ILP based scheduling

OBI NNAMDI ELIJAH

***Abstract*—This paper seeks to present an overview behind the approach of ILP (Integer Linear Program) based scheduling which typically involves the use of a formal mathematical description for all of the resource and application constraints, e.g., an Integer Linear Programming (ILP) formulation, which can be used by different solvers to compute a feasible schedule. All variables must be set to integers in order to have an optimal solution.**

***Index Terms*—CDFG - Control Data Flow Graph, DFG - Data Flow Graph, HLS - High Level Synthesis, ILP - Integer Linear Programming**

I. INTRODUCTION

In the 2000s, there was a transition to an electronic system-level (ESL) paradigm that facilitated the exploration, synthesis, and verification of complex SoCs. This brought about the advent of languages having system-level abstractions, such as SystemC, SpecC, or SystemVerilog, as well as the introduction of transaction-level modeling (TLM). The rise of system complexities, a plethora of components in a product (hundreds of processors in a car, for example), a wide selection of chips of different versions (for better product differentiation), and the interdependence of component suppliers forced the market to focus on hardware and software productivity; dependability; interoperability; and re-usability. Processor customisation and HLS have become important approaches to effective ESL design in this setting. [4]

In high-level synthesis, ILP modeling consists of an objective function that must be minimized or maximized given the specified constraints. Most techniques use scheduling domains to minimize design space and runtimes. When modeling an integer linear programming model, some or all of the variables used to describe the objective function and constraints are limited to being integers. In order to accomplish simultaneous scheduling and binding, the constraints in an integer linear programming model must be appropriately stated. The Control steps are assigned to each DFG node based on the constraints and it is commonly known as ”time steps” and are the basic sequencing units in synchronous systems. During binding, operators are assigned to minimize the hardware required to implement the data flow graph. The work of synthesis begins with a behavioral description of a digital system as well as a set of time and/or resource restrictions. The objective is to create a digital system structure that meets the requirements. It is divided into four basic subtasks. [5]

The first subtask is to use a hardware description language to describe the behavior of the digital system (HDL). This is frequently followed by the description being translated into a graph-based form known as the control data flow graph (CDFG). The next subtask is operation scheduling, which involves assigning a control step to each operation in the CDFG. The third subtask mainly allocates the resources. Function units are assigned to perform the operations, storage units are assigned to store the values, and wires are assigned to connect them using data transfer information provided from the CDFG. A data route is now complete. Finally, a control unit is created based on the scheduling graph and data path to synchronize the operation executions. The two key subtasks among the aforementioned processes are operation scheduling and hardware allocation. These two subtasks are interrelated. A system should do both subtasks concurrently in order to have an optimum design [l]. However, due to the temporal complexity, many systems process them individually or include iteration loops between the two subtasks. [5]

II. HIGH LEVEL SYNTHESIS (HLS)

High-level synthesis (HLS) has a lengthy history. By the late 1970s, HLS had established itself as an active research discuss in the EDA community, and it was frequently referred to as ”the next big thing” by the early 1990s, following the considerable and extremely successful adoption of logic synthesis [1]. Machine code (binary sequence), for instance, was previously the sole language that was mainly used to program a computer. The notion of assembly language (and assembler) was initially developed. Finally, high-level languages (HLLs) and other related compilation methodologies were created to enhance software productivity. HLLs, are known to be independent of various platform, and adheres to human language principles and structure such as grammar, syntax, and semantics. With this, we are able to realise the requirements of flexibility and portability by hiding details of the computer architecture. With the increasing complexity of current system architectures and software applications, it is apparent that using HLLs and compilers produces superior overall outcomes.

Similarly, specification languages and design techniques have emerged in the hardware realm. [2], [3] Looking back, until the late 1960s, integrated circuits were developed, optimized, and set out mainly by hand. Gate-level simulation first appeared in the early 1970s, and cycle-based simulation was accessible by 1979. Place-and-route, schematic circuit capture, formal verification, and static timing analysis were all introduced in the 1980s. Hardware description languages (HDLs) like Verilog (1986) and VHDL (1987) have facilitated the widespread use of simulation tools. These HDLs have also been used as inputs to circuit synthesis tools, resulting in the construction of synthesizable subsets. The first wave of commercial high-level synthesis (HLS) tools were available in the 1990s.Around the same time, research interest in hardware-software co-design, covering topics such as estimation, exploration, partitioning, interfacing, communication, synthesis, and

co-simulation, grew exponentially. The idea of IP core and platform-based design began to also grow.

The new HLS flows in such a way that it reduces the time required to create the hardware, and also aids in reducing the time required to test it and support additional flows such as power analysis. As a result, HLS is used in a wide variety of production designs, including memory and DMA controllers, cache controllers, power controllers, interconnect fabrics, bus interfaces, and network traffic switches, as well as computation-acceleration IPs like image and graphic processors and DSP algorithms. [4]

A. WHAT IS HIGH LEVEL SYNTHESIS (HLS)

As the name implies, HLS often begins with a higher degree of abstraction design description than RTL, the most popular design-entry point in use today. Raising the abstraction level of the hardware design is critical for assessing system-level exploration for architectural decisions such hardware and software design, synthesis and verification, memory organization, and power management. HLS also allows for the reuse of the same high-level specification, which is intended to handle a diverse set of design restrictions and ASIC or FPGA technology. A designer typically starts with the specification of an application that will be implemented as a custom processor, dedicated co-processor, or any other custom hardware unit such as interrupt controller, bridge, arbiter, interface unit, or special function unit with a high-level description capture of the desired functionality using an HLL. [4] This initial stage entails constructing a functional specification (an untimed description) in which a function consumes all of its input data at the same time, executes all computations without delay, and outputs all of its output data at the same time. Variables (structure and array) and data types (usually floating point and integer) are not connected to the hardware design domain (bits, bit vectors) or the embedded program at this abstraction level.Thus, realistic hardware implementation necessitates the conversion of floating-point and integer data types into bit-accurate data types of specific length (rather than a standard byte or word size, as in software) with acceptable computation accuracy, followed by the generation of an optimized hardware architecture based on this bit-accurate specification.

HLS tools can also convert an untimed (or partially timed) high-level specification into a fully timed implementation. They do this by constructing a bespoke architecture automatically or semiautomatically in order to efficiently implement the requirements. In addition to the memory banks and communication interfaces, the created architecture is detailed at the RTL level and includes a data route (registers, multiplexers, functional units, and buses) and a controller, as required by the supplied specification and design limitations. The input language contributes to some of the abstraction of HLS. Most HLS tools now in use use C++ as their input language, either directly or through the SystemC extension of C++. Tools based on proprietary languages have not gained popularity, owing to a lack of compilers and debuggers, as well as a lack of legacy code. SystemC is a C++ extension that may be implemented by a C++ library, allowing it to be created using conventional C++ compilers and debuggers. The C++ programming language was designed to construct sophisticated software systems and has a variety of features that aid with abstraction and encapsulation. This they do by giving the hardware designer the ability to apply abstraction techniques developed for software in a way that goes far beyond the capabilities of typical hardware design languages(HDLs) such as VHDL, Verilog. [4]

In addition to C++ abstraction methods, HLS tools may optimize hardware-specific design challenges, allowing for extra abstraction of the design description. These features can boost designers' productivity by automating time-consuming and error-prone processes like providing low-level details. They also make the input description more reusable by eliminating implementation-specific information and allowing tools to identify them instead, subject to the tools' own implementation restrictions. HLS tools aid RTL designers by optimizing resource sharing automatically. This allows for better encapsulation of functionality, making it easier to comprehend and extend. They are also in charge of scheduling, which is the assignment of operations to cycles.This transformation is comparable to logic synthesis retiming, but HLS scheduling frequently modifies the architecture much more drastically than logic synthesis retiming. [4]

B. KEY CONCEPTS IN HIGH LEVEL SYNTHESIS (HLS)

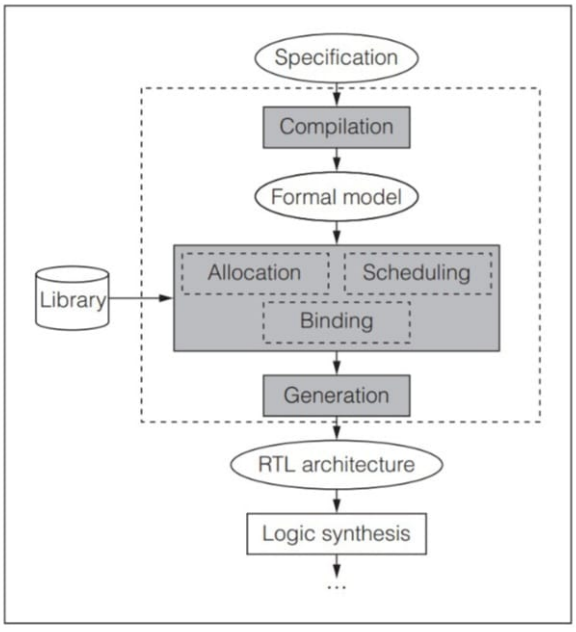

Fig. 1. HLS Design steps

[3]

An HLS tool performs the following functions (see Figure 1) based on a high-level description of an application, an RTL component library, and certain design constraints:

Compiles the specification,

Allocates hardware resources (functional units, storage components, buses, and so on),

Schedules the operations to clock cycles,
Binds the operations to functional units,
Binds variables to storage elements,
Binds transfers to buses, and
generates the RTL architecture.

[3] Tasks 2 through 6 are interrelated, and for a designer to reach the best solution, they should be optimized together. However, in order to manage the computing complexity of synthesis in real-world designs, the activities are generally conducted in succession. The sequence of certain of the synthesis activities, as well as a measure of how effectively the interdependencies are calculated and accounted for, have a substantial impact on the quality of the resulting design.

1) Compilation and modeling: The functional specification is always compiled first in HLS. This is the initial stage in converting the input description into a formal representation. Traditionally, this initial stage involves code optimizations such as dead-code reduction, false data dependence elimination, and constant folding and loop transformations. The formal model created by the compilation shows the data and control relationships between the processes in a traditional way. A data flow graph (DFG) may readily express data dependencies since each node represents an activity and the arcs between the nodes indicate the input, output, and temporary variables. A pure DFG solely models data dependencies. In certain circumstances, this model can be obtained by eliminating the basic specification's control dependencies from the model at compilation time. To achieve this, Loops are unrolled fully by converting to noniterative code blocks, and conditional assignments are addressed by producing multiplexed values. The generated DFG expresses all of the specification's inherent parallelism. However, the requisite transformations might result in a massive formal representation that necessitates a significant amount of memory used to store it during synthesis. The usage of pure DFG representations is thus limited to a few applications.

The control and data flow graphs have been added to the DFG model to incorporate control dependencies (CDFG).A CDFG is a directed graph with control flow represented by its edges. A CDFG node is a straight-line series of statements with no branching or internal entrance or exit points. Conditional edges can express if and switch operations. A CDFG demonstrates data dependencies inside basic blocks and records control flow between them. Because they may describe loops with unbounded iterations, CDFGs are more expressive than DFGs. However, parallelism is only explicit inside basic blocks; extra analysis or transformations are necessary to uncover parallelism that may exist between basic blocks. Loop unrolling, loop pipelining, loop merging, and loop tiling are some examples of such transformations. [3]

These methodologies are used to optimize the latency or throughput, as well as the size and number of memory accesses, by disclosing the parallelism across loops and between loop iterations. These modifications can be carried out automatically or manually. Data dependencies between basic blocks can be introduced to the CDFG model in addition to control dependencies, as seen in the hierarchical task graph representation used by the SPARK tool.

2) Allocation: Allocation specifies the type and quantity of hardware resources (for example, functional units, storage, or connection components) required to meet design restrictions. Some components may be introduced during the scheduling and binding operations, depending on the HLS tool. Connectivity components (such as buses or point-to-point connections between components, for example) can be introduced before or after binding and scheduling operations. The RTL component library is used to pick the components. It is critical to include at least one component in the specification model for each operation. To be utilized by other synthesis jobs, the library must additionally include component attributes (such as area, latency, and power) and metrics.

3) Scheduling: All operations needed by the specification model must be scheduled into cycles. In other words, variables b and c must be read from their sources (either storage components or functional-unit components) and brought to the input of a functional unit that may perform operation op, and the output a must be brought to its destinations for each operation such as $a = b$ op c. (storage or functional units). The operation might be planned inside one clock cycle or over multiple cycles, depending on the functional component to which it is mapped. Operations can be linked together (the output of an operation directly feeds an input of another operation). Operations can be scheduled to run in parallel if there are no data dependencies between them and enough resources are available at the same time.

4) Binding: Each variable that maintains values across cycles must be associated with a storage unit. Furthermore, variables with non-overlapping or mutually exclusive lifespan can be tied to the same storage units. Every operation in the specification model must be assigned to a functional unit that is capable of carrying out the operation. If many units have this capability, the binding method must optimize the selection. Connectivity binding, which requires that each transfer from component to component be bound to a connection unit such as a bus or a multiplexer, is also required in storage and functional unit binding. High-level synthesis should ideally estimate the connection delay and area as early as feasible so that subsequent HLS phases may better optimize the design.Another option is to detail out the entire architecture during allocation so that preliminary floor planning findings can be used during binding and scheduling.

5) Generation: The purpose of the RTL architecture generation stage is to apply all design decisions made in the preceding tasks of allocation, scheduling, and binding and build an RTL model of the synthesized design.

6) Architecture: A collection of register-transfer components implements the RTL architecture. It generally consists of a controller and a data path (see Figure 2). A data path is made up of storage components (such registers, register files, and memory), functional units (like ALUs, multipliers, shifters, and other specialized functions), and connecting elements (such as tristate drivers, multiplexers, and buses). All of these register-transfer components can be assigned in various amounts and types, and they can be connected dynamically through buses. Each component can be executed in one or more clock cycles, can be pipelined, and can have input or

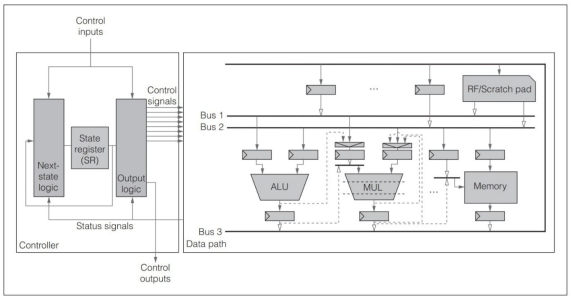

Fig. 2. HLS Design steps

[3]

output registers. Furthermore, the complete data flow and controller may be pipelined in phases. [3]

The design's primary input and output ports communicate with the outside world to convey data and control (used for interface protocol handshaking and synchronization). The data path is connected to the data inputs and outputs, while the controller is connected to the control inputs and outputs. Control signals from the controller to the data path are also present, as are status signals from the data path to the controller. However, other designs may lack all of the connection indicated above, and in general, some of the controller functions may be implemented as part of the data path, such as a counter and other logic in the data stream that creates control signals. [3]

The controller is a finite state machine that orchestrates data flow along the data path by setting the values of control signals (also known as control words) such as select inputs of functional units, registers, and multiplexers. The controller's inputs might originate from either main inputs (control inputs) or data path components such as comparators and so on (status signals). A state register (SR), next-state logic, and output logic comprise the controller. The SR contains the processor's current state, which is the same as the current state of the finite-state machine (FSM) model defining the controller's functioning.A basic dedicated coprocessor's controller is traditionally designed with hardwired logic gates. A controller, on the other hand, can be programmable with read-write or read-only program memory for a specific custom processor. [3]

C. RESOURCE AND LATENCY

III. BASIC SCHEDULING TECHNIQUES

Before delving into the algorithms for unconstrained scheduling, it would be idle for us to discuss the problem's importance. When dedicated resources are employed, unconstrained scheduling is used. Practical scenarios that lead to dedicated resources are those in which operations differ in type or have a little cost when compared to steering logic, registers, wiring, and control. Unconstrained scheduling is also employed when resource binding occurs before scheduling and resource conflicts are resolved by serializing operations that uses the same resource. The area cost of an implementation in this case, is first defined and independent of the scheduling step. Unconstrained scheduling can eventually be used to determine latency bounds for constrained systems. Unconstrained scheduling can compute a lower bound on latency since the minimal latency of a schedule with some resource limitation is obviously at least as high as the latency computed with limitless resources. [5]

A. UNCONSTRAINED SCHEDULING: AS SOON AS POSSIBLE (ASAP)

The most basic scheduling strategy is "as soon as possible" (ASAP) scheduling, in which the activities in the CDFG are scheduled step by step from the first control step to the last. If all of its predecessors are scheduled, an operation is said to be ready. This technique regularly schedules ready operations to the next control step until all operations are scheduled. By topologically sorting the vertices of the sequencing graph, the unconstrained minimum-latency scheduling issue can be solved in polynomial time. This method is known as as soon as possible (ASAP) scheduling because the start time for each operation is the earliest one permitted by the dependencies. We denote by $\boldsymbol{t}^s$ the start times computed by the ASAP Algorithm i.e by a vector whose entries are $\{t_i{}^s; i = 0, 1, ..., n\}$

```
ASAP ( G_s(V, E)) {
    Schedule v_0 by setting t_0^S = 1;
    repeat {
        Select a vertex v_i whose predecessors are all scheduled;
        Schedule v_i by setting t_i^S = max_{j:(v_j,v_i)∈E} t_j^S + d_j;
    }
    until (v_n is scheduled) ;
    return (t^S);
}
```

Fig. 3. ASAP Pseudo Algorithm

[6]

1) Architecture: Due to the limitation on the number of function units, it is not practicable to assign too many operations of the same type into a control step. A modification of ASAP is to delay the ready operations when their number exceeds the number of function units. The operations to be delayed are chosen at random. This method is known as ASAP with conditional postponement.

B. LATENCY-CONSTRAINED SCHEDULING: AS LATE AS POSSIBLE (ALAP)

similar method to ASAP scheduling, however unlike ASAP, ALAP schedules operations from the final control step to the first. As all of its successors are planned, an operation is scheduled to the next control step. The issue with ASAP and ALAP scheduling is that when resource use is limited, no priority is given to activities on critical pathways. As a result, less critical activities can be scheduled first, effectively blocking critical ones.

1) Architecture: To achieve the overall latency constraint, an operation can only be started at one given time step and this is termed as zero mobility. When the mobility is greater than zero, it measures the time interval in which it may be initiated.

```
ALAP( G_s(V, E), λ̄ ) {
    Schedule v_n by setting t_n^L = λ̄ + 1;
    repeat {
        Select vertex v_i whose successors are all scheduled;
        Schedule v_i by setting t_i^L = min_{j:(v_i,v_j)∈E} t_j^L − d_i;
    }
    until (v_0 is scheduled) ;
    return (t^L);
}
```

Fig. 4. ALAP Pseudo Algorithm

[6]

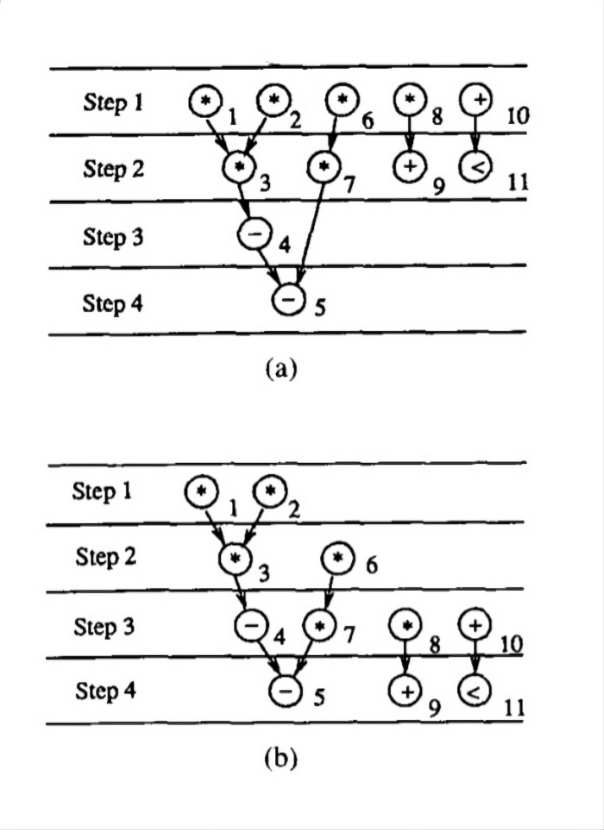

Fig. 5. ALAP and ASAP DFG

[6]

2) Architecture:

C. SUMMARY OF OTHER BASIC SCHEDULING TECHNIQUES

Many high-level synthesis systems have adopted the list scheduling approach, which was first employed in microcode compaction. The activities in the CDFG, like ASAP, are allocated to control steps from the first to the last. According to heuristic rules, ready operations are assigned a priority and scheduled into the next control phase based on this priority. The remaining operations are postponed when the number of scheduled operations exceeds the quantity of resources.

A popular way of prioritizing list involves labeling the vertices with weights(numerical values) based on their longest path to the sink and ranking them in decreasing order. The most critical operations are prioritized. It is said that the algorithm produces an optimal solution for tree-structured sequencing graphs only when the operations have a unit latency/delay and there is only one resource type. List scheduling may manage a schedule with resource and a relative timing constraints. Minimum temporal limitations, for instance, can be met by postponing the selection of operations from the candidate set. The priority list is updated to reflect the proximity of an unscheduled operation to a deadline in conjunction with a maximum time constraint.

The other form of scheduling is ”global” in the sense that it picks the next operation to be scheduled and determines in which control step it should be inserted. They are two types: freedom-based scheduling and forced scheduling. The operations on the critical path are scheduled first In freedom-based scheduling. The operations that are not on the critical path are allocated one by one based on their degree of freedom. In constraint scheduling, force values are calculated for all operations at all conceivable control steps. The operation-control step pairing with the greatest enticing force is selected and assigned. The forces of the operations that have not been scheduled are re-evaluated after allocation. The allocation and evaluation is iterative until all operations are allocated. List scheduling requires the number of functional units to be specified, while force-based scheduling requires the maximum number of control steps to be specified. They correspond to time-constrained and resource-constrained scheduling, respectively.

IV. INTEGER LINEAR PROGRAMMING (ILP)

In mathematical programming, integer linear programming (ILP) models are commonly employed. Integer linear programming is a mathematical optimization technique in which some or all variables are constrained to an integer and the solutions are discrete after a global search. Integer linear programming is the formal method for optimal solution of planning and constraint scheduling problems. The way these models are described and their objective functions varies. Integer linear programming methods are more appealing since they yield optimum solutions and can be quickly adjusted to account for new constraints. But there are drawbacks here as well, such as computational complexity and longer execution times.

V. ILP FORMULATIONS FOR THE SCHEDULING PROBLEM

In this section, We shall give major focus to ILP formulations based on time-constrained scheduling, resource-constrained scheduling, and feasible scheduling. For a more general approach, ASAP, ALAP, and list scheduling are used to trim the solution space in our formulation. That is, ASAP determines the earliest possible time, ALAP determines the latest possible time of the operation, and finally the List scheduling sets an upper limit on the number of control steps for resource-constrained scheduling. As can be seen from [5]:

The notations used in our formulations are defined as follows: suppose the data flow graph, $G(V, E)$ contains $n(|V|)$ operations, $e(|E|)$ data dependencies, and is going to be scheduled into s steps. Each of the operations is labeled as o_i,, where $1 \leq i \leq n$. A precedence relation between two

operations o_i, and o_j is denoted by of $o_i \rightarrow o_j$, where o_i, is the immediate predecessor of o_j. The earliest possible time (ASAP) and the latest possible time (ALAP) of o_i of are S_i, and o_j,, respectively. The cost of a function unit of type $t_k(FU_{tk})$ is C_{th}, and there are m types of function units available. A relation between an operation o_i, and a function unit FU_{tk}, is denoted by $O_i \in FU_{tk}$, if FU_{tk}, can perform the function of O_i.

A. TIME-CONSTRAINED SCHEDULING

From the research paper of [5], we can formulate a definition of time-constrained scheduling as follows:

A time-constrained scheduling problem can be defined as follows. Given a maximum number of control steps, find a schedule with minimum cost that satisfies the given set of constraints. Here, the cost of a data path can be the cost of functional units, connections, and registers. To simplify the formulation, only the costs of functional units are considered. The others will be considered in the next section. It is obvious that the cost of functional units is minimized when all functional units in a system are fully utilized. In other words, operations of the same type should be equally distributed among all control steps. This is achieved in our model by minimizing the maximum number of operations of the same type in each control step.

From my research, I found that an approach to time-constrained scheduling includes three sub steps:

1) ASAP: This is used to find the earliest feasible time for each operation;
2) ALAP: This is used to find the latest feasible time for each operation;
3) ILP: This is used to reduce the cost of resources.

The following variables are utilized in the formulation:

1) M_{tk} are integer variables that represent the number of function units of type t_k that are required
2) $x_{i,j}$ are 0-1 integer variables related with o_i . If o_i is scheduled into step j, then $x_{i,j} = 1$; otherwise, $x_{i,j} = 0$.

The problem may now be stated as minimizing: In (1), it is

$$\sum_{k=1}^{m} (c_{t_k} * M_{t_k}) \tag{1}$$

subject to

$$\sum_{o_i \in FU_{t_k}} x_{i,j} - M_{t_k} \leq 0, \quad \text{for } 1 \leq j \leq s,\ 1 \leq k \leq m; \tag{2}$$

$$\sum_{j=S_i}^{L_i} x_{i,j} = 1, \quad \text{for } 1 \leq i \leq n; \tag{3}$$

$$\sum_{j=S_i}^{L_i} (j * x_{i,j}) - \sum_{j=S_k}^{L_k} (j * x_{k,j}) \leq -1, \quad \text{for all } o_i \rightarrow o_k. \tag{4}$$

Fig. 6.

[5]

specified that our aim function will be to reduce the overall cost of functional units. Constraint (2) stipulates that no control step in a schedule shall include more than M_{tk}, function units of type t_k. It is obvious that o_i, can only be scheduled between step S_i and L_i, as evidenced by (3). Constraint (4) assures that the data flow graph's (DFG) precedence relations are kept.

For a detailed illustration of the above formulation, [5] has examples that adequately covers this domain.

B. RESOURCE-CONSTRAINED SCHEDULING

Formally, a resource-constrained scheduling problem is as follows: Find the quickest schedule that fulfills the given set of constraints given a maximum number of resources. In general, the number of functional units, such as adders, multipliers, ALUs, and buses, is supplied. Despite the fact that registers and connections contribute to overall area, they are difficult to express as resource limitations.

The four substeps of resource-constrained scheduling are as follows:

1) List scheduling: This is used to find the upper limit on the number of control steps;
2) Modified ASAP: This is used to find the earliest feasible time for each operation;
3) Modified ALAP: This is used to find the latest feasible time for each operation;
4) ILP: This is used to reduce the cost of resources. [5]

In order to get a tighter range for each operation, our initial ASAP and ALAP scheduling would have to be modified by taking the resource constraints into consideration. An excerpt from [5] shows that: *Assume there are p_i operations which are executed by the same type of function unit as O_i and proceed to O_i, and the available number of function units for O_i is n_i,. Then, O_i cannot be scheduled before step $\lceil p_i/n_i \rceil$. It also follows that any successor of O_i, cannot be scheduled before $\lceil p_i/n_i \rceil + 1$. Modified ASAP and ALAP are particularly useful when the number of function units or buses is small.*

The variables used in the formulation as seen from [5] are the following:

1) C_{step} is an integer variable which is the total number of control steps required.
2) $x_{i,j}$,j are 0-1 integer variables associated with O_i, $x_{i,j} = 1$, if O_i is scheduled into step j; otherwise, $x_{i,j} = 0$.

A resource-constrained scheduling problem is formulated as minimizing:

$$C_{step} \tag{1}$$

with respect to:

$$\sum_{o_i \in FU_{tk}} x_{i,j} \leq M_{tk}, \qquad for \quad 1 \leq j \leq s, \quad 1 \leq k \leq m; \tag{2}$$

$$\sum_{j=S_i}^{L_i} x_{i,j} = 1 \qquad for \quad 1 \leq i \leq n; \tag{3}$$

$$\sum_{j=S_i}^{L_i} (j * x_{i,j}) - \sum_{j=S_k}^{L_k} (j * x_{k,j}) \leq -1, \quad for\ all\ o_i \rightarrow o_k; \tag{4}$$

$$\sum_{j=S_i}^{L_i} (j * x_{i,j}) - C_{step} \leq 0, \quad for\ all\ o_i\ without\ successors. \tag{5}$$

The objective function in 1 indicates that the total number of control steps will be minimized. Constraint 6 stipulates that no control step in a schedule shall include more than M_{tk} function units of type t k. It should be noted that M_{tk} in 6 is a constant. Constraints 7 and 8 are identical to those found in time-constrained scheduling. As specified in constraint 5, no operations should be scheduled after C_{step}. [5]

Once again, for a detailed illustration of the above formulation, [5] has examples that adequately covers this domain.

C. FEASIBLE SCHEDULING

The preceding two formulations are combined into a third scheduling problem. This approach does not seek an optimum, but rather determines if a feasible solution exists. A feasible scheduling task is defined as follows: *given a fixed number of resources and a specified number of time steps, decide if there is a schedule which satisfies all the constraints. Output the solution if it exists.*

The problem formulation does not include an objective function, but it does provide a set of constraints:

$$\sum_{o_i \in FU_{tk}} x_{i,j} \leq M_{tk}, \qquad for \quad 1 \leq j \leq s, \quad 1 \leq k \leq m; \tag{6}$$

$$\sum_{j=S_i}^{L_i} x_{i,j} = 1 \qquad for \quad 1 \leq i \leq n; \tag{7}$$

$$\sum_{j=S_i}^{L_i} (j * x_{i,j}) - \sum_{j=S_k}^{L_k} (j * x_{k,j}) \leq -1, \quad for\ all\ o_i \rightarrow o_k; \tag{8}$$

The answer for earlier scheduling difficulties can be developed by addressing feasible scheduling problems.

Below are the advantages of using the feasible scheduling approach:

1) Because while solving an ILP formulation, we pick a set of values for all variables, the time complexity grows with the number of variables in the formulation. Because the number of functional units and time are constrained, the domain for each operation is smaller from this perspective. This translates into a reduced solution space. [5]
2) The formulation is a 0-1 ILP problem, and there are strong strategies for solving this sort of issue. Furthermore, because we only need to explore a portion of the solution space, the time necessary to locate a solution via feasible scheduling is much shorter than that required by optimal scheduling. [5]
3) The speed-time trade-off may be controlled by a person or an expert system using this method. As a result, we may develop a collection of optimum solutions and leave the aspect of choosing the best time/area implementation. to the user to decide. [5]
4) Other faster heuristics methodologies can be used to estimate the number of function units and time steps. [5]

Based on the ideas presented above, feasible scheduling appears to provide a generic model for solving scheduling problems. [5]

D. COMPLEXITIES OF THE SCHEDULING PROBLEM

The number of variables and equations is the criteria used to determine the difficulty of feasible scheduling. The amount of resources and control steps have been determined in feasible scheduling. As a result, the only unknowns are the 0-1 variables $x_{i,j}$. The precise value of $x_{i,j}$ is $\sum_{i=1}^{n}(L_i - S_i + 1)$, which is constrained by $s.n$. Because of constraint 7, only n variables will have a value of 1. As a result, once $x_{i,j}$ is determined to be 1, the remaining $L_i - S_i$ variables are automatically set to 0. As a result, the problem is simpler to solve than it appears.

The number of equations necessary for constraints 6, 7, and 8 is $(s.m)$, n, and e, respectively, where e is the number of edges in the DFG. Overall, the number of variables in our formulation rises as $O(s.n)$, and the number of equations grows as $O(s.m + n + e)$.

VI. CONCLUSION

In the course of this paper, having looked briefly on the methodology of list scheduling, ALAP, ASAP and ILP scheduling problems in high level synthesis. It can be categorically said that ILP formulation is very efficient, and its complexity in the number of variables is O(s * n) where s and n are the number of control steps and operations, respectively. With the feasible scheduling formulation, we can explore the solution space more efficiently.

REFERENCES

[1] R. Camposano, W. Wolf, High-Level VLSI Synthesis, Kluwer Academic Publishers, 1991.
[2] A. Sangiovanni-Vincentelli, "The Tides of EDA," IEEE Design and Test, vol. 20, no. 6, 2003, pp. 59-75.
[3] D. MacMillen et al., "An Industrial View of Electronic Design Automation," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, 2000, pp. 1428-1448.
[4] An Introduction to High-Level Synthesis, Coussy, Philippe and Gajski, Daniel and Meredith, Michael and Takach, Andres,
[5] A Formal Approach to the Scheduling Problem in High Level Synthesis Cheng-Tsung Hwang, Jiahn-Humg Lee, and Yu-Chin Hsu, Member, IEEE
[6] SYNTHESIS AND OPTIMIZATION OF DIGITAL CIRCUITS by Giovanni De Micheli