Chapter 2

SYSTEM DESIGN METHODOLOGIES

In this chapter we will look at different design methodologies, or design flows, for multi-processor systems. Design methodologies have evolved together with manufacturing technology, design complexity, and design automation. Improvements in technology have increased design complexity to the point that designers are no longer capable of making complex designs manually. To solve this problem, design automation tools, also known as computer-aided design (CAD) tools, were introduced. In order to make CAD tools more efficient and design algorithms more manageable, design-automation researchers as well as tool developers were forced to introduce more stringent design rules, parameterize components and minimize component libraries. As design complexities continued to increase, tool developers created new design abstraction levels and tried to use the same design strategy from the circuit level, to the logic level, to the processor level, and finally to the system level.

In this chapter, we will explain some basic system design methodologies related to the different abstraction levels in the Y-chart we introduced in Chapter 1.

2.1 BOTTOM-UP METHODOLOGY

Bottom-up methodology started even before CAD tools were invented. It is still in use in much of the industry today, at least partially, because it follows an intuitive methodology of building parts before assembling the whole product. In a typical bottom-up methodology, designers develop components and then store them in a library for use on the next-higher abstraction level.

As we can see in Figure 2.1, we have libraries of transistor, logic, RTL, and processor components. Components in each of the libraries are used to build

DOI: 10.1007/978-1-4419-0504-8_2,

components in the library on the next abstraction level. On the Circuit level, we use transistors and develop circuits and their layouts for the basic logic components such as gates, flip-flops, bus drivers, and others. These components become standard cells for higher level design and layout tasks. These standard cells, with their functionality, structure, and layout, are stored in the Logic component library for use on the Logic level in Figure 2.1. On the logic level, we create register-transfer components such as registers, register files, ALUs, multipliers, and other components for processor micro architecture using Boolean expressions or FSM and FSMD models. After logic synthesis of these RTL components, we perform the placement and routing with standard cells for each component and store them in the RTL component library. On the Processor level, we start with C code or an Instruction set and generate the structure of processing elements (PEs) or communication elements (CEs). At this level we also perform floorplanning, placement, and routing of these PEs or CEs using the components from RTL library, and store them in Processor library. On the System level, we start with a model of computation (MoC) and generate the system structure consisting of multiple PEs and CEs from the Processor component library. Finally, we perform the system layout by using the component layouts from the Processor library. Note that each component library has functional, structural, and layout models for each component in the library. So by creating components and storing them in libraries, we can then apply them in each successive abstraction level.

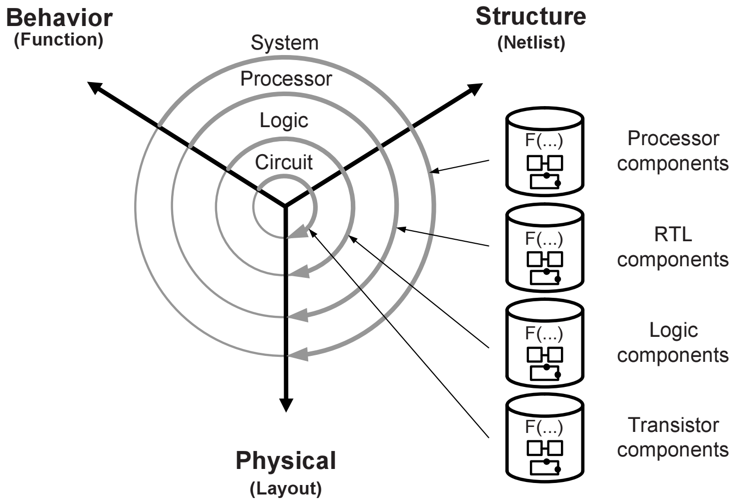

FIGURE 2.1 Bottom-up methodology

The advantage of bottom-up methodology is that abstraction levels are clearly separated, each with its own library. This allows for globally-distributed locations for design on each abstraction level, and for easier management of design on each abstraction level, since each group supplies a component library for the next level of abstraction. The disadvantage of this approach, however, is that the libraries must include all possible components with all possible parameters and that these must be optimized for the metrics required by any present and possible future applications. This is a very difficult and never-ending task since it is very difficult to anticipate on the lower abstraction level all the needs on the next higher abstraction level.

2.2 TOP-DOWN METHODOLOGY

In contrast to bottom-up methodology, top-down methodology does not attempt a component or system layout until the entire design is finished. A top-down methodology begins with a particular MoC and generates from it a system platform or system structure in which every component has its parameters and required metric values defined, but not its structure or layout. On the next level of abstraction, each PE or CE component is further decomposed into smaller RTL components. For example, in Figure 2.2, PE and CE components that were generated on the System level are decomposed into RTL components with their parameters and required metrics defined.

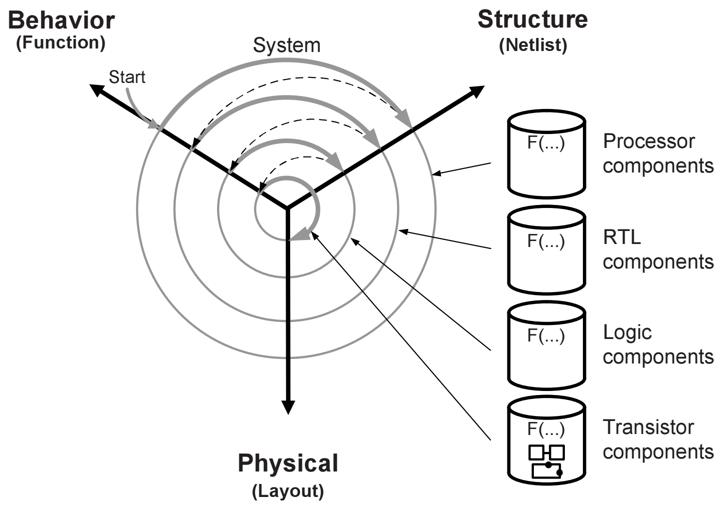

FIGURE 2.2 Top-down methodology

In this case, each functional unit, such as the ALU, has all its functions specified, as well as its delay and power requirements. After those are determined, each of the RTL components is further decomposed into logic components or gates. Finally, each logic component is broken down into a transistor netlist, in which each transistor layout represents a basic cell. All such basic cells, for the entire system, are placed on silicon and connected accordingly using placement and routing methods and tools. Such top-down methodologies were in use in design of early computers but today's designs are too complex for such a complete top-down methodology.

In general, top-down methodology leaves placement and routing for the last step by avoiding the layouts on other levels of abstraction. Unfortunately, the system and component metrics are not known until the last step and therefore it is very difficult to optimize the whole design. The design decomposition or synthesis has to be repeated over and over again without designers really knowing whether optimization is going in the right direction. In order to avoid too many design iterations, designers need the concept of metric closure in which different metric values from lower levels of abstraction are used to annotate design on higher level of abstraction. In this case designers can estimate optimized metric values on the lower level of abstraction during the next design iteration on the higher levels of abstractions. Unfortunately, metric closures are difficult to achieve since metric estimations are as difficult as performing real designs.

2.3 MEET-IN-THE-MIDDLE METHODOLOGY

Most designers today use some kind of meet-in-the-middle methodology [124, 160] in order to take advantage of the benefits of both bottom-up and top-down methodologies, while also minimizing their drawbacks. This is convenient because the design standards and CAD tools on the lower levels of abstractions are well understood and developed, but on the processor and system level they are not. While there are some tools on the processor level, almost none, with exception of general simulation tools, exist on the system level. A meet-in-the-middle methodology allows a designer to take advantage of the tools available for lower level abstractions while also reducing design layouts on higher abstraction levels.

In general, a meet-in-the middle methodology applies a top-down methodology to higher abstraction levels and a bottom-up methodology to lower abstraction levels [124, 160]. The main distinguishing feature of this approach is how these styles meet. As shown in Figure 2.3, a meet-in-the-middle methodology could start with a MoC and synthesize the system platform with virtual PEs and CEs which are after that synthesized with RTL components from the RTL library. These PEs and CEs also include commercially available IPs which are also supplied as netlists of RTL components. Therefore, all PEs and CEs are decomposed into RTL components from the library. Each RTL component has

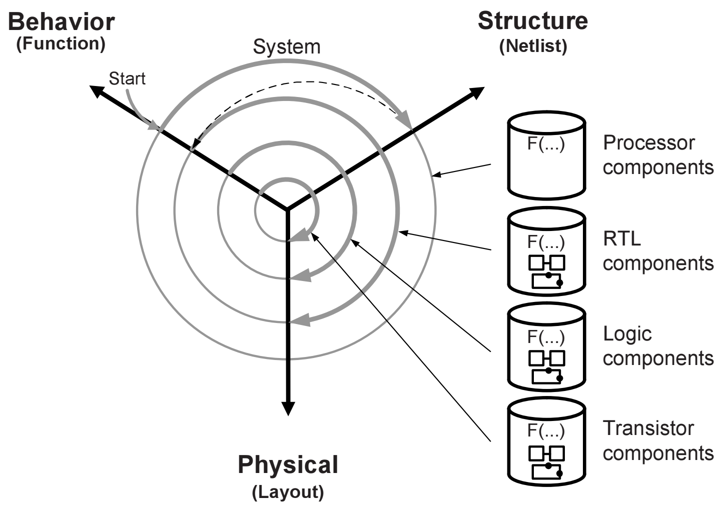

FIGURE 2.3 Meet-in-the-middle methodology (option 1)

its own structure and layout layout generated through some bottom-up methodology in the library. These RTL component layouts are combined through floorplanning, and routing into the layout of the multi-core platform. For example, ALU components in such a library would be limited to lengths of 8, 16, 32, 64 bits, and nothing in between.

Therefore, with such a meet-in-the-middle methodology, we do physical design or layout three times: once for standard cells, a second time for RTL components and a third time for the entire system platform, using the layout of these RTL components. This mixed methodology has the advantages of both bottom-up and top-down methodologies, since RTL components with their metrics are available from the libraries while the system is synthesized top-down from the RTL components. However, this approach has the drawback of requiring designers to do layout more than once. Moreover, system optimization is more difficult using already-made RTL components because they may not be tuned to the requirements of each PE or CE in the targeted system platform.

Another possibility for a meet-in-the-middle methodology would be to perform system layout with logic components or standard cells, as shown in Figure 2.4. As with the first meet-in-the-middle methodology we described, this one starts with a MoC and synthesizes the system platform with virtual PEs and CEs. Those PEs and CEs are then synthesized with RTL components, which themselves are further synthesized with logic components. Commercially available IPs that are described on the RTL level are also synthesized with RTL and logic synthesis tools that generate logic components netlists. Therefore, every IP component, as well as the synthesized PEs and CEs, are

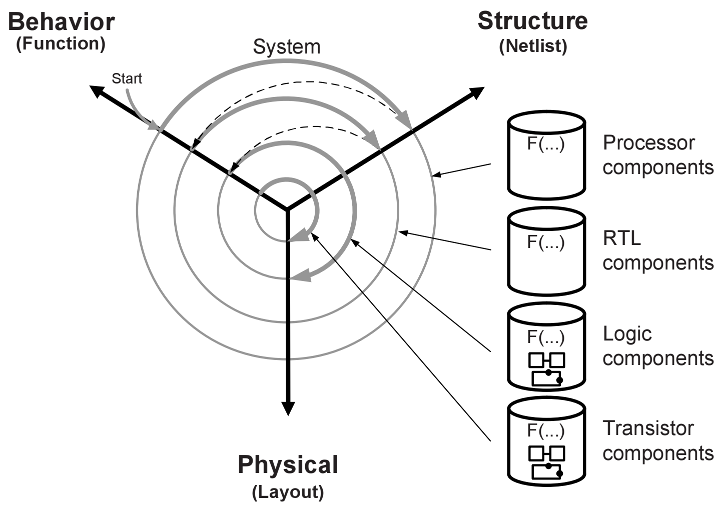

FIGURE 2.4 Meet-in-the-middle methodology (option 2)

decomposed into logic components from the Logic component library. Since each logic component has a layout as a standard cell, they are finally combined through floorplanning and routing into the layout of a multi-core platform.

In this case, we do physical design or layout only twice: once for generating standard cells and a second time for the entire system platform, using the standard cells layouts. This mixed methodology has an advantage in that only the standard cell layout has to be upgraded with the introduction of a new fabrication technology. The RTL component layouts, which are much more complex and in higher numbers do not need to be upgraded. An additional benefit is that the whole design is flattened to standard cells and the layout is performed only once. However, a system layout using standard cells is more complex than it would be with RTL components, and the design metrics are less predictable and controllable since standard cells for each RTL component may not be all in one place. Furthermore, using such inaccurate metrics makes it difficult to perform any system optimization on higher abstraction levels.

2.4 PLATFORM METHODOLOGY

The three design methodologies presented in the previous sections represent ideal cases of three different design concepts. In reality, design methodologies differ from company to company and even between different groups in the same company. They are also very much product oriented [165]. In this case, system design usually starts with an already-defined platform, usually one defined by a

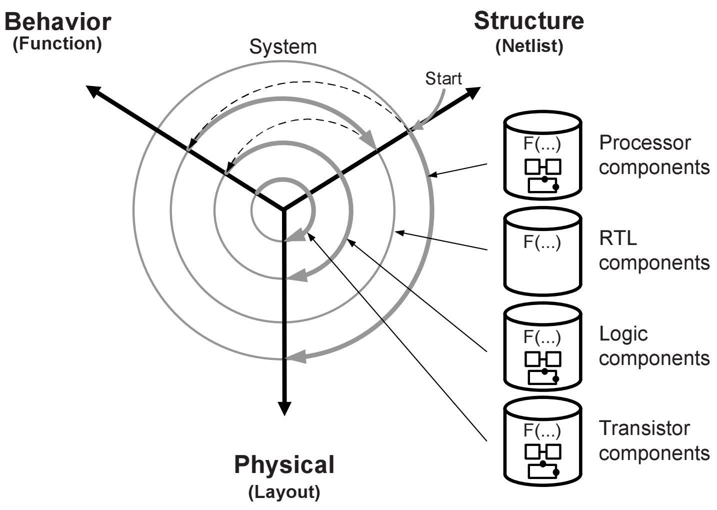

FIGURE 2.5 Platform methodology

well-known platform supplier or defined locally inside the company as shown in Figure 2.5. Such platforms may have already some standard components, such as memories and standard processors with well-defined layouts. The system platform may also be upgraded with the addition of custom components that will be synthesized with processor and RTL synthesis tools, after which the layout of these custom components can be obtained through standard cells. Furthermore, imported IPs are also converted to standard cell layout. Therefore, every custom component or imported IP can be defined with a netlist of standard cells, which is combined with netlists of other custom components for the combined standard cells layout. Such standard cell layout is then combined on the System level with layouts of standard processor and memory components into the system platform layout. When using such a platform, we perform physical design or layout three times: once for standard cells, then we use standard cells for the layout of custom components, and finally we use processor component layouts for the final platform layout.

In order to simplify platform design, some platforms have system layout for all standard processor components finalized with some space left open for the standard cell layout of custom components. When using such a platform, therefore, we perform layout only two times: once for standard cells and second time we use standard cells for the layout of custom components.

This mixed methodology has advantages from both bottom-up and top-down methodologies since standard processor components are available from the libraries and custom components can be inserted for application optimization. However, this approach has the weakness of requiring us to do layout more than once. Also, custom components have to be adapted to reflect the structure and layout requirements of the given platform.

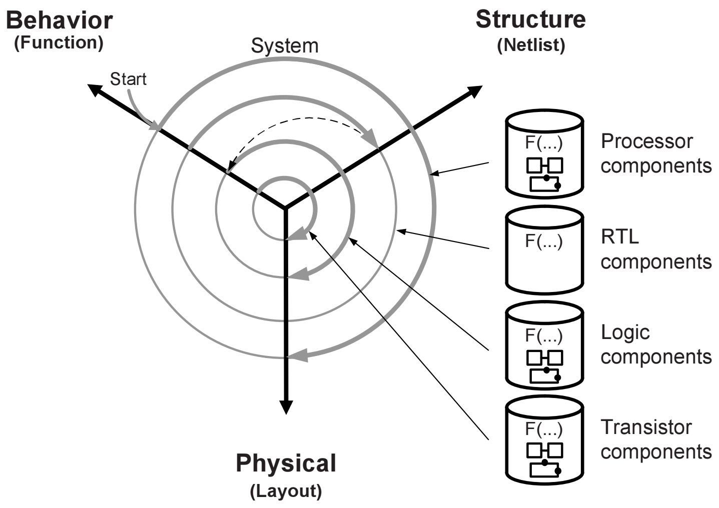

FIGURE 2.6 System methodology

The platform methodology can be upgraded to a system-level methodology by introduction of standard architecture cells and retargetable compilers. An architecture cell contains a parametrizable programmable processor such as one shown in Figure 1.5. The parameters include number, type and size of components, component connectivity, and the number of pipeline stages in the functional units, controller and the datapath. Such a standard architecture cells can be pre-synthesized with standard cells and inserted into the library of Processor components or generated on demand. A typical system-level methodology based on such architecture cells is shown in Figure 2.6. It starts with a MoC and generates the platform architecture consisting of standard or custom architecture cells. Since all the architecture cells have the layout model in the library the final system layout is obtained by combining the layouts of architecture cells.

This methodology has advantage of dealing only with two highest abstraction layers. Therefore, it is well-suited for application experts with minimal knowledge of system and processor design. However, it requires a retargetable compiler to cover different architecture cells.

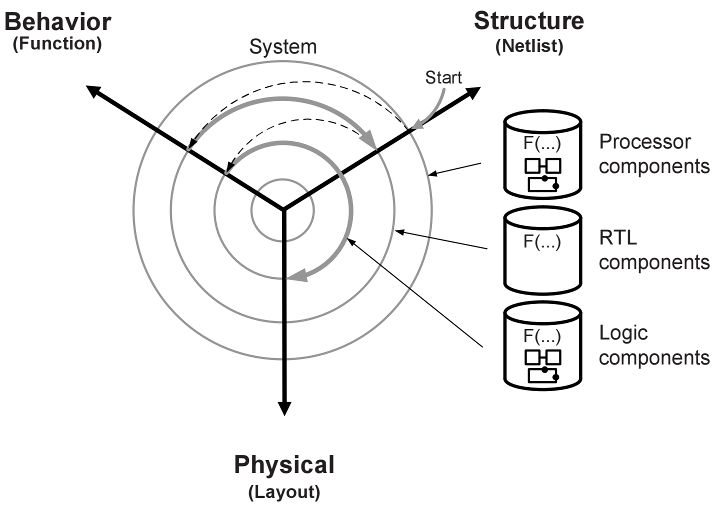

FIGURE 2.7 FPGA methodology

2.5 FPGA METHODOLOGY

Field-Programmable-Gate-Array (FPGA) methodology is based on the FPGA substrate, which consists of a multitude of 4-bit ROM cells called Look-up Tables (LUTs). These LUTs can implement any 4-variable Boolean function. Therefore, in this methodology, every RTL component in the RTL component library must be decomposed into these 4-variable functions. Then, the Processor components are synthesized out of available RTL components.

In other words, a FPGA methodology shown in Figure 2.7 uses a top-down methodology on both the System and Processor levels, in which standard and custom PEs and CEs are all expressed in terms of LUTs. A system design starts by mapping an application onto a given platform and then synthesizing custom components down to RTL components which are defined in terms of LUTs. Standard processors components in the Processor library are already defined in terms of LUTs. Once all components in the platform are defined, we flatten the whole design to LUTs and BRAMs and perform the placement and routing with the tools provided by FPGA suppliers.

This type of top-down system design has the same weaknesses as any top down methodology in that it is difficult to optimize the whole design by flattening the whole design just to basic LUT cells. Furthermore, designers do not know how the FPGA supplier-provided layout tools will map and connect all the LUTs and BRAMs.

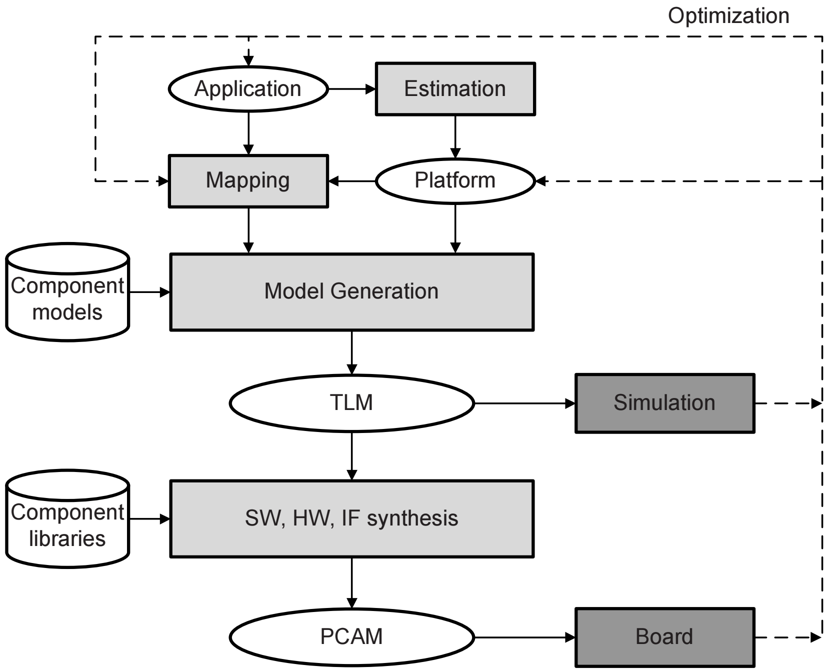

FIGURE 2.8 System-level synthesis

2.6 SYSTEM-LEVEL SYNTHESIS

In the previous sections we described several basic strategies in system design. However, system design flow has been changing alongside fabrication technologies and automation tools over the last 50 years. The changes started with lower levels of abstraction, which are well understood today. However, the higher levels of abstraction are still under investigation and discussion. In this section and the next, we will describe briefly the synthesis process from a behavioral description to a structural description on system and processor levels.

As shown in Figure 2.8, system-level synthesis starts with an application written in some MoC such as a set of sequential and parallel processes communicating through message-passing channels. Such a MoC must execute on a platform of multiple standard and custom processors connected through an arbitrary network. This type of platform can be defined partially or completely after estimating some characteristics of the application in terms of performance, cost, power, utilization, configurability, and other considerations. Platform definition can be done manually or automatically.

Once the platform is defined, an application must be partitioned and each partition assigned to a processor or IP in the platform. In order to verify that the application executes on the platform and satisfies all the requirements, we need to generate a simulatable and possibly verifiable model such as a timed Transaction-Level Model (TLM). After simulation, the design can be optimized if it does not satisfy the requirements by changing the platform, the application code, or the algorithms used in that code. We can also change the mapping of the application to the platform. For example, we can minimize external communication by grouping heavily communicating processes and assigning the whole group to one processor. It is also possible to assign performance-demanding processes to different processors or specialized IPs, or to pipeline performance-demanding processes if possible.

After we obtain a satisfactory application code, platform, and mapping, we can synthesize each component. Three types of components are needed: custom SW, HW, or IF components. SW components are for scheduling of processes such as different types of RTOS, and for communication and interfacing across the platform. HW components are various custom processors and custom hardware units, as described in the previous chapter. We also need communication components such as bridges and transducers for protocol conversion, and interface components such as bus arbiters and interrupt controllers.

Having synthesized these platform components, we need to generate a CAM model that contains binaries for downloading to processors and RTL descriptions for the HW parts in the platform. This can be done automatically or manually. Such a CAM is downloadable to standard FPGA boards for system prototyping, whose results can be used for final optimization of the whole design.

Details on each of these tasks will be given in the chapters that follow.

2.7 PROCESSOR SYNTHESIS

On the processor level, the components are synthesized as standard processors, custom processors, and custom hardware units, which are sometimes called IPs. The standard and custom processors are usually defined by their instruction sets. Custom processors can be also defined by the algorithm or the programming language code that they execute. They are programmable so that new algorithms and the code can be added or existing one modified. Custom hardware units or IP are usually not programmable. They are used as accelerators to execute special functions for a particular application, such as multimedia applications.

As shown in Figure 2.9 the synthesis process starts with a given Specification in a programming language, which is compiled into some Tool model such as

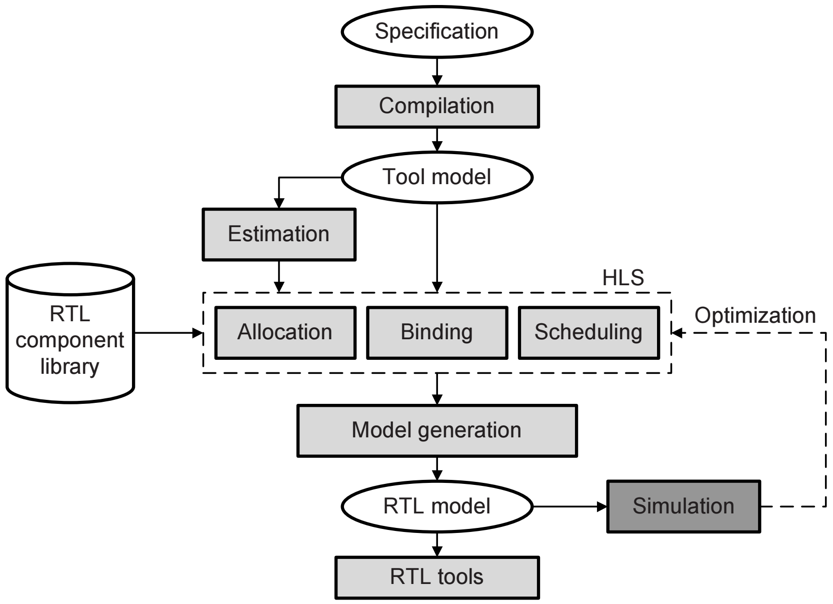

FIGURE 2.9 Processor synthesis

CDFG or a FSMD or a three-address code. This formal model can be used for Estimation of the future processor architecture and its metrics. It can be also used for some partial or complete allocation, binding, and scheduling. Processor synthesis, sometimes called High-Level Synthesis (HLS), takes the formal model and performs Allocation, Binding and Scheduling. The Allocation task selects necessary and sufficient components from the RTL component library and defines their connectivity. The Binding task defines binding of variables to registers, register files, and memories, operations to specific functional units and register-to-register transfers to specific buses. Scheduling assigns operations and register transfers to clock cycles. These three tasks compete with each other, so a completely optimized design is not easy to achieve. That is why estimation and pre-HLS comes handy. Pre-allocation helps in partial or full definition of processor architecture. This way we can avoid the clock-cycle estimates; since many or all of the register-to-register delays are known ahead of time, there is no need to wait until the end of HLS to find out the clock cycle time. Pre-binding may bind frequently-used variables to fast registers, register files, or a scratch-pad memory to avoid lengthily delays caused by loading and storing to the main memory. Pre-scheduling can assign key inner loops to high-speed, pipelined functional units or it can pre-schedule such loops to specific paths in a pipelined datapath.

Once HLS is finished we need to generate a RTL Model of the processor that can be synthesized with standard RTL synthesis tools.

2.8 SUMMARY

In this chapter, we explained the differences between top-down, bottom-up, and meet-in-the-middle methodologies by exposing their taxonomizing structures. We also highlighted some features of key ASIC and FPGA methodologies. We also detailed one methodology branch, synthesis, explaining how the process might work on both the processor and system levels.

However, we have to acknowledge that there are many more design methodologies, almost one for every group, product, and company [103, 47, 63, 100, 129, 195, 184]. They may start with different specifications, may use other models for verification of different concepts and metrics, and they may need a different type of outputs. However, all design methodologies must address the basic system needs and issues we have introduced in this chapter These methodology issues will be discussed in more detail in the succeeding chapters.