The third step of the force-directed scheduling algorithm is to calculate the force associated with every feasible c-step assignment of each operation. For an operation with a time frame that spans from c-steps f to t, the force associated

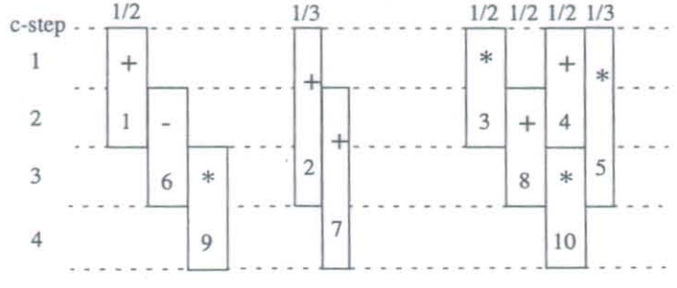


Figure 3.7: Time frames for Example 3.1.

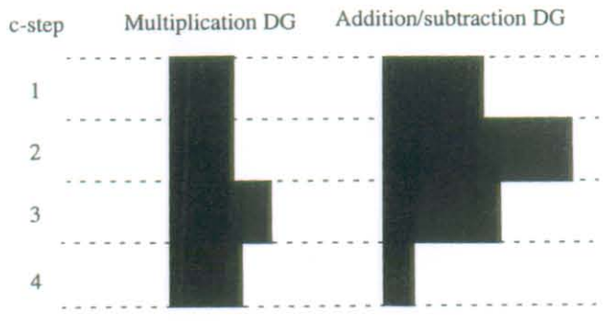


Figure 3.8: Distribution graphs for Example 3.1.

with its assignment to c-step j ($f \le j \le t$) is

$$Force(j) = DG(j) - \sum_{i=f}^{t} \left[\frac{DG(i)}{(t-f+1)} \right]$$

In other words, the force associated with the tentative assignment of an operation to c-step j is equal to the difference between the distribution value in that c-step and the average of the distribution values for the c-steps bounded by the operation's time frame.

For example, with the assignment of operation 10 to c-step 3, we have

$Force(3) = DG_{mult}(3)$ - average DG_{mult} value over time frame of operation 10
$= 1.333 - (1.333 + 1)/2 = 0.167.$

On the other hand, the assignment of operation 10 to c-step 4 yields

$Force(4) = DG_{mult}(4)$ - average DG_{mult} value over time frame of operation 10
$= 1 - (1.333 + 1)/2 = -0.167.$

As the DG shows, if operation 10 is assigned to c-step 3, the distribution is not very well balanced, while the assignment of operation 10 to c-step 4 will generate a better result. This is reflected by the negative value of the force associated with the latter assignment.

We must also calculate the force for all predecessors and successors of the current operation whenever their time frames are affected. These additional forces are called indirect forces. The total force is the sum of the direct and indirect forces [PaKn89].

In Example 3.1, operation 10 has four predecessors, operations 3, 4, 5, and 8. The assignment of operation 10 to c-step 4 will not affect the time frame of any of its predecessors. Therefore the total force of assigning operation 10 to c-step 4 equals the direct force calculated above, namely -0.167. The assignment of operation 10 to c-step 3, on the other hand, will affect the time frames of operations 3, 4, 5, and 8. For example, operation 8 will now only be able to be performed in c-step 2 instead of both c-steps 2 and 3. Therefore, the assignment of operation 10 to c-step 3 implies that operation 8 will be assigned to c-step 2 and the indirect force of the latter assignment must be calculated and added to the total force of the former assignment. The indirect force of assigning operation 8 to c-step 2 equals $DG_{a/s}(2)$ - average $DG_{a/s}$ value over the time frame of operation 8 (c-steps 2 and 3), i.e., 2.667 - (2.667 + 1.667)/2, which is 0.5. The same calculation should be performed for operations 3, 4, and 5. All the indirect forces will then be added to the direct force to yield the total force of assigmning operation 10 to c-step 3.

Once all the forces are calculated, the operation-control step pair with the largest negative force (or least positive force) is scheduled. The distribution graphs and forces are then updated and the above process is repeated until all operations are scheduled.

Since the force-directed scheduling algorithm schedules one operation in each iteration, it is also constructive. Different from other constructive approaches, however, force-directed scheduling makes global analysis of the operations and control steps when selecting the next operation to be scheduled. Therefore force-directed scheduling is more expensive computationally than, for example, list scheduling. Force-directed scheduling has complexity $O(cN^2)$, while list scheduling $O(cN \log N)$, where c is the number of control steps and N the number of operations [MPC90].

3.2.3 Transformation-Based Scheduling

The scheduling techniques discussed up till now are all of constructive type. Another basic class of scheduling algorithms is based on transformations. A transformation-based algorithm begins with an initial schedule, usually either maximally serial or maximally parallel, and applies transformations to it to obtain other schedules. The basic transformations are converting serial operations, or blocks of operations, into parallel ones and the inverse, converting parallel operations into series ones. Transformation-based algorithms differ in how they choose what transformations to apply and in which order these are applied.

One extreme technique is to use exhaustive search. That is, all possible combinations of serial and parallel conversions are tried and the best design will be chosen. This method has the advantage that it looks through all possible designs and guarantee the optimal solution. However, it is computationally very expensive and not practical for large designs. Exhaustive search can be improved, to reduce the computation time needed, somewhat by using branch-and-bound techniques, which cut off the search along any path that can be recognized to be suboptimal.

Another approach to transformation-based scheduling is to use heuristics to

guide the process. Transformations are chosen that promise to move the design closer to the optimal design. Both the CAMAD system [Pen86, PKL89, PeKu94] and the Yorktown Silicon Compiler [Cam90] use this approach.

One important advantage of the transformation-based approach is that in each iteration, a complete schedule exists and accurate estimation of the design in terms of different criteria can be made. It is therefore straightforward to extend this type of algorithms to handle many advanced issues related to scheduling to be discussed in the next section. Further combination of scheduling and allocation can also be achieved using the transformation-based approach, as in the case of the CAMAD system, which is discussed in Chapter 6.

3.2.4 Advanced Scheduling Topics

The scheduling problems we have discussed so far are simplified versions of the real problems. We will now discuss several advanced topics which must be considered by any practical scheduling algorithm.

Control constructs

Until now we have assumed that a CDFG corresponds to a single basic block, i.e., one section of straight-line code with only one entry and one exit point. However, most hardware description languages, such as VHDL, support conditionals, loops and other control structures. The scheduling algorithm must consider these constructs during the scheduling process.

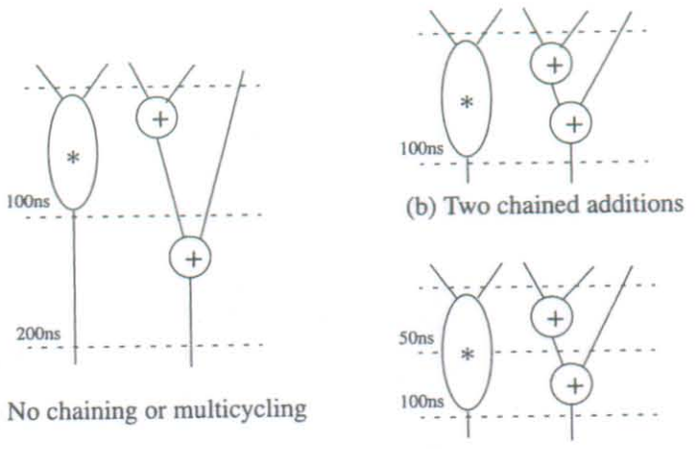
When scheduling conditional branches, the scheduling algorithm should make use of the possibility to share functional units between mutually exclusive branches as much as possible. For example, the same adder can be used in both the "then" and "else" clauses of an "if" statement [WaCh95].

Chaining and multicycling

We have up till now assumed that all operations requires the same amount of time to execute, and this time is the control-step length or clock cycle time. In practice, different operation types may have different execution times and the above assumption results in the situation that the clock cycle time is dictated by the most time consuming operation.

Example 3.5: Figure 3.9a illustrates a design where the addition and multiplication operations are mapped into an adder and a multiplier with 50ns and 100ns delay, respectively [WaCh95]. With the single operation per cycle assumption, the clock cycle time is determined to be 100ns and the overall schedule length is 200ns. □

In order to avoid the above problem, chaining and multicycling techniques



(a) No chaining or multicycling

(b) Two chained additions

(c) A multicycle multiplication

Figure 3.9: Chained and multicycle operations.

can be used.

Chaining is the task of combining more than one operation in a control step. Figure 3.9 illustrates the case when chaining is used in Example 3.5, where the two addition operations are chained and scheduled in the same control step. In this way, the overall schedule length becomes 100ns. However, an extra adder is needed in this example since the two addition operations can not longer be mapped into the same adder because they are performed in the same control step.

Chaining can be performed before scheduling, and then the combined operations can be considered as a single entity in the scheduling process. Chaining can also be performed together with scheduling.

Another alternative to improve the schedule of Example 3.5 is to set the clock period to 50ns, and to execute the multiplication over two control steps, as illustrated in Figure 3.9c. An operation which is to be executed continuously over several clock cycles is called a *multicycle* operation. In Example 3.5, multicycling decreases the schedule length to 100ns, just like chaining, but without the cost of another adder. However, multicycling uses twice as many control steps as chaining, which may result in a larger controller. Further, multicycling usually needs additional registers to latch the results from one cycle to the other. For example, the result produced by the first addition operation must be latched while this is not the case in the chaining solution.

Scheduling with timing constraints

Most scheduling techniques try to find a schedule which has the shortest schedule length while satisfying a set of constraints or one which requires the

least resources and has a shorter schedule length than a given constraint. In both cases the global performance of the design, in terms of the overall schedule length, is used as one design criterion. However, few digital systems work in isolation, so there may also be a need to specify more detailed timing constraints on certain operations, or sets of operations. For example, we might want to give a minimum timing constraint, which specifies that one operation must be executed less than a specified amount of time after another operation, or a maximum timing constraint, which specifies that one operation must be executed at least a specified amount of time after another operation.

Most scheduling techniques handle these timing constraints by adding additional constraint edges to the CDFG, and then treating those additional edges in much the same way as other constraints. Special techniques based on heuristics which deal with local timing constraints in a systematic manner have also been developed [HaPe96].

Discussion

In the general sense, scheduling is to assign operations to control steps so as to minimize a given objective function while meeting a set of design constraints. The object function may include the number of control steps, delay, power, hardware resource, and testability. The general scheduling problem can be formulated as an Integer Linear Programming (ILP) problem. Using the ILP formulation, for example, the time-constrained scheduling problem with a minimal total silicon area can be easily formulated. We need only to let the cost function be the estimated total area of the final implementation. The concept of ILP will be further elaborated later in this chapter when data path allocation and binding are discussed as well as in Chapter 5.

3.3 Data Path Allocation and Binding

In general, data path allocation and binding deal with the problem of which resources are used to realize in the physical implementation. Such resources include registers, memory units and different functional units as well as their communication channels. The basic principle is to share resources as much as possible provided that the performance and other design criteria can be satisfied.

Allocation and binding carry out selection and assignment of hardware resources for a given design. Allocation determines the type and number of hardware resources for a given design. Binding assigns the instance of an allocated hardware resource to a given data path node. Different data path operations can share the same hardware resource if they are not executed at the same time. For example, an adder can be shared by two additions if they are not

executed during the same clock cycle. A register can also be used to store the values of two variables if the life times of these variables do not overlap.

As pointed out earlier, the term allocation is sometimes used to denote both the allocation and binding tasks.

Example 3.6: An allocation for the example depicted in Figure 3.2 selects two adders and one multiplier. A possible schedule for this allocation is depicted in Figure 3.10. Note that the scheduling assumes, in this case, that at most two adders and one multiplier in every clock cycle can be used, as determined by the allocation. The binding step assigns every operation of the scheduled graph to a physical hardware component. A possible binding for this example is also depicted in Figure 3.10. Addition/substraction nodes 1, 2, 6 and 7 are assigned to adder #1, addition nodes 4 and 8 are assigned to adder #2, and multiplication nodes 3, 5, 9 and 10 are assigned to the multiplier. □

The selection of the type and number of hardware resources during the allocation step is usually formulated as an optimization problem. The main goal is to find the minimum number of resources while fulfilling given area/performance constraints.

The basic assumption made by many high-level synthesis systems regarding binding is that each data path node has at least one module in a module library which implements the function of the data path node. For example, a node which performs an addition operation may correspond to an adder or an ALU in the module library. The different modules can have different areas and/or latencies. This gives the possibility to make trade-offs between different implementations.

The binding problem is also an optimization problem and can be formulated using existing optimization methods. For example, an Integer Linear Programming method or a graph clustering technique can be used to solve it. It can also be solved using heuristic methods.

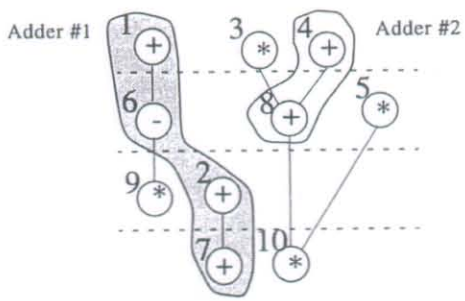


Figure 3.10: Binding for Example 3.6 with 2 adders and 1 multiplier.

3.3.1 Integer Linear Programming

Integer linear programming (ILP) is a subclass of the linear programming problems where the decision variables are of integer values. If we assume that decision variables are represented by a vector X the ILP problem can be formally stated as:

Maximize (or Minimize)	$C^T X$	(3.1)
Subject to	$AX = B$	(3.2)
	$X \geq 0$	(3.3)
	X integer	(3.4)

In this formulation, the maximization/minimization criterion is defined as (3.1) and the constraints for the optimization problem are defined as equations (3.2) and inequalities (3.3). In addition all decision variables are constrained to be integers (3.4). If binary decision variables are used instead of integer ones, the problem is called 0/1 linear programming problem. Once our problem is defined as an ILP problem known methods for solving it can be directly applied.

As an example, we will define the problem of finding a binding, for a given schedule, as a 0/1 linear programming problem. We will concentrate on the formulation of binding constraints. The minimization criterion will not be included in this discussion. The reader can imagine different types of cost functions which can be used to minimize different aspects of the design, for example, interconnection cost or power consumption. For the purpose of this presentation we also make the simplifying assumption that every operation is executed in exactly one clock cycle, i.e., no chaining or multicycle operations are possible.

We define the following decision variables and constants:

- $binding_{ij}$ where $i = 1, 2, \ldots, ops$ and denotes operation number, and $j = 1, 2, \ldots, r$ and denotes component number; the decision variable $binding_{ij}$ is 1 iff operation i is bound to hardware resource j.
- $schedule_{ik}$ where $i = 1, 2, \ldots, ops$ and denotes operation number, and $k = 1, 2, \ldots, max\_step$ and denotes the number of the steps in a given schedule. Since we assume that the schedule is already decided, the constant $schedule_{ik}$ is 1 iff operation number i is scheduled in step number k.

The binding problem can be defined as a solution satisfying the following constraints:

$$\sum_{j=1}^{r} binding_{ij} = 1, \qquad i = 1, 2, \ldots, ops; \tag{3.5}$$

$$\sum_{i=1}^{ops} binding_{ij} \cdot schedule_{ik} \leq 1, \qquad j = 1, 2, \ldots, r;\ k = 1, \ldots, max\_step; \tag{3.6}$$

$$binding_{ij} \in \{0, 1\}, \quad i = 1, 2, \ldots, ops;\ j = 1, 2, \ldots, r. \tag{3.7}$$

The constraint (3.5) denotes that an operation can only be assigned to one resource while the constraint (3.6) requires that at most one operation can be executed on a hardware recourse during an execution step.

The above stated constraints can usually be satisfied by several solutions. Since we are looking for a solution which minimizes a given design criterion, a cost function should be defined to guide the selection of the best solution.

Example 3.7: Let us consider the problem of finding a binding for the example given in Figure 3.2 with the schedule presented in Figure 3.10. For this example, the binding of the two adders has to fulfill the following constraints:

$$binding_{i1} + binding_{i2} = 1, \text{ for } i = 1, 2, 4, 6, 7, 8;$$

$$\sum_{i \in \{1,2,4,6,7,8\}} binding_{i1} \cdot schedule_{ik} \leq 1, \text{for } k = 1, 2, 3, 4;$$

$$\sum_{i \in \{1,2,4,6,7,8\}} binding_{i2} \cdot schedule_{ik} \leq 1, \text{ for } k = 1, 2, 3, 4.$$

There are 16 different solutions which satisfy these constraints. One possible solution is:

$binding_{11} = 1$	$binding_{12} = 0$
$binding_{21} = 1$	$binding_{22} = 0$
$binding_{41} = 0$	$binding_{42} = 1$
$binding_{61} = 1$	$binding_{62} = 0$
$binding_{71} = 1$	$binding_{72} = 0$
$binding_{81} = 0$	$binding_{82} = 1$

The above binding solution is depicted in Figure 3.10. □

The ILP-based definition of the binding problem can be extended in several ways to include more complex and realistic requirements. It can also be defined to include both scheduling and binding.

3.3.2 Clique Partitioning and Graph Colouring

Allocation and binding can also be defined as graph problems. They can be formulated either as the problem of finding cliques in a compatibility graph or that of coloring vertices in a conflict graph.

A *compatibility graph* is used to represent information on resource sharing.

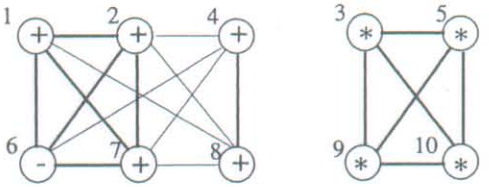


Figure 3.11: A resource-compatibility graph example.

Two operations are compatible, and can share the same hardware resource, if they are of the same type and are not executed at the same time. A compatibility graph $G_{comp}(V, E)$ is built of vertices V denoting operations and edges E denoting the compatibility relation between the operations. A vertex v_i is connected to a vertex v_j if the two operations they represent can be assigned to the same resource. An example of a compatibility graph is depicted in Figure 3.11.

To solve the binding problem using compatibility graphs, we have to find a maximal set of compatible operations. This can be formulated as a maximal *clique partitioning* problem. A clique is defined as a subgraph where all nodes are connected to each other. It is maximal if it is not contained in any other clique.

Example 3.8: Let us consider the example depicted in Figure 3.10. Based on the given schedule and the assumption that addition and substraction operations can share the same resource, a compatibility graph can be built as shown in Figure 3.11. Three maximal cliques can be identified as depicted by bold edges in Figure 3.11. Two cliques {1, 2, 6, 7} and {4, 8} represent the binding of the addition/substraction operations to two adders while the clique {3, 5, 9, 10} represents the binding of the multiplication operations to one multiplier. Note that the solution is not unique and other solutions are also possible. For example, we can also identify other two cliques, {2, 4, 7, 8} and {1, 6}, for the addition/substraction operations. □

A conflict graph, on the other hand, captures the opposite information as the compatible groups. It denotes explicitly the operations that cannot share the same resource. A conflict graph $G_{conflict}(V, E)$ is built of vertices V denoting operations and edges E denoting the conflict relation between them. A vertex v_i is connected to a vertex v_j if the two operations they represent cannot be assigned to the same resource. An example of a conflict graph is depicted in Figure 3.12.

The binding problem in this case is solved using a *graph coloring* algorithm. The algorithm assigns different colors to vertices connected by edges while minimizing the number of colors.

Example 3.9: Let us consider the previous example. The resource-conflict graph, depicted in Figure 3.12, has only two edges representing two resource conflicts for addition/substraction operations. Addition operations 1 and 4 are executed in parallel during the first control step and addition/substraction operations 6 and 8 are executed in parallel in control step 2. In this case, the subgraph for addition/substraction needs be colored using two colors while the multiplication subgraph requires only one color. A coloring scheme is captured in Figure 3.12, which again is not unique. □

Both the maximal clique-partitioning problem and the minimal graph-coloring problem are intractable for realistic-size examples. Thus heuristics, which generate solutions quickly without guaranteeing optimality, have widely been used. We can also make use of the fact that these two problems form each other's dual. The two graphs are complementary to each other and the complexity of one formulation could be much less than the other. This makes it possible to select the formulation with a less complex graph for a given binding problem in order to speed up the binding process.

3.3.3 Left-Edge Algorithm

To show other approaches to allocation and binding we discuss the left-edge algorithm which is used very often for register allocation and binding.

The left-edge algorithm was originally introduced to perform the channel routing task. It was used for assigning interconnections (trunks) into a number of tracks in a channel. In the original formulation of the algorithm, the channel was oriented horizontally and the algorithm sorted trunks in an increasing order of their left end-points. The algorithm assigned trunks into successive tracks starting from the first one. For every track it scanned the ordered list of unplaced trunks and placed them one by one into successive available parts of the track. We will use this algorithm for the assignment of variables into registers.

The left-edge algorithm requires information about the life-time of variables. The life-time of a variable is the time interval when the variable is used by the computation. It can be represented as a bar drawn in parallel to our scheduled

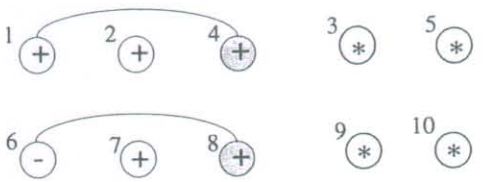


Figure 3.12: A conflict-graph example.

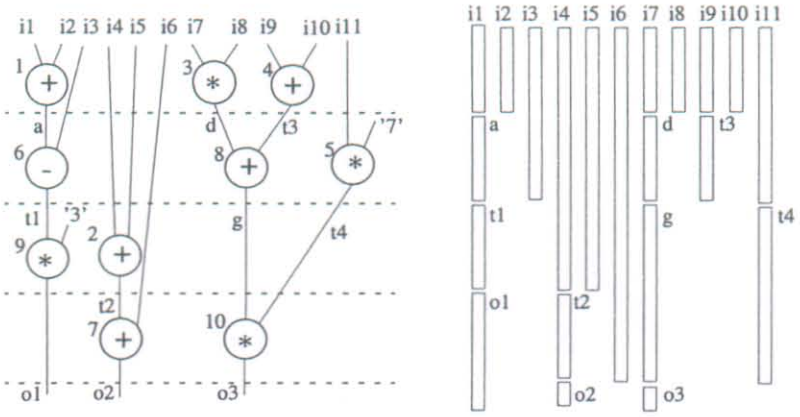


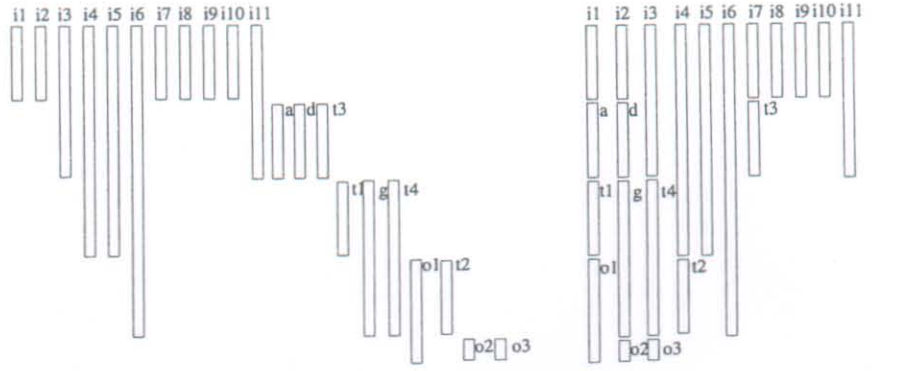
Figure 3.13: Variable life-times

design. The starting point of the bar represents the time when the variable is set and the end of the bar represents the time when the variable is released and its value is not used any longer. In other words, the bar represents define-use time for the variable. On the right hand side of Figure 3.13 there are bars representing the life-times of the 21 variables used in the example given on the left hand side.

A possible solution to the register-assignment problem can use the original formulation of the left-edge algorithm. In this solution all bars representing the life-times are sorted in increasing order of their starting points. The algorithm proceeds then with one register at a time. It assigns the first variable from the list, represented by a bar, into the first register. It then scans the sorted bars and the first encountered unbound bar which has start time higher than the end-time of the already assigned bar is placed into the same register. It continues this process by assigning the next variables into the same register. If the life-time of this register reaches the last scheduled step the algorithm starts to assign variables into the next register using the same method.

Example 3.10: Consider the assignment of the 21 variables used in the example depicted in Figure 3.13. The first step sorts all bars and the resulting sorted list of variables is depicted in Figure 3.14a. The final assignment of variables into registers is depicted in Figure 3.14b. □

The left-edge algorithm will allocate the minimum number of registers, but has two disadvantages. First, not all life-time tables might be interpreted as intersecting intervals on a line. For example, the existence of conditional branches prohibits the interpretation of intersecting intervals on a line, since values occuring in mutural exclusiove branches may share a register although they seem to overlap in life-time [MLD92]. Second, the allocation produced is



(a) The sorted list of variables (b) Assignment of variables into registers

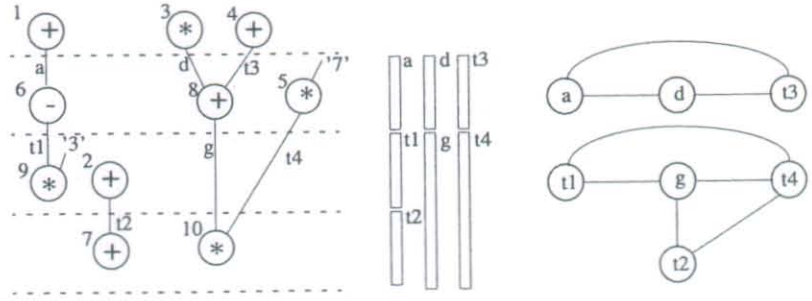
Figure 3.14: Applying the left-edge algorithm for register allocation

neither unique nor necessarily optimal in terms of, for example, the number of multiplexors required.

Example 3.11: Consider the sequencing graph depicted in Figure 3.15a. Figure 3.15b depicts its variable life-times while in Figure 3.15c the conflict graph for all variables is shown. This graph can be colored with three colors which gives an assignment of three registers. □

3.4 Controller Synthesis

As stated earlier, system synthesis deals with systems usually specified as a set of communicating concurrent processes. In this case the controller synthesis is



(a) Sequencing graph (b) Variable life-time (c) Conflict graph

Figure 3.15: Variable life-times and conflict graph.

highly dependent not only on the functionality of the controllers but also on the interaction and synchronization requirements. Traditional high-level synthesis methods, which are limited to synthesis of one concurrent process at a time, produce poor results or cannot deal with such designs. One important system synthesis task is thus to propose an efficient control engine which may include several cooperating controllers.

An efficient method to implement a controller is to use a finite state machine (FSM) notation. The FSM formalism makes it possible to specify a number of states together with transitions between them. An FSM can be defined using either Mealy or Moore machine style. Formally, an FSM can be defined by the following 5-tuple:

$<S, I, O, \delta, \lambda>$, where

S is a set of states;

I is a set of inputs (conditions);

O is a set of outputs (control signals);

δ is a next-state function, $\delta: S \times I \rightarrow S$; and

λ is an output function, $\lambda: S \times I \rightarrow O$ for Mealy machine or $\lambda: S \rightarrow O$ for Moore machine.

In this book, we assume that a controller is finally implemented as one or several FSMs.

The above formulation defines an FSM as a machine which has a number of states. The machine can go from one state to another by executing the next-state function. The next state is determined based on the current state and the input signals. The output signals are determined by the output function. For Mealy machines the output signals are generated based on the current sate and the input signals while for Moore machines they depend only on the current state.

A controller is usually graphically represented by a state diagram. The state diagram is a directed graph with nodes denoting states and arcs denoting transitions from one state to another. The control signals can be assigned either to states or arcs depending on the machine style while guarding conditions are assigned to arcs. A transition from a given state to the next state takes place if and only if the controller is in the given state and the guard assigned to the transition arc connecting both states is true.

In our formulation the controller, consisting of one or several FSMs, is used to control data path activities. The output signals are used to control data path operations while the input signals are conditions generated from the data path to influence the execution of the controller. Figure 3.16 represents schematically the general view of the controller and its relation to the data path.

After generating the FSM specification it is a logic synthesis task to perform further optimization steps, such as state minimization and state encoding, using

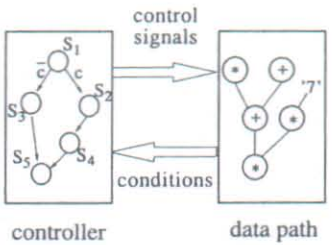


Figure 3.16: A controller/data path architecture.

standard FSM synthesis methods [DeMi94a]. However, some decisions regarding the selection of the control structure have to be made during system synthesis. At this stage it should be decided if a single controller will be used to control the whole design or rather several cooperating controllers will be used. It can also be decided if a hierarchical controller can be used. The selection of controller style depends on the synthesis problem formulation and design criteria, such as performance, area and testability. These decisions deal mainly with the overall architecture of the controller rather than its detailed implementation.

3.4.1 Controller-Style Selection

In this section, we discuss several design styles which can be used to implement complex controllers. The main idea of the presented approaches is to implement a complex control structure by several smaller controllers in order to simplify the design of the generated FSMs.

Single Controller

In this style, the controller is modeled by a single FSM, which is the simplest solution to the controller-style selection problem. Since an FSM is a sequential machine parallelism is only allowed for operations assigned to the same state. Otherwise operations have to be statically scheduled and assigned to consecutive states. The method is very efficient for simple computations but in the case of several parallel execution threads it can lead to the state explosion problem, especially when parallel loops are involved.

Hierarchical Controller

A *hierarchical controller* consists of a number of simple controllers organized in a hierarchy. It is assumed that the hierarchy of controllers is ordered by their parent-child relation. The parent controller distributes an execution task among

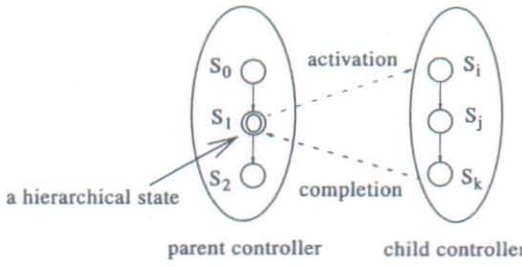


Figure 3.17: An example of a hierarchical controller.

a number of child controllers. Every child controller starts its execution upon receiving the activation signal and it sends a completion signal upon execution termination. The parent controller continues its execution when the child controller sends a completion signal. The child controller can be a parent controller for another lower-level child controller.

Hierarchical controllers usually make use of two special internal control signals, called *activation* and *completion*. The activation signal is used to start another controller and the completion signal provides information about termination of the execution of a controller. They are only used for controller synchronization.

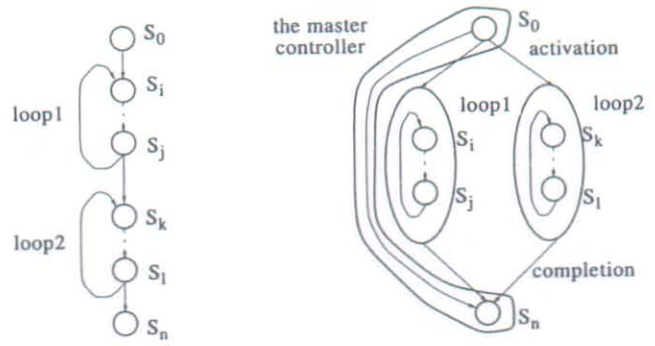
Example 3.12: Figure 3.17 presents an example of a hierarchical controller. In this example, state S_1 is hierarchical and is implemented by a child controller consisting of three states S_i, S_j and S_k. It is activated by the parent controller, with an activation signal. The parant controller will continue its execution upon receiving the completion signal from the child controller.

□

Hierarchical controllers can be used to implement in a natural way several control structures which are commonly used in algorithmic languages, such as procedure and loop constructs. Procedures are well suited for this because their purpose is to create a hierarchy of subprograms. A process calls a procedure and continues upon the procedure return. It matches the hierarchical controller principles very well. A large program can also be restructured by partitioning it into loop-free parts by structuring loops as subprograms. These structures can later use the hierarchical controller concept to implement efficient control engines [DeMi94a].

Parallel Controller

The function of a controller can also be distributed into several smaller controllers which are executed in parallel. These parallel controllers can



(a) The original specification (b) Use of parallel controllers

Figure 3.18: Using parallel controllers for two loops.

communicate to exchange data or synchronize their execution. Generally speaking, decomposition of a controller into a number of parallel controllers reduces the overall controller complexity and solves some design problems. However, It requires new design methods.

In some cases, several parallel controllers do not need to synchronize. They are executed independently of each other. The activation/completion signals can then be used for correct sequencing of operations between a master controller and these parallel controllers, which is illustrated in the following example.

Example 3.13: A design specification includes two consecutive loops as depicted in Figure 3.18a. The computations of the loops are data independent and can be executed in parallel to speed up the computation. The generation of one sequential controller for the parallel computations is impossible in practice since the generated controller should include all combination of states in the two loops which results in a combinational explosion of states. It is possible, however, to generate one master controller which activates two independent parallel controllers, one for each loop, as illustrated in Figure 3.18b. This solves the performance problem while keeping the controller size small.

□

Conditions are used to specify synchronization between parallel controllers, when needed. The controller which is to be synchronized has a conditional state transition. This transition from one state to another will take place only in the case when the condition assigned to the edge connecting these two states is TRUE. The condition can be set by another controller. Using conditions different communication protocols between two or more parallel controllers can be implemented. Using this strategy we can specify hierarchical controllers

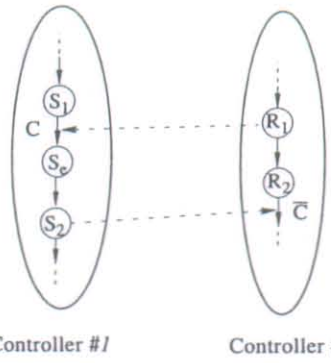


Figure 3.19: Two parallel controllers communicating using condition C.

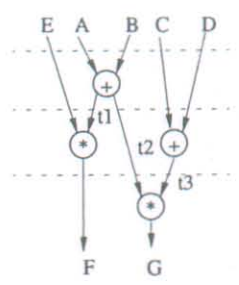
with very complex synchronization schemes.

Communication between parallel controllers can be achieved using a special protocol implemented using the basic technique described above together with data exchange through shared data path objects.

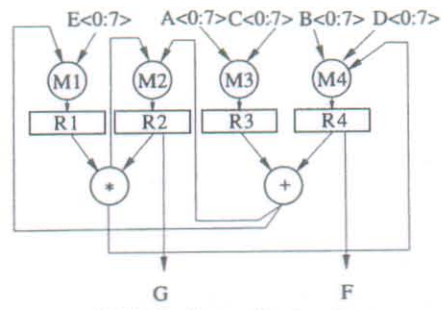
Example 3.14: Let us consider two parallel controllers which communicate as depicted in Figure 3.19. They use a handshake protocol to synchronize their execution and exchange data. Controller #1 waits in state S_1 for condition C to become TRUE. The condition is set by controller *#2* in state R_1. In state S_e controller *#1* can fetch data set by controller *#2*. Then controller *#1* resets condition C and controller *#2* continues its execution. □

Many current design methods offer efficient algorithms for the synthesis of synchronous FSMs. In many cases, however, this limits possible design space exploration and results in over-synchronized controllers. It should be noted that a proper combination of asynchronous and synchronous design styles can solve some of these problems. For example, basic controllers can be implemented as synchronous machines controlled by one clock or several independent clocks while their communication can follow an asynchronous protocol. This solution gives the freedom of using asynchronous design rules for controller synchronization and communication while using well-known synchronous design methods for single controllers.

It should be noted that this solution with ctivation and completion signals is the only possible one for the class of designs which have operations with unbounded-delays. We cannot schedule these operations to any particular clock cycle or a number of clocks cycles since the execution time can not be determined beforehand (see, for example, [DeMi94a]).



(a) Scheduled data-flow description



(b) RTL data path structure

S0: M3=0, Load R3, M4=0, Load R4 **next** S1;
S1: Add, M2=1, Load R2, M1=1, Load R1, M3=1, Load R3, M4=1, Load R4 **next** S2;
S2: Add, M1=0, Load R1, Mul, M4=2, Load R4 **next** S3;
S3: Mul, M2=0, Load R2 **next**...

(c) Control description (FSM)

Figure 3.20: A design example with the generated FSM.

3.4.2 Controller Generation

When the controller style has been selected and the number of FSMs is decided, we need to generate these FSMs. The FSMs can be represented, for example, as state diagrams or symbolic FSMs and later implemented using standard FSM synthesis methods.

The controller generation task has to decide whether the Moore or the Mealy machine should be used. This decision is mainly dependent on the design representation used and the available back-end logic synthesis tools. However, every Mealy machine can be converted into an equivalent Moore machine and vice versa. In this book, we will assume that the Moore machine is used.

Example 3.15: Consider the scheduled data-flow graph given in Figure 3.20a which assumes the use of one adder and one multiplier. The schedule for this graph consists of three control steps. A data path implementation after the binding of functional units, registers and multiplexors, is depicted in Figure 3.20b. Variables E and t3 are assigned to register R1, variables G, t1 and t2 to register R2, variables A and C to register R3, and variables B, D and F to register R4. Modules M1, M2, M3 and M4 represent multiplexors. The controller is given in Figure

3.20c in the form of a symbolic FSM instead of a state diagram. It has four states. The first state, S0, loads registers R3 and R4 with the values of variables A and B respectively. Control signals for multiplexors and registers are generated in this state. The signals M3=0 and M4=0 open the first input of the respective multiplexors while control signals Load R3 and Load R4 trigger the register loading. The next states perform the computations specified in the data-flow graph.

The VHDL description depicted on Figure 3.21 gives a possible synthesizable code for this FSM. The FSM has two input signals, `reset` and `clock`, and a number of control output signals. In this case, there are no input conditions which decide about the next-state selection. The description contains three processes: The process `state_decode_logic` implements the next-state function and the process `output_decode_logic` implements the output function. If the design would have had input condition signals the process `state_decode_logic` would have implemented the next-state selection as additional conditional statements, such as if-statement, in stead of the main case-statement. The process `state_register` implements the change of the state every clock cycle as well as the resetting of the state register to state S0 on the reception of the `reset` signal.

□

3.4.3 Controller Implementation

Depending on the selected style of the controller a different controller structure has to be implemented. However, the basic structure is based on the implementation of a single FSM. This basic implementation, together with additional hardware for complex controller synchronization and communication, is used for other types of controllers.

The single FSM controller can be implemented using random logic, microcode or PLAs. The general implementation structure is very similar, not matter which technique is used. A state register is used to store the current state while combinational logic is used to generate the next state based on the current state and the conditions coming from the data path. The current state is also used to generate control signals for the data path since we assume the use of Moore machines. In some cases, additional decoding and coding can be used for the control signals and the conditions respectively. Figure 3.22 illustrates the general implementation structure of the basic controller.

The basic controller can then be used as a building block to create complex controllers described previously. Both control signals and conditions generated by the basic controller are used for synchronization purpose. In addition, data path elements can be used to create complex communication facilities required by the given controller.


```
entity FSM is
  port (reset, clock : in BIT;
        M1, M2, M3, Add, Mul, Load_R1, Load_R2, Load_R3,
        Load_R4 : out Bit;
        M4 : out Bit_vector (0 to 1));
end FSM;

architecture controller of FSM is
  type state is (S0, S1, S2, S3);
  signal present_state, next_state : state;
begin -- controller
state_register: process (reset, clock)
        begin
          if (reset = '0') then
            present_state <= S0;
          elsif (clock = '1' and clock'EVENT) then
            present_state <= next_state;
          end if;
        end process state_register;

output_decode_logic : process (present_state)
        begin
          M1 <= '0'; M2 <= '0'; M3 <= '0'; M4 <= '00';
          Load_R1 <=  '0'; Load_R2 <=  '0'; Load_R3 <=  '0';
          Load_R4 <=  '0'; Add <= '0'; Mul <= '0';
          case present_state is
            when S0 => M3 <= '0'; Load_R3 <= '1'; M4 <= '00';
              Load_R4 <='1';
            when S1 => Add <= '1'; M2 <= '1'; Load_R2 <= '1';
              M1 <= '1'; Load_R1 <= '1'; M3 <= '1';
              Load_R3 <= '1'; M4 <= '01'; Load_R4 <= '1';
            when S2 => Add <= '1'; M1 <= '0'; Load_R1 <= '1';
              Mul <= '1'; M4 <= '10';
            when S3 => Mul <= '1'; M2 <= '0'; Load_R2 <= '1';
              :
          end case;      -- present_state
        end process output_decode_logic;

state_decode_logic : process (present_state)
        begin
          case present_state is
            when S0 => next_state <= S1;
            when S1 => next_state <= S2;
            when S2 => next_state <= S3;
            when S3 => next_state <= ...;
              :
          end case;       -- present_state
        end process state_decode_logic;
end controller;
```

Figure 3.21: VHDL specification of the FSM given in Figure 3.20c.

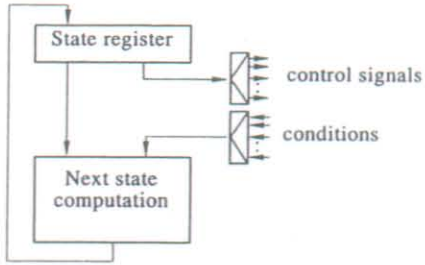


Figure 3.22: The general structure for the controller implementation.

Example 3.16: Figure 3.23 depicts a possible implementation of hierarchical controllers introduced earlier. The parent controller uses one of the control signals as an activation signal for a child controller. The child controller gets this signal as a condition signalling the start of the controller. After finishing its execution the child controller sends another control signal which is interpreted as a completion signal by the parent controller. Both the parent and the child controller have to use the same clock in this case.

□

Parallel controllers can synchronize using a method similar to the one sketched for hierarchical controllers. In more complex situations they require additional hardware for synchronization and data exchange. In general a protocol must be implemented between parallel controllers to provide correct means for communication. The most frequently used protocol is called the handshaking protocol. It uses request-acknowledge signals to establish communication. The controller which starts communication sends a request signal to the other controller and waits for the acknowledge signal. When the other controller receives the request signal it executes the needed operations and after that sends back the acknowledge signal. Both controllers continue their normal execution after the communication.

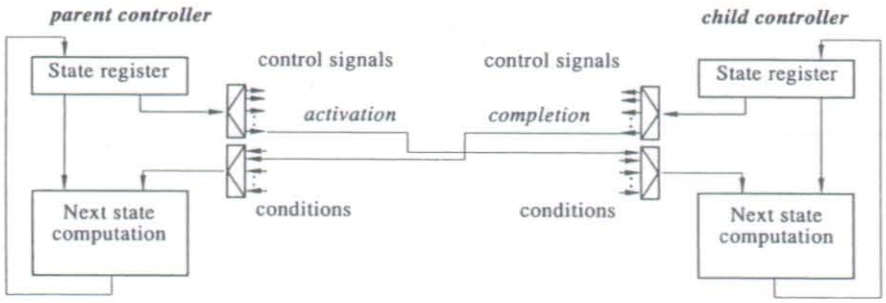


Figure 3.23: A hierarchical controller implementation.

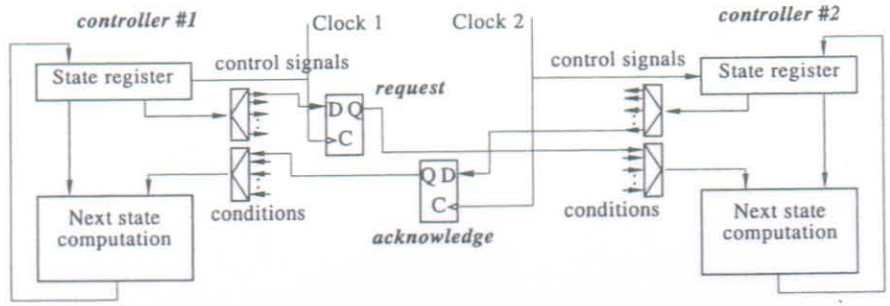


Figure 3.24: An implementation of a handshaking protocol between two parallel controllers.

Example 3.17: Figure 3.24 presents a possible implementation of the request-acknowledge protocol between two parallel controllers using two D flip-flops. Controller *#1* sends a request signal by setting a D flip-flop to 1. It stays in this state until controller *#2* sends an acknowledge signal by setting another D flip-flop to 1. This state is recognized by the first controller which continues its execution. It can be noted that this solution makes it possible to use independent clocks for controllers. For example, controller *#1* uses Clock 1 while controller #2 uses Clock 2, as illustrated in the figure. This means that although both controllers are synchronous themselves the communication between them are performed asynchronously.

□