ABSTRACT

KIM, TAEMIN. Exploration of High-level Synthesis Techniques to Improve Computational Intensive VLSI Designs. (Under the direction of Professor Xun Liu).

Optimization techniques during high level synthesis procedure are often preferred since design decisions at early stages of a design flow are believed to have a large impact on design quality. In this dissertation, we present three high-level synthesis schemes to improve the power, speed and reliability of deep submicron VLSI systems. Specifically, we first describe a simultaneous register and functional unit (FU) binding algorithm. Our algorithm targets the reduction of multiplexer inputs, shortening the total length of global interconnects. In this algorithm, we introduce three graph parameters that guide our FU and register binding. They are flow dependencies, common primary inputs and common register inputs. We maximize the interconnect sharing among FUs and registers. We then present an interconnect binding algorithm during high-level synthesis for global interconnect reduction. Our scheme is based on the observation that not all FUs operate at all time. When idle, FUs can be reconfigured as pass-through logic for data transfer, reducing interconnect requirement. Our scheme not only reduces the overall length of global interconnects but also minimizes the power overhead without introducing any timing violations. Lastly, we present a register binding algorithm with the objective of register minimization. We have observed that not all pipelined FUs are operating at all time. Idle pipelined FUs can be used to store data temporarily, reducing stand-alone registers.

Exploration of High-level Synthesis Techniques to Improve Computational Intensive VLSI Designs

by
Taemin Kim

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2009

APPROVED BY:

Dr. Xun Liu Committee Chair	Dr. Eric Rotenberg
Dr. W. Rhett Davis	Dr. James M. Tuck

DEDICATION

To my beloved wife **Yoon Jung Lee**

and

our lovely daughter **Bohyun Kim**

BIOGRAPHY

Taemin Kim received BS and MS degrees in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST), Korea in 1998 and 2000, respectively. He also worked as Staff Engineer in GCT Semiconductor in Seoul, Korea, where he gained extensive experience on processor/SoC/ASIC design. Since 2005, he has been a Ph.D candidate in the department of Electrical and Computer Engineering. During his Ph.D study, he spent three summer and a spring semesters as an intern in Qualcomm, Intel and NEC Laboratories. His research interests include VLSI CAD, Reconfigurable Computing, High Speed and Low Power VLSI design, and Optimization. He is currently working on interconnect optimization in high level synthesis.

ACKNOWLEDGMENTS

First of all, I would like to express the deepest gratitude to God. He is indeed my father.

I want to thank my adviser Dr.Xun Liu. His advice for my research was invaluable. He always tried to give insight and teach me so that I can be a strong and independent researcher. I believe that his advice and discussion with him always motivated me to come up with new research ideas. Without his constant support and guidance, my research would not have been successful, and moreover, this thesis would not have been possible.

I am also grateful to my Ph.D committee members, Dr. Eric Rotenberg, Dr. W. Rhett Davis and Dr. James M. Tuck for their insightful comments and invaluable suggestion.

I am thankful to all of my seniors in faith, Deacon Youngsoo ”Richard” Kim, Deacon Jihye Kim, Deacon Jisoo Park, Deacon Sung Eun Yoo and Reverend Jung Woo Chong. Their unconditional love for Korean students and their family really touched my mind. The fellowship with them was also such a great spiritual experience which I cannot forget.

I am greatly indebted to my parents and parents-in-law for their great love and support. Regardless of my situation, they always encouraged me with confidence, which, needless to say, makes me comfortable and stand firm.

Lastly, but not least, to my wife Yoon Jung, and our daughter Bohyun, all I can say is that I am so thankful for your great sacrifice, patience and deep love for me. I can hardly express how grateful I am for you. Your love has always upheld me. I love you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

1.1.1 Global Interconnect

With the development of semiconductor fabrication technologies, the feature size of transistors has kept decreasing at exponential rates. Thus, delay and power characteristics of a logic gate improve due to the fact that the junction and gate capacitances of a MOSFET are scaled down. On the other hand, chip dimensions have increased continuously, since the VLSI system complexity, measured by the total number of devices on a chip, has become enormous [1, 2]. Thus, the lengths and the number of global interconnects increase. Global interconnects severely limit the further improvement of three design metrics: performance, power consumption and chip reliability.

While the delay of a logic gate improves as transistors are scaled down, that of global interconnects does not improve. The delay of an inverter can be estimated by $T = 0.69 \times R_{inv}C_j$, where R_{inv} is average value of on-resistance and C_j represents the intrinsic capacitances of an inverter. As feature size of a transistor is scaled down, C_j is also scaled down whereas R_{inv} remains same. As a result, the delay of an inverter decreases. When the inverter drives a long interconnect ended with another inverter as shown in Figure

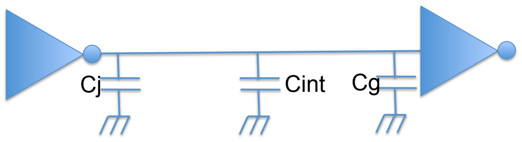

Figure 1.1: Interconnect between two inverters

1.1, the propagation delay is calculated by $T = 0.69 \times R_{inv}(C_j + C_g + C_{int})$, where C_{int} is interconnect capacitance and C_g is input capacitance of the receiving inverter. Since interconnect length increases due to increasing chip dimension, C_{int} is much larger than $C_j + C_g$ and hence, dominates the propagation delay. Therefore, optimization of global interconnects is crucial to achieve delay improvement from the technology advancement.

The power consumption of a chip is composed of dynamic and static parts as indicated in $P_{total} = P_{dyn} + P_{stat}$. The global interconnects contribute to the dynamic power consumption which is substantial portion of total power consumption. As shown in Equation 1.1, dynamic power consumption is a function of load capacitances of driving gates (C), supply voltage (V), switching activity (α) and operating frequency (f). The capacitance of global interconnects gets bigger as technology is scaled down and chip dimensions increase. Therefore, capacitances of global interconnects contribute to the dynamic power consumption substantially. As Magen and et al. indicated in [3], interconnect switching contributes to 50% of total power consumption of Intel microprocessors. Interconnects contribute to static power, too. In particular, since the delay of global interconnect is a function of the square of its length, repeaters are usually inserted to reduce the interconnect delay. Due to the increasing number of global interconnects, the number of repeaters inserted increases, which results in increase of static power consumption. Therefore, the global interconnect complexity should be optimized for reduction of power consumption.

$$P_{dyn} = 0.5\alpha C_l V^2 f \tag{1.1}$$

Global interconnects also affect chip reliability. The number of interconnects increases due to increasing number of functionalities in a chip. In addition, in order to minimize the cross-talk effect among interconnects, the minimum spacing among them cannot be reduced substantially. One potential solution to route all interconnects without increasing chip area is to increase the number of metal layers. However, the number of metal layers affects the thermal characteristic of a chip [4]. Since insulator material with poor thermal conductivity is filled between adjacent metal layers, the temperature of a chip will increase due to a large number of metal layers for interconnections, which affects the reliability and cooling cost of a chip [4]. Furthermore, the more metal layers are stacked up, the more vias are required. The combination of increased number of vias and scaled via dimension exacerbates the chip reliability.

1.1.2 Register

Registers are another factor we have to consider for the improvement of circuit quality. Registers are essential building blocks of synchronous circuits. They are used to improve throughput by pipelining computational paths. Moreover, because registers can isolate combinational circuit paths, designs based on registers are easy to verify and test. However, when a large number of registers are used, their cost can offset the benefit. Specifically, synchronous designs are sensitive to clock skews caused by different locations of registers placed across the entire chip. Clock skews affect the operating clock frequency and correct functionality of VLSI systems. They should be minimized to achieve high performance and race free circuits [5]. Furthermore, power consumption of registers is substantial because of the high switching activity and high load capacitance of clock signals. Registers and register related components such as clock buffers and clock network interconnects can consume about 15%−45% of total chip power [6, 7, 3]. In addition, large amount of registers increase the chip area, lower fabrication yield, and raise chip cost. As a result, register reduction is critical in improving chip performance and efficiency.

1.1.3 Optimization in High Level Synthesis

Hardware architectures are determined during high level synthesis. In this dissertation, we consider high level synthesis the architectural definition stage of a design flow. More specifically, the number of functional units and registers, interconnect topology and system latency are determined at this stage. Hardware architectures affect the quality of result (QoR) of the circuits implemented at the later stages of design such as logic synthesis, placement and routing. Optimization of global interconnects and registers at architectural definition stage is important and effective to achieve a good QoR of the final implementation. We can explore many different interconnect topologies during high level synthesis, resulting in the optimized interconnect architecture. In addition, the number of registers and how intermediate values are stored in the registers affect chip area, clock skew, power consumption and length of global interconnects. Therefore, optimization of registers at architectural definition stage has a big impact on the quality of final implementation of circuits.

1.2 Contribution

This dissertation studies two global interconnect optimization algorithms and a register reduction algorithm in high level synthesis.

- We have proposed a simultaneous register and functional unit (FU) binding algorithm in high level synthesis [8]. Our algorithm targets the reduction of multiplexer inputs, shortening the total length of global interconnects. Specifically, our algorithm constructs a weighted and ordered compatibility graph. This graph captures interconnect reduction opportunities by considering flow dependencies, common primary inputs and common register inputs. Our algorithm searches for operations that form a long path in the compatibility graph and binds them to a single FU. In addition, operation variables generated by the FU are assigned to the same register. Therefore, our algorithm performs FU and register binding concurrently. We have implemented our algorithm within a MATLAB to Verilog conversion tool, and applied it to a suite of benchmark programs. Our experimental results have shown that the proposed scheme achieves 29.19%, 21.98% and 32.56% multiplexer input count reduction on average over the weighted bipartite matching algorithm, *k-cofamily* algorithm and left edge algorithm, respectively. To assess the impact on interconnect length reduction, we have generated circuit layouts from our Verilog description. It is shown that our approach delivers a 17.29% reduction in total wirelength of global interconnects with minor area overhead in comparison to the left edge algorithm.

- We have proposed an interconnect binding algorithm during high-level synthesis for global interconnect reduction. Our scheme is based on the observation that not all functional units (FUs) operate at all the time. When idle, FUs can be reconfigured as pass-through logic for data transfer, reducing interconnect requirement. Our algorithm formulates the interconnect reduction problem as a modified min-cost max-flow problem. It not only reduces the overall length of global interconnects but also minimizes the power overhead without introducing any timing violations. Experimental results show that, for a suite of digital processing benchmark circuits, our algorithm reduces global interconnects by 8.5% on the average in comparison to previously proposed schemes [9, 8]. It further lowers the overall design power by 4.8%.

- We have proposed a register binding algorithm in high level synthesis with the objective of register minimization [10]. Our main observation is that not all pipelined functional units are operating at all time. Idle pipelined functional units can be used to store data temporarily, reducing stand-alone registers. The proposed register binding scheme is applied to a suite of benchmark circuits. Experimental results demonstrate that register counts can be lowered by 44% on average in comparison to the left-edge algorithm [11], which is often considered as the optimal approach. Our optimized circuits are synthesized, placed, and routed using state-of-the-art industrial

EDA tools. The reduction of register leads to reduction of rising and falling clock skews by 33% and 30% on average, respectively. Our scheme achieves the reduction of register counts and clock skews with little degradation to total wirelength and power dissipation. The change of interconnect length ranges from 4% reduction to 10% increase. The variation of power ranges from 4% reduction to 20% increase.

The rest of the dissertation is organized as follows. Chapter 2 proposes the simultaneous functional unit and register binding algorithm for global interconnect optimization. Chapter 3 proposes our interconnect assignment algorithm to minimize the total wirelength using idle functional units as interconnect components. Chapter 4 presents our register minimization algorithm during high level synthesis using the internal registers of pipelined functional units for variable storage. Chapter 5 summarizes this dissertation.

Chapter 2

A Functional Unit and Register Binding Algorithm for Interconnect Reduction

2.1 Introduction

Optimization techniques at early stages in a design flow are often believed to be more effective than those at later stages [12, 13, 14, 15, 16, 17]. As a result, interconnect reduction schemes during the high level synthesis step are preferred. Unfortunately, efficient interconnect optimization techniques in high level synthesis are challenging to develop due to the lack of circuit geometric information, e.g., placement and routing, in the architecture description. Only operations and their dependencies are available. Consequently, metrics that track interconnects such as numbers of connections among functional units or multiplexer inputs are usually used as the objective function for minimization [9, 8, 18, 19, 20, 21, 22, 23].

In this chapter, we present a simultaneous register and functional unit binding algorithm that targets interconnect reduction. Similar to techniques in [19, 20, 9], we use the total input count of multiplexers to approximate the cost of interconnects. By minimizing the multiplexer inputs, our scheme reduces global interconnect complexity and total wirelength. We introduce three graph parameters that guide our functional unit and register binding procedure. They are flow dependency, common primary inputs, and common register inputs in the given data flow graph (DFG). In particular, flow dependency represents data transfer directions. The common primary inputs and common register inputs denote the resource sharing potentials. All three parameters can capture characteristics of the DFG and thus assist interconnect optimization.

Our scheme proceeds in five steps. Specifically, for a given DFG, compatibility graphs are generated at first, with edge weights calculated based on the three graph parameters. Then, our scheme groups operations into the minimal number of long paths that contain only single type of operations. The path weights are maximized. The function unit and register binding results are derived concurrently from the creation of long paths. Specifically, a single functional unit is assigned to the operations in each path, and output variables in each path are bound to the same register. Thirdly, multiple paths are further concatenated to form mega-paths of multi-type operations to increase register sharing. Fourthly, lifetimes of all variables in mega-paths are analyzed to find special variables not yet bound to registers. Primary input variables are also searched. The register binding is completed in the fifth step.

Our algorithm minimizes the total number of multiplexer inputs. It also reduces the numbers of registers and functional units. In particular, the minimization of path count results in the reduction of functional units and registers. The maximization of path weights leads to simplified interconnect configurations and therefore less multiplexer inputs. We have implemented our scheme into a software tool and applied it to a suite of benchmark designs. Experiments have shown that our scheme derives binding results with the fewest multiplexer input counts in comparison to previously proposed schemes. In particular, 29.19%, 21.98% and 32.56% reductions are achieved over weighted bipartite matching algorithm [18], *k-cofamily* algorithm [19] and left edge algorithm [11], respectively, with a small overhead of functional units and registers. Moreover, the multiplexer input count reduction shortens global interconnect length. For the circuit layouts generated from our binding results using commercial EDA tools, the total wirelength is reduced by 19.00%, 17.29% and 25.24% over weighted bipartite matching algorithm, *k-cofamily* algorithm and left edge algorithm, respectively.

The remainder of this chapter is organized as follows. Section 2.2 reviews previous research on interconnect reduction techniques in high level synthesis. Section 2.3 gives an example that motivates our work. Section 2.4 formulates the problem of functional unit and register binding for interconnect reduction. Our proposed algorithm is described in details in Section 2.5. Section 2.6 provides the experimental results. Section 2.7 summarizes this chapter.

2.2 Interconnect Reduction in High Level Synthesis

For the past two decades, a plenty of techniques have been proposed for interconnect optimization in high level synthesis. Most of these techniques can be categorized into three groups based on how global interconnects are estimated. Specifically, in the first group, the interconnect complexity is represented by the data communication among various operation clusters, which will be bound to different computational units[24, 25, 26]. The interconnect reduction problem is recast as a min-cut graph partitioning problem. The design objective is to reduce data transfers among these computational units.

The second group of schemes improve interconnect estimation accuracy by incorporating physical design techniques, e.g., floorplanning or placement, into high level synthesis [27, 28, 29, 22, 21, 30, 31]. Since these schemes compute location information of circuit blocks, they are able to derive interconnect parameters such as total length and congestion, and conduct interconnect optimizations. To confine the runtimes in a reasonable range, only quick and sub-optimal physical design methods are applied, however. As a result, there may exist large discrepancy between the estimates and actual interconnect measurement that degrades the effectiveness of these schemes.

The third group of interconnect reduction schemes in high level synthesis link interconnects to circuit components such as FUs, registers and multiplexers. They bind operations with common inputs to the same FU and/or variables with common drivers[1] to the same register. Consequently, the connection among registers and functional units is simplified, leading to interconnect reduction. For register binding, different register models are considered such as centralized register files [32, 20] and distributed registers [9, 33, 34, 35, 18, 19, 36, 23]. Register binding techniques are also extended to perform variable-memory binding in [37, 38, 39]. The majority of FU and register binding algorithms are developed independently. Namely, FU binding is conducted followed by register binding[9, 33, 34, 18, 19, 36]. Such a strategy does not considers the interdependency between two

[1]A driver of a variable is the FU what derives the variable.

binding results and thus may not capture all optimization potentials. Simultaneous FU and register binding techniques have been proposed in [35, 23]. In particular, [23] iterates the sequential register and functional unit binding. During each iteration, binding result from the previous iteration is used to adjust the assignment of operations and variables to FUs and registers, respectively, for interconnect optimization. In [35], FU and register binding is performed one c-step at a time. The binding results of earlier c-steps are used to guide the optimization at the current c-step. The interconnect cost is often approximated using multiplexer input count, which is easy to derive during binding procedure. Since multiplexer minimization is believed to be NP-complete [40], multiple heuristics are used, including linear programming [37, 38, 33, 34], edge coloring algorithm [32], network flow algorithm [35, 19, 23], matching algorithm [9, 18, 20] and simulated annealing [36].

Our proposed work belongs to the third category since it targets the minimization of multiplexer input count. The uniqueness of our scheme is that it searches for long paths in the compatibility graph of a given DFG and derives the FU and register binding results concurrently. Moreover, it identifies novel graphic properties of the DFG that capture the potential for interconnect reduction effectively.

2.3 Motivating Example

This section gives an example that motivates our work. Figure 3.1(a) shows a scheduled DFG in which operations are represented as vertices. For simplicity, our example contains only one type of operations, i.e. addition, although our algorithm can handle DFGs with different operation types. Each operation has been scheduled in a time slot, called c-step. Directed edges represent data flow dependencies. Namely, variables are generated by starting vertices and are sent to the ending vertices. The names of variables are given next to corresponding edges. Figure 2.1(b) shows a FU binding solution without interconnect consideration. The operations in the two shaded regions are bound to two adders, either of which can perform an addition in one c-step. Figure 2.1(c) presents the register binding result by left edge algorithm (LEA) and the corresponding interval graph which shows the lifetime of each variable in figure 3.1(a). This register binding result is derived after the FU binding in figure 2.1(b). In the interval graph, variables $v1$ and $v1'$ are bound to the same register because they are the same output of the top-left addition operation and considered as one variable in the LEA. So are $v2$ and $v2'$. Figure 2.1(d) is the netlist derived from figure 2.1(b) and 2.1(c). The number of multiplexer inputs is 20. The numbers of adders and registers are two and four, respectively.

The number of multiplexer inputs can be reduced. Figure 2.1(e) gives the interconnect oriented FU binding. Figure 2.1(f) shows the lifetimes of the variables in figure 2.1(e) and register binding. The operations in either shaded region are bound to the same FU. The variables computed by either FU are stored in the same register. As can be seen, the number of registers does not increase in comparison to that in figure 2.1(c). Figure 2.1(g) shows the synthesis result. The number of multiplexer inputs is nine. The numbers of adders and registers are two and four, respectively. The number of multiplexer inputs is reduced by 55% in comparison to that of figure 2.1(d) while the numbers of FUs and registers do not increase. This example reveals that it is possible to reduce multiplexer input count substantially without adding registers or FUs. In the rest of this paper, we will present our scheme to achieve such a good result automatically. In addition, we will show that the minimization of multiplexer input count can lead to global interconnect length reduction.

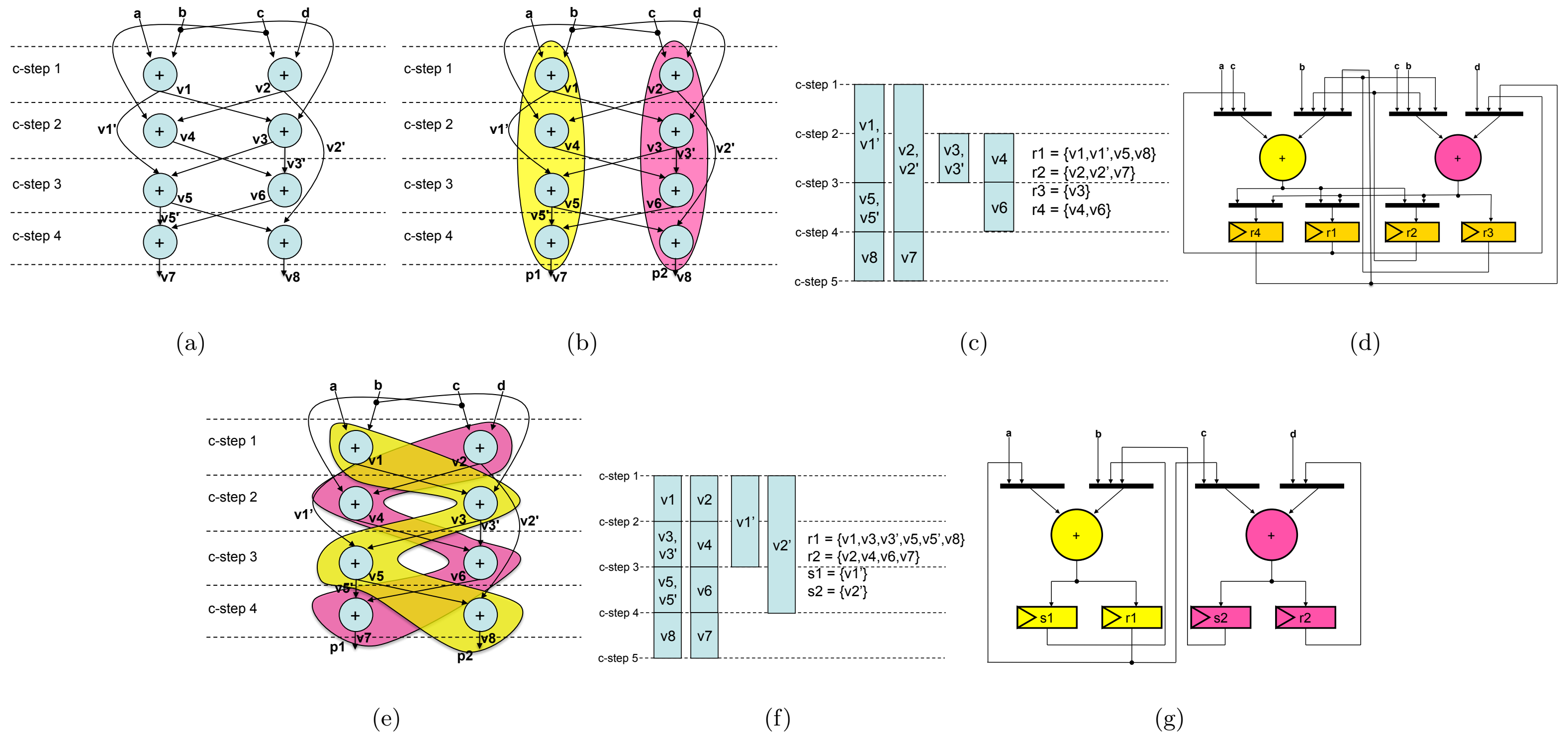

Figure 2.1: Motivating example (a) scheduled DFG (b) FU binding without interconnect consideration (c) register binding by left edge algorithm (d) synthesis result of part b and c (e) interconnect oriented FU binding (f) interconnect oriented register binding (g) synthesis result of part e and f

2.4 Preliminaries and Problem Formulation

In this section, we introduce some notations and use them to formulate the problem of register and FU binding for multiplexer input minimization.

2.4.1 Preliminaries

Given a scheduled DFG, two operations are compatible if they are executed in different c-steps. Compatible operations can be mapped to the same FU. The compatibility graph [16, 41] is often used to represent the compatibility among operations. In a conventional compatibility graph, operations are represented as vertices and compatible operations are connected by edges. In our scheme, we generate a modified compatibility graph, called weighted and ordered compatibility graph (WOCG), as follows.

Definition 1 *A weighted and ordered compatibility graph G(V,E) is a directed acyclic graph (DAG) with vertex set V and edge set E. Vertex set V is composed of vertices, each of which represents an operation in the DFG. Operations in V have the same operation type. A directed edge $u \rightarrow v$ is created between u and v if they are compatible, and u is scheduled earlier than v in the DFG. There is a weight w_{uv} on the edge $u \rightarrow v$. The value w_{uv} is calculated based on whether there is flow dependency between u and v and how many common inputs the operation u and v have.*

Definition 2 *A path in WOCG is a set of compatible operations $\{op_1, op_2, ..., op_i, ...op_j, ...op_n\}$ ordered based on their scheduled times. Namely, c-step(op_i) $<$ c-step(op_j) if $i < j$, where c-step(op) represents the time slot in which op is scheduled.*

Definition 3 *The lifetime of a path p, denoted as L(p), is from the start execution time of the first operation in p to the end execution time of the last operation in p.*

With the definition of path lifetime, we extend the compatibility concept to paths. The path compatibility graph (PCG) are derived from the union of all WOCGs created from a given DFG as follows.

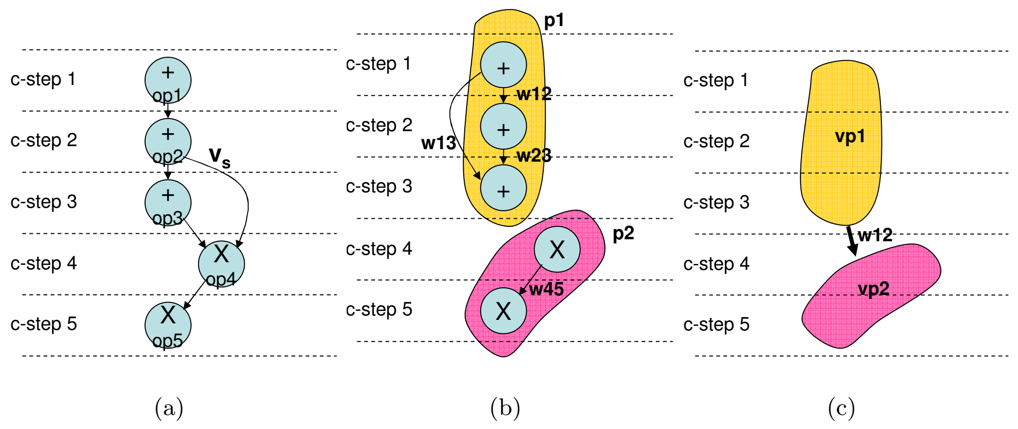

Figure 2.2: The procedure to construct a PCG (a) scheduled DFG (b) paths constructed (c) PCG constructed

Definition 4 *Path compatibility graph (PCG) $G_p(V_p,E_p)$ is a weighted and ordered compatibility graph for paths. The vertices of the graph represent paths found in all WOCGs. An edge is inserted to connect two vertices if their lifetimes do not overlap. Weights are assigned to edges to represent the data flow between the two corresponding paths and common inputs shared by them.*

Figure 2.2 shows the procedure to construct a PCG. Figure 2.2(a) is a scheduled DFG. Figure 2.2(b) shows paths $p1$ and $p2$ in the addition type WOCG and the multiplication type WOCG, respectively. The lifetime of $p1$ is from c-step 1 to c-step 3, and that of $p2$ is from c-step 4 to c-step 5. Since the lifetimes of $p1$ and $p2$ do not overlap, $p1$ and $p2$ are compatible. Figure 2.2(c) represents the PCG. The vertices $vp1$ and $vp2$ represent the paths $p1$ and $p2$, respectively. The edge from $vp1$ to $vp2$ shows the compatibility between $p1$ and $p2$. The weight $w12$ is assigned to the edge. The weight computation is explained in section 2.5.2.

Definition 5 *Let P be a path whose elements are $\{op_1, op_2, ..., op_n\}$. Suppose that operation op_i generates variable v_i. If any use-time of v_i is larger than c-step(op_{i+1}), v_i is a side variable.*

Figure 2.2(a) shows a *side variable* example. In particular, v_s is generated by $op2$, whose successor in path $p1$ is $op3$. Since v_s is used by $op4$ at c-step 4, which is larger than the c-step of $op3$, v_s is a side variable.

2.4.2 Formulation of Interconnect Oriented Binding Problem

The problem of register and FU binding for multiplexer input reduction can be formulated as follows.

PROBLEM: Given a scheduled DFG, find register and FU binding so that the total input count of multiplexers among registers and FUs is minimized. In addition, the numbers of FUs and registers should be minimized.

2.5 The Proposed Heuristic

2.5.1 An Overview

Our algorithm performs register and FU bindings in an integrated fashion. Given a scheduled DFG, it constructs a WOCG for each FU type first. It then finds paths with large total weights in all WOCGs. One distinct FU is assigned to the operations in each path. At the same time, a register is assigned to the variables derived from flow dependency relation between adjacent operations within the path. Compatible paths are merged to construct mega-paths to further reduce the number of registers. The lifetime of each mega-path is recorded. Our algorithm also finds *side variables* and *primary input variables.* Finally, register binding for mega-paths, side variables and primary input variables is performed.

2.5.2 The Proposed Path Based Binding Scheme

In this subsection, we discuss the details of our algorithm. Figure 2.3 shows the overall flow of our algorithm, which can be divided into five steps as follows.

Step 1 (*generation of WOCG for each operation type in a given DFG*): Step 1 is from line one to line four in figure 2.3. In this step, WOCGs are created for all FU types. Within a WOCG, each edge has a weight. The weight is calculated as in equation (2.1), by incorporating three factors, i.e. flow dependency, the number of common primary inputs and common input registers between operations.

$$W_{i,j} = \alpha * F_{i,j} + NIN_{i,j} + R_{i,j} + 1 \tag{2.1}$$

The i and j in equation (2.1) are indices for vertices v_i and v_j in the $WOCG$. $F_{i,j}$ is a Boolean variable which indicates whether there is flow dependency between v_i and v_j. $NIN_{i,j}$ is the number of common primary inputs of v_i and v_j. $R_{i,j}$ represents how many input registers are shared by v_i and v_j and α is an integer constant. In our implementation, we set $\alpha = 2$. If there is no flow dependency, common primary inputs, or common input registers, the weight is one.

Path_FU_REG_Bind : Path based binding algorithm

```
step1
1   foreach futype in Functional unit type
2     Create $WOCG_{futype}$ for futype
3     Put $WOCG_{futype}$ into set WOCG
4   endforeach
step2
5   foreach $WOCG_{futype}$ in WOCG
6     until There is no vertex and edge in $WOCG_{futype}$
7       p = Find longest path in $WOCG_{futype}$
8       Put p into path set $P_{futype}$
9       Update $WOCG_{futype}$
10    enduntil
11    Assign a functional unit to each path in P
12  endforeach
step3
13  Create path compatibility graph(PCG)
    for all paths in path sets $P_{futype}$
14  until There is no vertex and edge in PCG
15    p = Find longest path in PCG
16    Record start and end time of p
17    Put p into global path set $P_g$
18    Update PCG
19  enduntil
step4
20  foreach $p_i$ in $P_g$
21    Find side variables in $p_i$
22    Record start and end times of the side variables
23  endforeach
24  Find input variables in a given data flow graph(DFG)
step5
25  Register binding for input, side and path variables
```

Figure 2.3: The proposed algorithm

The inclusion of $F_{i,j}$ in the computation of $W_{i,j}$ is due to our simultaneous FU and register binding strategy. Specifically, our scheme binds operations along a path to the same FU and the corresponding operation outputs to the same register. If there is flow dependency between one pair of adjacent operations in the path, interconnects are needed to link the register output to the FU input. Such interconnects are sufficient and no more interconnects are needed if additional adjacent operations have flow dependency. Thus, the more consecutive operations with flow dependency are in a path, the more interconnect sharing is achieved, resulting in interconnect complexity reduction. Moreover, since the output registers are directly connected to the FU and highly probably placed near the FU, the interconnect length between the FU and output registers is short.

Figure 2.4 shows how flow dependency in a path affects the minimization of multiplexer input count. For simplicity, the primary inputs of the DFG is not considered. Figure 2.4(a) demonstrates the FU and register binding without consideration of flow dependency. Figure 2.4(b) is the synthesis result. Variables $\{v1, v2, v3, v7, v8\}$ and $\{v6, v4, v5\}$ are bound to register $r1$ and $r2$, respectively. The number of multiplexer inputs is eight. Figure 2.4(c) presents the FU binding with the incorporation of flow dependency. The operations with adjacent dependency relations form two paths, $p1$ and $p2$, and are bound to a single adder. Figure 2.4(d) shows the corresponding synthesis result. Variables in path $p1$ and $p2$ are bound to register $r1$ and $r2$, respectively. No multiplexer is needed. Thus, binding with consideration of flow dependency is effective in multiplexer input reduction.

The number of common primary input $NIN_{i,j}$ of two operations provides useful information in guiding FU binding procedure. Intuitively, when several operations have the same primary input, if they are bound to the same FU, only the FU needs to be connected to the input. On the other hand, when these operations are bound to different FUs, all FUs must be connected to the input, resulting in the increase of interconnect complexity. Thus, $NIN_{i,j}$ is considered in the computation of $W_{i,j}$ in our scheme.

Figure 2.5 shows two different binding results for one WOCG, i.e. $\{p1, p1'\}$ in figure 2.5(a) and $\{p2, p2'\}$ in figure 2.5(c). All operations in $p1$, $p1'$, $p2$ and $p2'$ have dependency relation, respectively. For operations in path $p1$, primary input a and b are used by more than one operation. On the other hand, no primary input is shared by operations in $p2$. Consequently, the synthesis result for $\{p1,p1'\}$, shown in figure 2.5(b), has less number of multiplexer inputs than that for $\{p2,p2'\}$ as in figure 2.5(d).

When multiple operations get their inputs from the same register, binding these operations to the same FU can reduce interconnect complexity. As a result, similar to the common primary input $N_{i,j}$, our scheme also considers common register input $R_{i,j}$ in

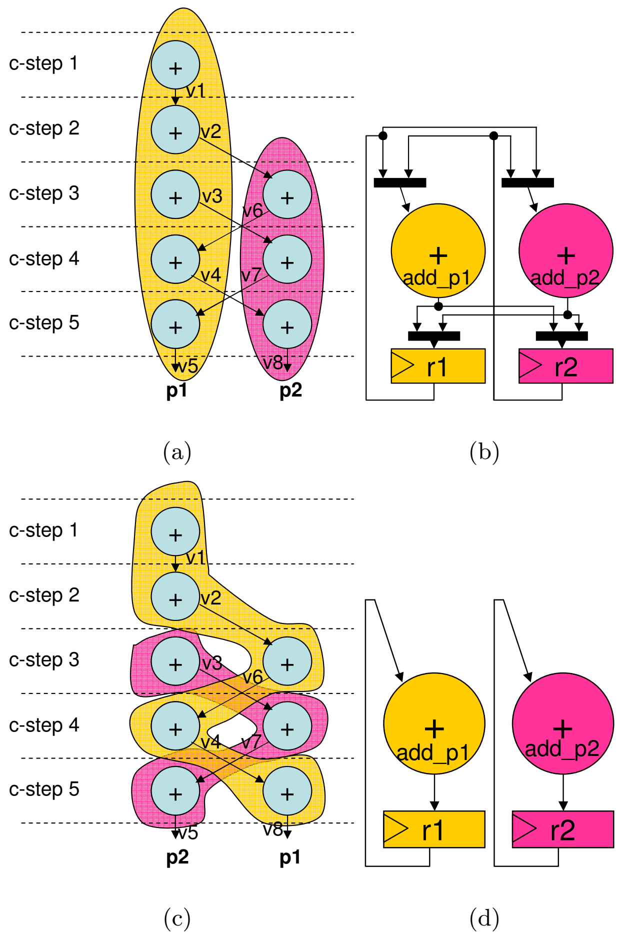

Figure 2.4: Illustration of how flow dependency affect binding result

calculating $W_{i,j}$. Unfortunately, different from $N_{i,j}$, which can be derived from the given DFG, $R_{i,j}$ is determined by the register binding result and is therefore unavailable before our optimization. In our scheme, $R_{i,j}$ is created during the binding procedure. Namely, as soon as a register is added to the register binding result, its impact on all $R_{i,j}$'s is recorded and the corresponding $W_{i,j}$'s are updated. As shown in the experimental result, the $R_{i,j}$'s based on the partial register binding are effective in reducing multiplexer inputs.

Figure 2.6 shows the WOCG constructed from figure 3.1(a). The directed edges represent the compatibility between operations and the scheduled order. The weight on

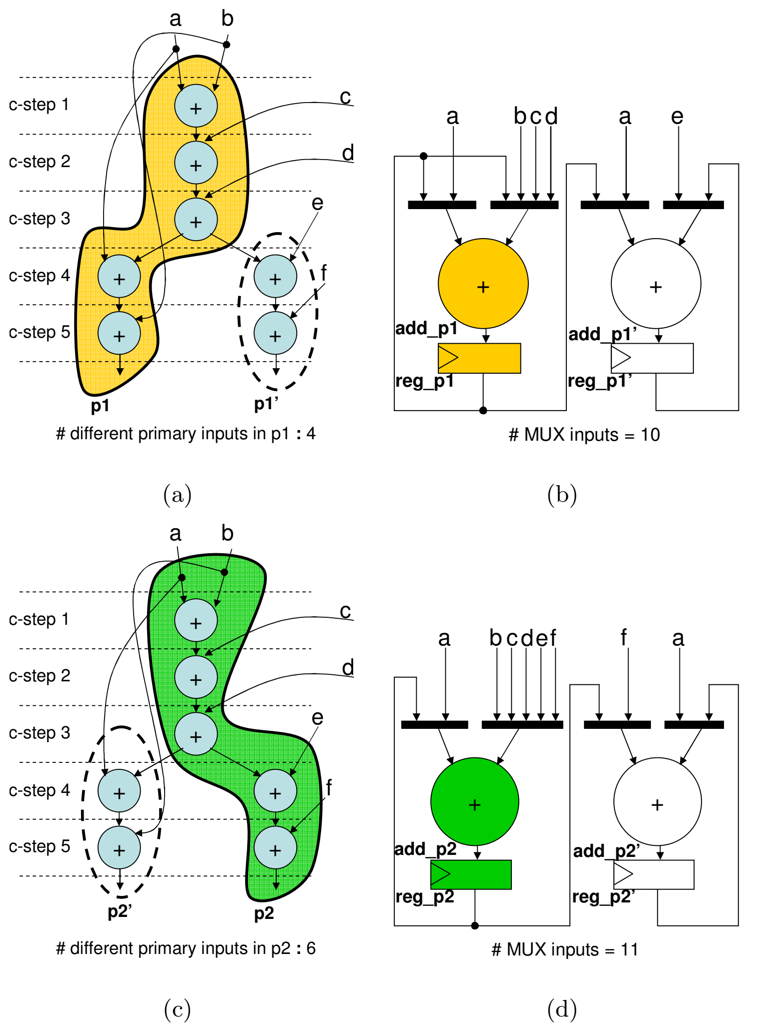

Figure 2.5: Illustration of how common inputs affect binding result

each edge is calculated by equation (2.1) at the beginning of our binding procedure. For example, the edge from $op1$ to $op4$ means that $op1$ and $op4$ are compatible, and $op1$ is scheduled earlier than $op4$. Since there is flow dependency between $op1$ to $op4$, $F_{1,4} = 1$. They have common primary input b, therefore $NIN_{1,4} = 1$. Since no register binding result is available, $R_{1,4}$ is zero. Thus, the weight $W_{1,4} = 2 * 1 + 1 + 0 + 1$.

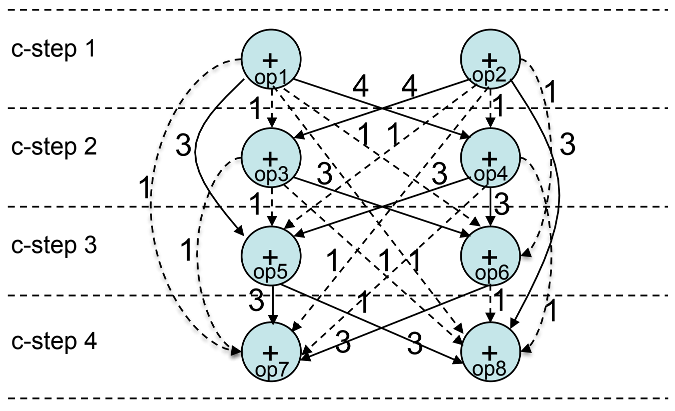

Figure 2.6: WOCG constructed from figure 3.1(a) before FU and register binding

Step 2 (*simultaneous FU and register binding step*): After generating WOCGs for all FU types, our algorithm partitions each WOCG into the minimum number of paths. The operations and variables in each path are bound to the same FU and register, respectively. To that end, our scheme utilizes a greedy heuristic. Specifically, for a given WOCG, the longest path algorithm [42] is applied first. After the longest path is found, edges and vertices in the path are removed from the WOCG. Edges that link vertices out of the path to those in the path are also removed. Edge weights of the modified WOCG are recalculated. In particular, since the register binding for the newly found path is derived, the common register input $R_{i,j}$ value in equation (2.1) might change for each edge in the modified WOCG. The search and removal of the longest path and weight update are repeated on the modified WOCG iteratively until all vertices and edges are removed. This pseudocode of this step is from line 6 to line 10 in figure 2.3.

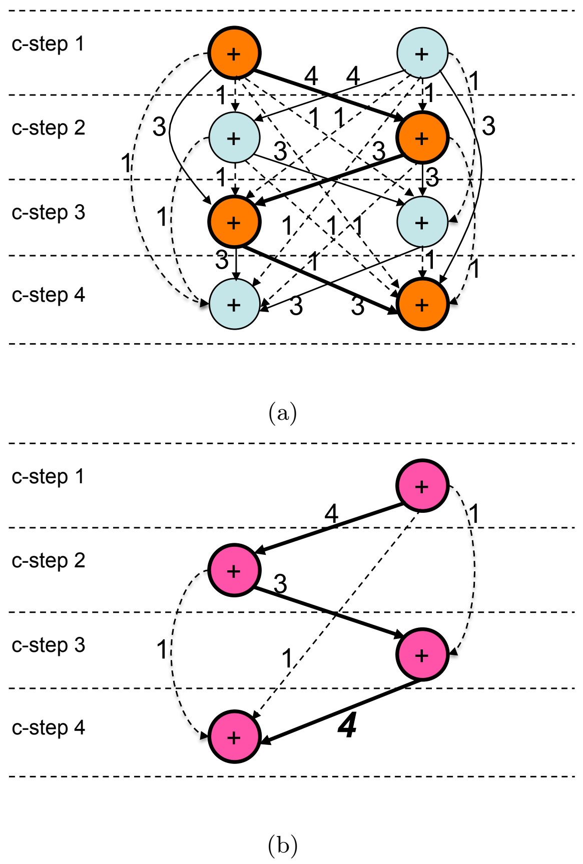

Figure 2.7: WOCG update process (a) The first longest path (b) updated WOCG and the second longest path

Figure 2.7 illustrates the FU and register binding procedure for the WOCG in figure 2.6. Specifically, the initial WOCG is shown in figure 2.7(a) with the longest path highlighted in bold. Figure 2.7(b) shows the updated WOCG after the longest path is removed. The longest path in the new WOCG is highlighted. Note that the weight of edge between the operation in c-step 3 and that in c-step 4 is changed from three to four. Such a change is due to the fact that both operations receive one input from the operations in the longest path in figure 2.7(a). Therefore, their common register input increases from 0 to 1 after the variables generated by the operations in that path are bound to a single register, resulting in the increase of edge weight.

Finding the longest path, i.e., path with the largest weight, helps in minimizing the total number of paths. Since the number of paths is equal to the number of FUs, path minimization reduces FUs. In addition, path weights are monotonic to flow dependency, common primary input and common register input. Maximization of path weight results in a FU and register binding solution with large values of the three metrics and thus substantial interconnect sharing. As a result, the interconnect complexity and the multiplexer inputs are reduced.

Step 3 (*finding path variables step*): Step 3 is from line 13 to line 19 of figure 2.3. In this step, our scheme reduces the register count by merging paths from different WOCGs. Specifically, we store the variables computed by the operations along a path in the same register. If the lifetimes of two paths do not overlap, one single register can be used for both paths, namely the paths are *merged*. Different from FU binding in which paths from different WOCGs cannot be combined together due to operation type difference, paths of all functional types can be bound to the same register.

Figure 2.8 illustrates the process of path merging. First, all WOCGs are combined as in figure 2.8(a). Only two operation types, addition and multiplication, are assumed. Either WOCG has been partitioned into several paths. The compatibility of all paths are then considered. Figure 2.8(b) shows the compatibility graph for paths in figure 2.8(a). Weights are assigned to edges of the path compatibility graph. The weights are calculated as flows:

$$W_{i,j} = F^{p}_{i,j} + 1 \ , \tag{2.2}$$

where i and j represent two paths. The Boolean variable $F^{p}_{i,j}$ indicates whether there is flow dependency between the paths. Equation (2.2) is similar to (2.1) except that the common primary input and common register input are not considered because common inputs between two different FUs do not contribute in the reduction of multiplexer inputs.

Our scheme applies the longest path algorithm to merge all paths. Figure 2.8(c) illustrates the merging result. The outputs of operations in a path will be assigned to a single register. The maximum number of drivers of the register is the number of operation types in the merged path. Figure 2.9 shows register binding result of a single merged path in figure 2.8(c). The register is driven by two drivers, i.e. the adder and multiplier. It is clear that although merging multiple paths reduces the register count, it can potentially increase the number of multiplexer inputs. In our current version of binding algorithm, high priority is given to register reduction. Therefore, the multiplexer input count derived is an upper bound that can be delivered by our scheme. Further multiplexer reduction is possible at the cost of increasing register count. After the path merging procedure, lifetimes of merged paths are recorded for the final register binding in step 5.

Step 4 (*finding side and input variables step*): The path based register binding in step 3 only introduces registers to store variables that are produced and consumed by consecutive operation pairs along paths. Additional registers are required for other types of variables. Figure 2.10 shows the example of register binding for a *side variable.* In figure 2.10(a), v_2' is a *side variable* because it has a consumer which is not an immediate successor of its generator within the compatibility path. Since our scheme uses a single register for $v1$, $v2$, $v3$ and $v4$, this register cannot hold $v2'$ until c-step 5. Thus, additional register is needed as in figure 2.10(b). In step 4 of our scheme, which is from line 20 to line 24 in figure 2.3, data dependency among paths and among nonadjacent operations within paths are analyzed to find side variables. In addition, registers for primary input variables are also searched to identify any need for registers. Constants are not stored in registers. The start and end use-time are recorded for the primary input and side variables for final register binding.

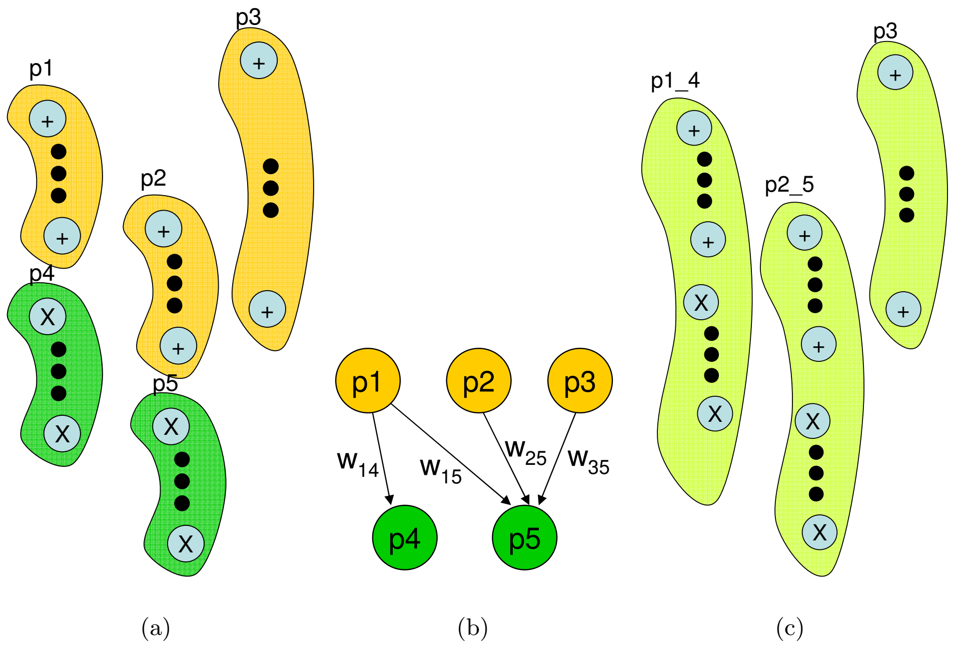

Figure 2.8: Construction of PCG (a) paths in a given DFG (b) path compatibility graph from (a) (c) merged paths

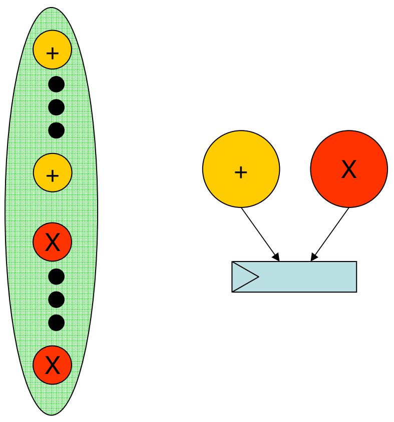

Figure 2.9: Functional unit and register binding for a merged path

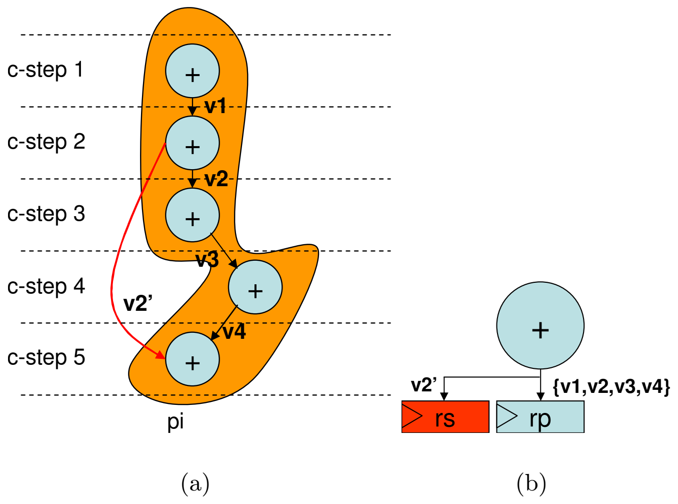

Figure 2.10: Side variable binding (a) path having a side variable v_2' (b) binding result for (a)

Step 5 (*final register binding step*): Step 5 is at line 25 in figure 2.3. The lifetimes of all variables from the previous steps are examined, including variables produced and consumed by adjacent operations with paths, side variables, and primary input variables. If the lifetimes of multiple variables do not overlap, registers assigned to these variables during step 2, 3, and 4 are merged by the left edge algorithm [11].

2.6 Experimental Results

We have developed a software program in C++ based on our algorithm. We integrated the program into our high level synthesis tool which converts a MATLAB program to a Verilog RTL description. We have applied our tool to a suite of benchmark programs from [43, 44, 45], which are data-oriented programs common in digital signal processing (DSP) applications. Since the benchmarks are coded in C and VHDL originally, we converted the C and VHDL codes to MATLAB descriptions. The scheduling of DFGs was performed by the Force Directed Scheduling algorithm [46].

We compared our simultaneous register and FU binding algorithm to previously proposed binding algorithms in the literature. Specifically, we used the weighted bipartite matching algorithm [9] for interconnect oriented FU binding and then used three different register binding algorithms i.e., weighted bipartite matching algorithm [18], LEA [11], and k-cofamily algorithm [19], for multiplexer input reduction. Since LEA guarantees the minimum number of registers if there is no control dependency in a given DFG, we compared the register counts of the results derived by LEA and our algorithm. In addition to the number of registers, we compared the number of FUs of the results generated by the weighted bipartite matching algorithm and ours.

Table 2.1: The number of multiplexer inputs

Benchmarks	LEA[11]	bipartite[18]	k-cofamily[19]	ours	LEA	bipartite	k-cofamily
aircraft	4178	4298	3393	2391	42.77	44.37	29.53
chem	767	753	612	420	45.24	44.22	31.37
dir	406	340	288	172	57.64	49.41	40.28
feig_dct	1272	1279	1243	1010	20.60	21.03	18.74
honda	223	212	192	120	46.19	43.40	37.50
mcm	272	278	257	230	15.44	17.27	10.51
pr	107	103	97	92	14.02	10.68	5.15
u5ml	1008	1009	849	590	41.47	41.53	30.51
wang	130	132	120	110	15.38	16.67	8.33
arai	94	90	83	77	18.09	14.44	7.23
lee	132	119	105	96	27.27	19.33	8.57
diffeq	28	23	26	20	28.57	13.04	23.08
fir11	60	52	27	23	61.67	55.77	14.81
cftmdl	191	198	183	138	27.75	30.30	24.59
cftb1st	514	482	497	377	26.65	21.78	24.14
fft	45	51	41	48	-6.67	5.88	-17.07
idct	285	276	234	169	40.70	38.77	27.78
matmul	278	276	264	207	25.54	25.00	21.59
wavelet	131	136	121	60	54.20	55.88	50.41
jacob	226	191	201	160	29.20	16.23	20.40
chendct	291	286	257	187	35.74	34.62	27.24
chenidct	307	303	283	237	22.80	21.78	16.25
kalman	39	36	36	23	41.03	36.11	36.11
lowpass	267	215	191	133	50.19	38.14	30.37
AVG					**32.56**	**29.19**	**21.98**

Table 2.1 shows the multiplexer input counts of binding results derived by LEA, weighted bipartite matching algorithm, k-cofamily algorithm and our algorithm. Column 1 lists the benchmark names. Columns 2 to 5 give the numbers of multiplexer inputs. Columns 6 to 8 show the multiplexer input reduction in percentage. As the table shows, our algorithm can reduce the multiplexer input by 32.56%, 29.19% and 21.98% on average in comparison to LEA, weighted bipartite matching algorithm and *k-cofamily* algorithm, respectively.

Table 2.2 shows the register count comparison between the optimal algorithm, i.e. LEA, and our algorithm. The values in the second and third column are the register counts. The fourth column shows the register count increase of our scheme in percentage. For half of the benchmarks, our algorithm produce the same results as LEA. On average, our algorithm increases the register count slightly by 6.97%. Such an increase does not affect the total layout area substantially since registers are small in size.

Table 2.3 compares the numbers of FUs. Since our benchmarks only contain additions and multiplications, N_+ and N_* are used to represent adder counts and multiplier counts, respectively. Columns 2 to 5 list the absolute FU counts whereas columns 6 to 7 give the increase percentages of our scheme over the weighted bipartite matching algorithm. As table 2.3 shows, the average increases of the numbers of adders and multipliers are 6.38% and 4.34%, respectively.

In order to verify that reducing the number of multiplexer inputs leads to global interconnect minimization, we generated the layout of all benchmark designs using Cadence SOC Encounter^{TM}, a widely used commercial EDA tool, and measured the actual total wirelength. Specifically, the binding results from both our scheme and previous approaches in comparison were first converted into Verilog RTL description. The same place and routing flow was applied to all the RTL designs. The FUs were predesigned as hard macro-cells. The timing target of each benchmark design was kept constant. After circuit layouts were created, we used SOC Encounter^{TM} to report the total wirelength of all

Table 2.2: The number of registers of left edge algorithm and our algorithm

Benchmarks	LEA[11]	ours	Increase
aircraft	159	159	0.00
chem	48	48	0.00
dir	72	72	0.00
feig_dct	114	132	16.79
honda	24	24	0.00
mcm	38	40	5.26
pr	18	20	11.11
u5ml	60	60	0.00
wang	20	21	5.00
arai	14	19	35.71
lee	17	20	17.65
diffeq	7	7	0.00
fir11	11	11	0.00
cftmdl	34	41	20.59
cftb1st	52	58	11.54
fft	16	16	0.00
idct	39	39	0.00
matmul	48	48	0.00
wavelet	25	25	0.00
jacob	30	30	0.00
chendct	33	39	18.18
chenidct	39	40	2.56
kalman	8	9	12.5
lowpass	44	49	11.36
AVG			**6.97**

Table 2.3: The number of multipliers and adders of left edge algorithm and our algorithm

	bipartite		ours		Increase	
Benchmarks	N_+	N_*	N_+	N_*	N_+	N_*
aircraft	38	40	38	40	0.00	0.00
chem	15	16	15	16	0.00	0.00
dir	8	7	8	7	0.00	0.00
feig_dct	37	12	40	12	8.11	0.00
honda	7	6	7	7	0.00	16.67
mcm	12	9	13	9	8.33	0.00
pr	5	8	6	9	20.00	12.50
u5ml	16	17	16	17	0.00	0.00
wang	5	8	5	8	0.00	0.00
arai	6	3	6	3	0.00	0.00
lee	8	4	8	4	0.00	0.00
diffeq	2	2	2	3	0.00	50.00
fir11	1	2	1	2	0.00	0.00
cftmdl	15	16	16	16	6.67	0.00
cftb1st	12	6	14	6	16.67	0.00
fft	4	4	5	5	25.00	25.00
idct	10	9	11	9	10.00	0.00
matmul	16	32	16	32	0.00	0.00
wavelet	8	16	8	16	0.00	0.00
jacob	10	8	10	8	0.00	0.00
chendct	12	8	13	8	8.33	0.00
chenidct	15	12	15	12	0.00	0.00
kalman	2	2	3	2	50.00	0.00
lowpass	6	8	6	8	0.00	0.00
AVG					**6.38**	**4.34**

interconnects beyond the macro-cells. Table 3.4 shows the wirelength comparison results. Our algorithm reduces total global interconnects by 25.24%, 19.00% and 17.29% over LEA, weighted bipartite matching algorithm and k-cofamily algorithm, respectively.

Table 2.4: The total wirelength for each algorithm

Benchmarks	LEA	bipartite	k-cofamily	ours	LEA	bipartite	k-cofamily
aircraft	2.97×10^7	$2.86times10^7$	2.56×10^7	1.26×10^7	57.68	56.02	50.81
chem	2.65×10^6	2.51×10^6	2.14×10^6	1.32×10^6	50.00	47.35	38.18
dir	1.62×10^6	1.23×10^6	1.19×10^6	8.40×10^5	48.02	31.55	29.22
feig_dct	9.42×10^6	8.40×10^6	8.61×10^6	8.00×10^6	15.06	4.78	7.09
honda	7.40×10^5	5.25×10^5	5.70×10^5	3.09×10^5	58.27	41.22	45.82
mcm	1.18×10^6	1.22×10^6	1.20×10^6	1.17×10^6	0.83	4.27	1.94
pr	4.34×10^5	4.19×10^5	3.98×10^5	3.83×10^5	11.81	8.65	3.83
u5ml	4.62×10^6	3.65×10^6	2.86×10^6	1.99×10^6	56.80	45.41	30.30
wang	5.29×10^5	5.02×10^5	5.17×10^5	4.72×10^5	10.67	5.85	8.66
arai	3.78×10^5	2.84×10^5	3.07×10^5	2.62×10^5	30.68	7.67	14.52
lee	3.33×10^5	3.51×10^5	3.83×10^5	3.24×10^5	2.85	7.71	15.31
diffeq	8.47×10^4	7.35×10^4	7.64×10^4	9.43×10^4	-11.43	-28.27	-23.47
fir11	1.05×10^5	8.02×10^4	6.92×10^4	6.46×10^4	38.38	19.43	6.61
cftmdl	8.74×10^5	1.15×10^6	8.79×10^5	7.52×10^5	13.97	34.55	14.49
cftb1st	1.50×10^6	1.64×10^6	1.70×10^6	1.46×10^6	3.05	11.28	14.23
fft	1.44×10^5	1.62×10^5	1.64×10^5	1.63×10^5	-13.15	-0.66	0.95
idct	1.01×10^6	1.04×10^6	8.36×10^5	6.40×10^5	36.88	38.57	23.50
matmul	2.07×10^6	1.62×10^6	1.96×10^6	1.66×10^6	19.68	-2.51	15.16
wavelet	8.41×10^5	6.42×10^5	7.39×10^5	5.12×10^5	39.09	20.20	30.67
jacob	1.02×10^6	7.58×10^5	1.04×10^6	6.19×10^5	39.05	18.28	40.56
chendct	1.16×10^6	1.08×10^6	9.00×10^5	8.53×10^5	26.67	21.38	5.28
chenidct	1.15×10^6	1.30×10^6	1.27×10^6	1.11×10^6	3.82	14.94	12.65
kalman	9.40×10^4	9.16×10^4	7.49×10^4	7.36×10^4	21.74	19.68	1.74
lowpass	8.99×10^5	6.89×10^5	6.73×10^5	4.91×10^5	45.31	28.73	27.00
AVG					**25.24**	**19.00**	**17.29**

2.7 Conclusion

In this chapter, we present a simultaneous FU and register binding algorithm for interconnect reduction. Our algorithm identifies long paths in the compatibility graph generated from a DFG, and conducts FU and register binding concurrently. Our scheme targets the minimization of multiplexer inputs by analyzing the flow dependency and common inputs of operations. Experimental results show that our algorithm reduces the number of multiplexer inputs by more than 20% on average in comparison to previously proposed algorithms [11, 19, 18]. Our scheme achieves a total wirelength reduction by 17.29% on average at the cost of slight FU and register increases.

Chapter 3

A Global Interconnect Reduction Technique during High Level Synthesis

3.1 Introduction

Global interconnects are used to convey data among system modules. Thus, if there exist other ways of data transfer, the global interconnects can be replaced and reduced. We have observed that some combinational functional units (FUs) can be reconfigured as pass-through logic. In addition, because the computational need of a system varies with respect to time, not all FUs are active at all the time. When a FU is idle, it can be used for data delivery. For instance, an adder with one input set to 0 can pass data from the other input to the output port.

In this chapter, we propose a global interconnect reduction algorithm during the interconnect binding step of high level synthesis (HLS). Our algorithm identifies idle FUs and selects some of them for data communication. The assignment of FUs to data transfers is conducted judiciously to minimize the total length of global interconnects. In addition, the power overhead is carefully managed. Our algorithm also guarantees not to introduce any timing violations.

The proposed algorithm takes a scheduled data flow graph (DFG) and the FU and register binding result as its inputs. It outputs the interconnect binding solution with minimal total wirelength of global interconnects. Our scheme consists of two steps. First, it examines all variable transfers and idle FUs. A many-to-one mapping from idle FUs to variable transfers are derived. A set of FUs are selected as potential candidates to a variable transfer only when the selection leads to small power degradation and possible interconnect reduction. In the second step, our algorithm computes the interconnect binding results. Specifically, it builds a graph to model all circuit paths available to every data transfer at each time step in the DFG. Both interconnect length estimates and circuit delay estimates are incorporated in the graph. A modified min-cost max-flow solver is applied to the graph and calculates the best way to conduct each data transfer, either through global interconnects or idle FUs. Our algorithm is finished when such a procedure is repeated for all time steps.

We have applied our interconnect binding algorithm to a suite of benchmarks to demonstrate its effectiveness. For each benchmark design, we have created the final circuit layout using industrial EDA tools to accurately assess the impact of our method on total wirelength and power dissipation. Experimental data have shown that our scheme has delivered a 8.5% wirelength reduction on the average over previously proposed interconnect reduction schemes in HLS [9, 8]. The power degradation due to the usage of idle FUs is offset by the power savings due to interconnect reduction. The power consumption of all benchmarks under random input patterns is reduced by 4.8% on the average

The remainder of this chapter is organized as follows. Section 3.2 briefly discusses the related work in the literature. Section 3.3 introduces the problem of interconnect binding in HLS and gives an example to motivate our study. Section 3.4 describes our proposed heuristic in details. Section 3.5 provides the experimental results, followed by conclusion of this chapter in Section 3.6.

3.2 Related Works

Many researchers have addressed the problem of interconnect reduction in HLS since it is believed that optimization schemes applied in early design steps have large impacts on the final design. In this section, we can only present a brief and non-exhaustive discussion on the previous work as follows.

Interconnect reduction efforts have been made during both scheduling and resource binding in HLS. Specifically, the work in [46] optimized the number of buses by incorporating the transfer distribution graph into the force directed scheduling (FDS) procedure. Huang et al. proposed a bipartite weighted matching algorithm for both register and FU bindings that minimized the number of multiplexer (MUX) inputs, leading to low interconnect complexity [9]. Deming et al. modified the cost function of the bipartite-matching algorithm in [9] in register binding so that multiplexer inputs could be further reduced [18]. Deming et al. also formulated the register binding problem for MUX optimization as a k-$cofamily$ based register binding problem [19]. Cong et al. proposed a simultaneous register and FU binding algorithm for interconnect optimization [23]. The work in [8] also performed concurrent FU and register binding. It used data dependency and common operands among operations in DFGs to minimize multiplexer inputs. The works in [26, 24, 25] formulated the binding problem for interconnect minimization as a min-cut graph partitioning problem. They reduced interconnect length by assigning tightly coupled operations to the same computational modules. All the aforementioned schemes only used interconnects for data transfers, without considering non-operational FUs, however.

The utilization of redundant FUs for interconnect reduction has been investigated [27, 47, 48, 49, 50]. Weng and Parker [27] proposed a scheduling and binding algorithm which inserts redundant FUs to minimize interconnect length in the critical path. The work in [47, 48, 49] proposed binding algorithms that use redundant FUs to maximize interconnect sharing among data transfers in a CDFG. Jang and Pangrle [50] proposed a grid-based interconnect binding algorithm. They used idle FUs to minimize routing length among FUs and registers. They targeted the 1-dimensional layout of datapath only and their technique might increase the total wirelength.

3.3 Motivating Example

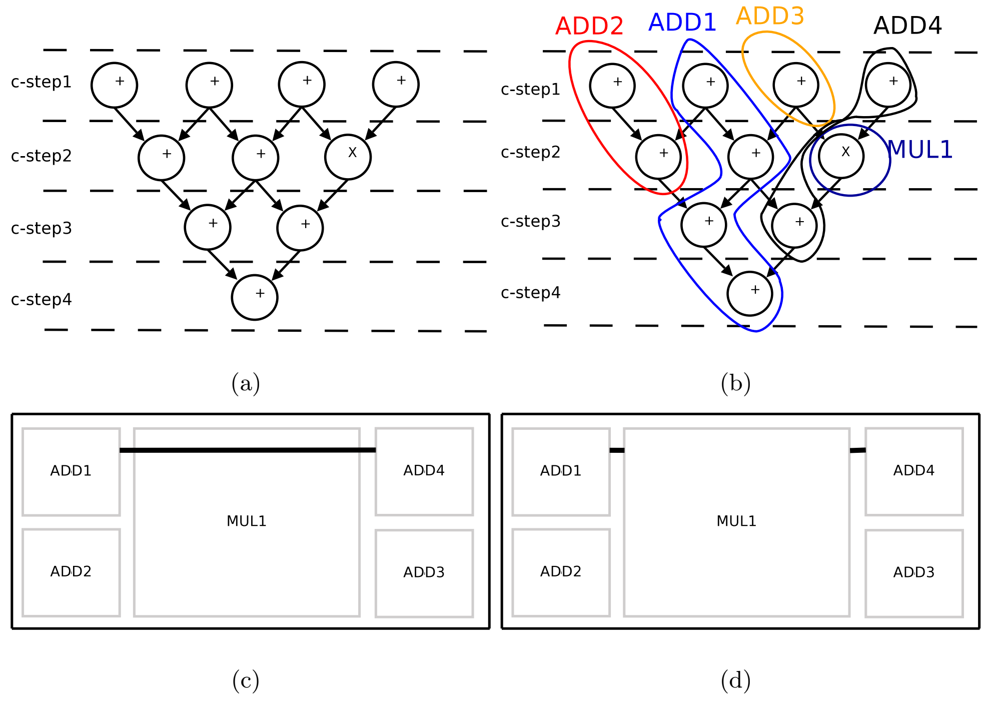

Figure 3.1: (a) Scheduled DFG (b) FU binding result (c) Interconnects for ADD1→ADD4 (d) Interconnects by using MUL1 for ADD1→ADD4

In this section, we use a simple example to describe the interconnect binding problem and motivate our work. Figure 3.1(a) shows a scheduled DFG in which operations are represented as vertices. The type of operation is labeled within each vertex. The '+' and 'x' signs represent addition and multiplication, respectively. The time slots when the operations are performed are shown as different c-steps. Edges represent data dependency among operations. Figure 3.1(b) shows the FU binding result. Operations in each closed region are bound to the same FU. The FU names are shown next to the regions. There are four adders and one multiplier in our example. When an edge crosses region boundary, a global data transfer is needed. For instance, the right addition operation in c-step 3 needs data from the middle operation in c-step 2. Since they are bound to different FUs, namely

ADD1 and ADD4, data must be sent from ADD1 to ADD4. A circuit path must be used at c-step 3 to complete the data transfer. Figure 3.1(c) shows the floorplan of our design example, in which a global interconnect are used for the data transfer from ADD1 to ADD4. Such an interconnect must span across the entire layout since ADD1 and ADD4 are placed at different end of the chip. Alternatively, since the multiplier MUL1 does not operate at c-step 3, it can be used to deliver the data as shown in Figure 3.1(d). Consequently, the global interconnect length is reduced substantially.

Although the concept of using idle FUs for global data transfers is straightforward, the implementation of such a strategy is difficult. Three challenges need to be resolved. First, additional global interconnects might be needed to utilize certain idle FUs. For our example in Figure 3.1, interconnects between ADD1 and MUL1 and interconnects between MUL1 and ADD2 must be added. These addition interconnects must not offset the reduction gain. To that end, accurate wirelength information on interconnects among all pairs of FUs is needed. Second, the signal propagation delays through idle FUs must be checked to ensure that no timing violation is introduced. Third, the power degradation of using idle FUs must be controlled. Particularly, if a FU is not used for several consecutive cycles, its inputs are usually set to constants to eliminate its internal switching activity [51]. However, if the FU is reconfigured and used for data transfer, its internal nodes will switch, resulting in power consumption. The extra power must be limited and preferably offset by the power savings due to global interconnect reduction. In the following section, we present our interconnect binding algorithm that utilizes idle FUs for global interconnect reduction. Our scheme considers all three above issues. It reduces interconnect length, maintains timing closure, and causes negligible power increase.

3.4 Proposed Heuristic

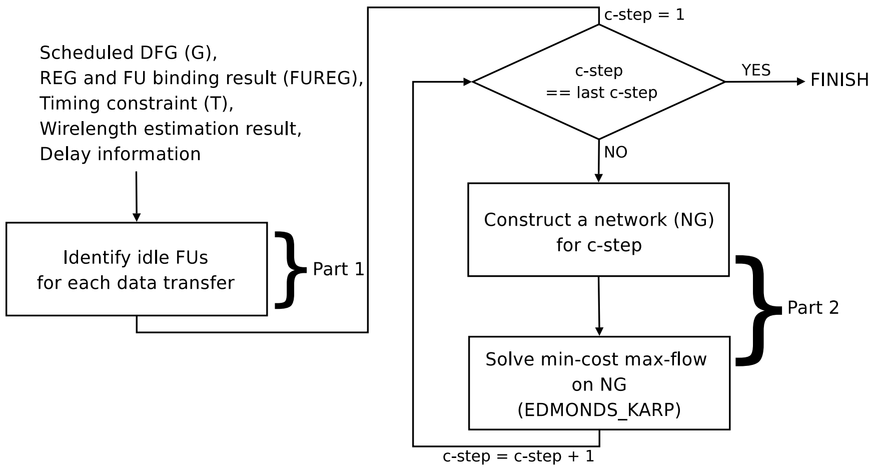

Figure 3.2: Flow chart of interconnect reduction algorithm

The figure 3.2 shows the overall flow of our interconnect reduction algorithm. Its input includes a scheduled DFG, the register and FU binding result, timing constraint, interconnect length estimates and circuit delay estimates. It computes the interconnect binding result, namely all interconnects needed among FUs and registers as well as how all data transfers are performed. Our algorithm consists of two parts. In the first part, it identifies candidate idle FUs that can potentially be used for each data transfer in the DFG. The second part is the core of our technique. It is made of a loop structure. Within each iteration of the loop, one c-step is selected. Our algorithm derives a circuit path for each data transfer of the c-step. Specifically, it constructs a graph in which registers and FUs are vertices. The interconnects are edges. The wirelength estimates and delay estimates are used to generate weights for the edges and vertices. Our algorithm solves a modified *min-cost max-flow problem* on the graph. The edges with non-zero flows become physical interconnects. Our algorithm is finished when all iterations are completed. In this following sections, the detailed description of our algorithm is presented with the design example in Section 2.3.

3.4.1 Identifying Idle Functional Units For Data Transfers

Three types of data transfers exist in a given design: register-to-FU transfers, FU-to-register transfers, and FU-to-FU transfers. For simplicity, we only consider register-to-FU transfers in this paper. Our algorithm can be extended straightforwardly to handle other types of data transfers as long as the locations of registers and FUs are given. We have re-drawn the scheduled and FU-bound DFG in Figure 3.1(b) as Figure 3.3. The FU binding result is represented by placing integers inside of vertices. For example, '+2' means ADD2. The shaded boxes denote the registers. Register names are located above the boxes. Registers labeled with the same name are the same physical register, in different time steps. Based on Figure 3.3, there are 9 types of register-to-FU transfers. They are represented as register-FU pairs and listed in Column 1 of Table 3.1 with the corresponding c-steps in Column 2. Note that one type of data transfer can occur at multiple time steps.

After the identification of data transfers, idle FUs are identified. Since the counts for various types of FUs are known for a given design, idle FUs can be found based on the FU-bound DFG. In Figure 3.3, the idle FUs are shown as shaded circles in braces. Idle FUs in the same c-step form a set.

Our algorithm then determines which idle FUs can be used for each type of data transfer. Although different idle FUs can be used for a single type of data transfer at different c-steps, in order to simplify the interconnect complexity, our scheme only allows a data transfer to use the same interconnects including idle FUs for all c-steps where it occurs. Consequently, a FU can be a candidate for a type of data transfer only if the FU is idle for all the c-steps when the data transfer occurs. For example, the data transfer r2$\rightarrow$+1 exists at c-steps 2, 3 and 4 in Figure 3.3. Only FU '+3' can be a candidate for the data transfer since it is idle for all three c-steps.

Algorithm 1 shows the pseudocode for our idle FU identification procedure. In Lines 1 and 2, all types of data transfers and idle FU sets are identified respectively for each c-step. The idle FU candidates for each data transfer type are then calculated in Lines 3–7

Algorithm 1 Identification of idle FUs for data transfers

Input: Scheduled DFG (G), FU and REG binding results

Output: Idle FU set for each data transfer in DFG

```
1: $DT$ = Identify all data transfer types in $G$
2: $IDLEFU$ = Identify idle FU sets for all c-steps
3: **foreach** data_transfer type $i$ in $DT$ **do**
4:     **foreach** c-step $k$ when $i$ occurs **do**
5:         idle_FU[i] $\bigcap=$ $IDLEFU$[k]
6:     **end foreach**
7: **end foreach**
```

by intersecting all idle FU sets at the c-steps when the transfer occurs. Column 3 of Table 3.1 lists the idle FU candidates for all data transfer types.

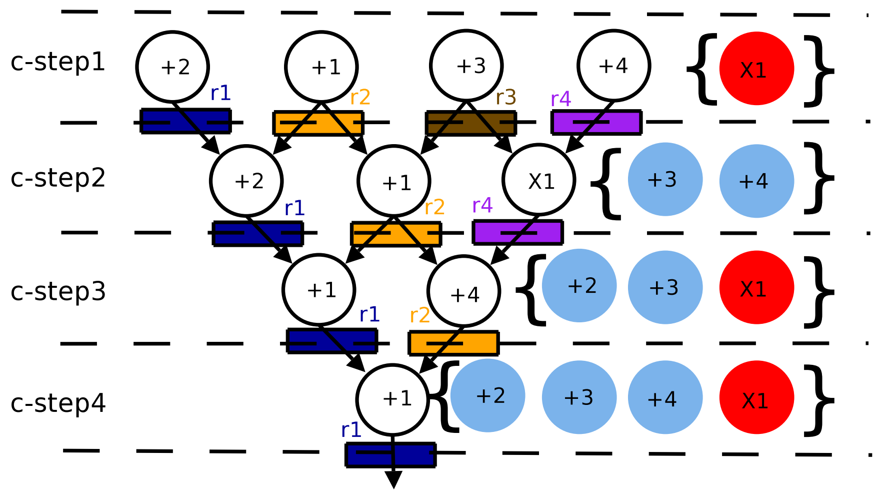

Figure 3.3: Idle Functional Units

Table 3.1: Idles FUs for data transfers

Data Transfer	c-step	Idle FU
r1 $\rightarrow$ +2	2	+3, +4
r1 $\rightarrow$ +1	3, 4	+2, +3, X1
r2 $\rightarrow$ +1	2, 3, 4	+3
r2 $\rightarrow$ +2	2	+3, +4
r2 $\rightarrow$ +4	3	+2, +3, X1
r3 $\rightarrow$ +1	2	+3, +4
r3 $\rightarrow$ X1	2	+3, +4
r4 $\rightarrow$ X1	2	+3, +4
r4 $\rightarrow$ +4	3	+2, +3, X1

3.4.2 Network Flow Formulation of Interconnect Binding Using Functional Units

After identifying idle FUs for each data transfer, we construct a directed graph $NG(V, E)$ to find interconnects and idle FUs for data transfers at each c-step, with the objective of minimizing the total wirelength. The $NG(V, E)$ contains every possible interconnects for data transfers in a particular c-step. Among them, minimum length interconnects are selected for each data transfer. The set V contains three types of vertices. For each transfer type which occurs in the c-step, two round vertices are added, representing the register and FU of the transfer, if they have not been added. In addition, one rectangular vertex is added for each idle FU candidates of the transfer. Two diamond vertices are also added, representing the primary source and target.

Figure 3.4(a) illustrates the graph construction using our example in Figure 3.3. The graph is for c-step 3. According to Table 3.1, there are 4 types of transfers in c-step 3, namely r1 $\rightarrow$ +1, r2 $\rightarrow$ +1, r2 $\rightarrow$ +4 and r4 $\rightarrow$ +4. Thus, five round vertices are in the graph, i.e., r1, r2, r4, +1 and +4. There are four sets of rectangular vertices. They are the candidate idle FUs for the four transfers. Note that if an idle FU is a candidate for multiple transfers, there are multiple copies of rectangular vertices. For example, 4 copies of '+3' are in Figure 3.4(a). Only rectangular vertices have vertex weights, which are equal to the propagation delay of the FUs when they are reconfigured as pass-through logic.

After the insertion of vertices, edges are added. There are six types of edges: edges from diamond source to round registers, edges from round registers to round FUs, edges from round FUs to the diamond target, edges from round registers to rectangular FUs, edges from rectangular FUs to round FUs, and edges among rectangular FUs of the same transfer. Each edge has four weights attached: interconnect length, interconnect delay, flow, and capacity. The interconnect length and interconnect delay are derived based on the inputs if both vertices of the edge are circuit components. If either vertex is a diamond, the interconnect length and interconnect delay are set to zero. Initially, flow of all of the edges is set to zero. The capacity of each edge from source diamond vertex is the number of data transfers which the associated register initiates. The capacity of each edge to the target diamond vertex is the number of data transfers which the associated FU receives. The capacity of all other edges are 1. In Figure 3.4(a), only the flow and capacity are shown. The interconnect length and interconnect delay are omitted due to the lack of space.

Algorithm 2 MODIFIED_EDMONDS_KARP algorithm

```
Input: Network ($NG$), Timing constraint ($T$)
Output: Minimum-cost Maximum-flow
1: $RNG$ = Construct Residual Network ($NG$)
2: while flows are less than capacities for outgoing edges of source do
3:   if MODIFIED_BELLMAN_FORD ($RNG$, $T$) == true then
4:     Update $NG$ and $RNG$
5:   else
6:     return false
7:   end if
8: end while
```

After constructing a network shown in figure 3.4(a), our algorithm assigns interconnects and idle FUs for the data transfers in the network by running MODIFIED-

EDMONDS-KARP algorithm, which finds flows with minimum cost under the given timing constraint.

Algorithm 2 shows the pseudocode of our MODIFIED-EDMONDS-KARP algorithm. Specifically, it first constructs the residual network in Line 1. It then applies a greedy strategy to find the flow paths from the source to the target using the loop in Lines 2–8. During each iteration of the loop, a flow path with the smallest total cost is derived in Line 3. The original EDMONDS_KARP algorithm finds such a flow path using the *Bellman Ford Shortest Path Algorithm* [42]. It therefore can only handle one edge cost, besides flow and capacity. In our problem, each edge has two costs, i.e., interconnect delay and interconnect length. The path to be computed should have the smallest total wirelength *and* satisfy the delay constraint, namely the total path delay must not exceed total timing budget T. To that end, we have modified the Bellman Ford Algorithm to calculate the *constrained shortest path*.

The Algorithm 3 shows overall procedure of the modified Bellman-Ford shortest path algorithm which takes timing constraint into consideration in addition to wirelength. The inputs of the algorithm are weighted directed graph (WDG) and timing constraint (T). In our problem, WDG is a residual network. The output is the shortest path in the residual graph, which means that the path is a flow with the minimum wirelength and a path delay less than or equal to T. The modified Bellman-Ford shortest path algorithm maintains the orginal structure of Bellman-Ford algorithm. The difference is that, instead of keeping a single cost at each vertex, a list of solution pairs is kept. Each solution pair has a delay and a wirelength. During the edge evaluation step in Lines 4–11, the new delay is first calculated by adding the corresponding wire delay and vertex v delay. If the new delay exceeds the timing constraints, no update will be performed. Otherwise, the new wirelength is derived in Line 7. The new solution pair is then inserted to the solution lists at vertex v in Line 8.

An efficient solution trimming strategy is adopted during the solution insertion procedure. In particular, it can be proved that the solution pair can be organized in a linear order. Within the linear order, the wirelengths of solution pairs are monotonically increasing and the delays of the solution pairs are monotonically decreasing. A new solution pair k will not be inserted if there exists an old solution pair i that

$$\text{i.wirelength} < \text{k.wirelength and i.delay} < \text{k.delay} \tag{3.1}$$

Similarly, the insertion of a new solution may result in the deletion of one or more old solutions. Therefore the number of solutions in the list remains limited. After the evaluation of edges is completed, the solution pair with the smallest wirelength is the constrained shortest path solution. The entire path can be derived using the backtracking procedure the same as that in the orginal Bellman-Ford algorithm. If there is a shortest path from source vertex to target vertex, it returns **true**. Otherwise, it returns **false**.

The modified Bellman-Ford algorithm is able to find the flow with the shortest wirelength under the given timing constraint. This flow is the solution for the data transfer originated by the round register vertex in this c-step. For example, Figure 3.4(b) shows a flow with bold solid lines from source to target. It indicates that the multiplier 'X1' should be used for the data transfer 'r2'→'+4' at c-step 3.

After finding a minimum-cost flow for a data transfer, our algorithm updates the graph so that any edges related to the data transfer and vertices of the selected idle FUs cannot be chosen by other data transfers. Specifically, all edges related to the data transfer have their capacity to zero. In addition, the idle FUs on the minimal flow are marked and will not be used in deriving other flows. Figure 3.4(c) shows the example of updated network. After the graph is updated (Line 4 of Algorithm 2), residual network is also updated as original EDMONDS-KARP algorithm.

After performing the MODIFIED-EDMONDS-KARP algorithm at a c-step k, our algorithm updates of idle FUs sets for data transfer types at other c-steps that have not been processed, i.e., c-step $k+1$ and etc. If an idle FU is used for a particular data transfer m at c-step k, it cannot be used for other data transfers in the c-steps where the data transfer m exists. Thus, the idle FU should be removed from the idle FU sets associated

Algorithm 3 MODIFIED_BELLMAN_FORD algorithm

Input: Weighted Directed Graph (WDG), Timing Constraint (T)

Output: Shortest path with satisfying given timing constraint

```
1: relaxed = **false**
2: **for** $i = 0$ to $size(vertices(WDG)) - 1$ **do**
3:    **foreach** edge $uv$ in edges($WDG$) **do**
4:       **foreach** solution pair $k$ at $u$ **do**
5:          Delay = k.delay + v.delay + uv.wire_delay
6:          **if** Delay $\leq T$ **then**
7:             WL = k.WL + WL(uv)
8:             UPDATE(v, WL, Delay)
9:          **end if**
10:      **end for**
11:   **end foreach**
12: **end for**
13: **if** a shortest path exists in $WDG$ **then**
14:    **return true**
15: **else**
16:    **return false**
17: **end if**
```

with other data transfers in those c-steps.

3.4.3 Algorithm Analysis

The following theorems ensure the correctness of our Algorithm. Specifically, the first theorem guarantees that all data transfers are assigned to a path for delivery. The second theorem guarantees that there is no combinational feedback loops in the resulting circuits. Note that false combinational feedback paths may still exists.

Theorem 1. Let n_d denote the number of data transfers in a network $NG(V, E)$. Let n_f denote the number of *min-cost* flows found in $NG(V, E)$. Then, $n_d = n_f$.

Proof Sketch: Edmonds-Karp algorithm finds the maximum flow, n_f. Based on the max-flow min-cut theorem [42], n_f is equal to min-cut. We next show that min-cut is equal to the data transfer count, n_d.

First, a cut separating the source from all other nodes has a cutsize n_d. Therefore, the min-cut is no more than n_d, i.e. $n_f \leq n_d$.

Second, given a graph, we can remove all idle FU nodes and edges connected to them. Cutsizes of all cuts and, therefore, min-cut of the resulting subgraph are n_d. The min-cut of the original graph is no less than that of the new graph, n_d, i.e. $n_f \geq n_d$.

Therefore, n_f is equal to n_d.

Theorem 2. In a network $NG(V, E)$, no flow has a cycle.

Proof Sketch: Since edge weights are always positive, if a flow has a cycle, we can always find another flow by removing the cycles. The new flow would have less cost. Therefore, since Edmonds-Karp algorithm always finds the minimum cost flow, no flow has a cycle.

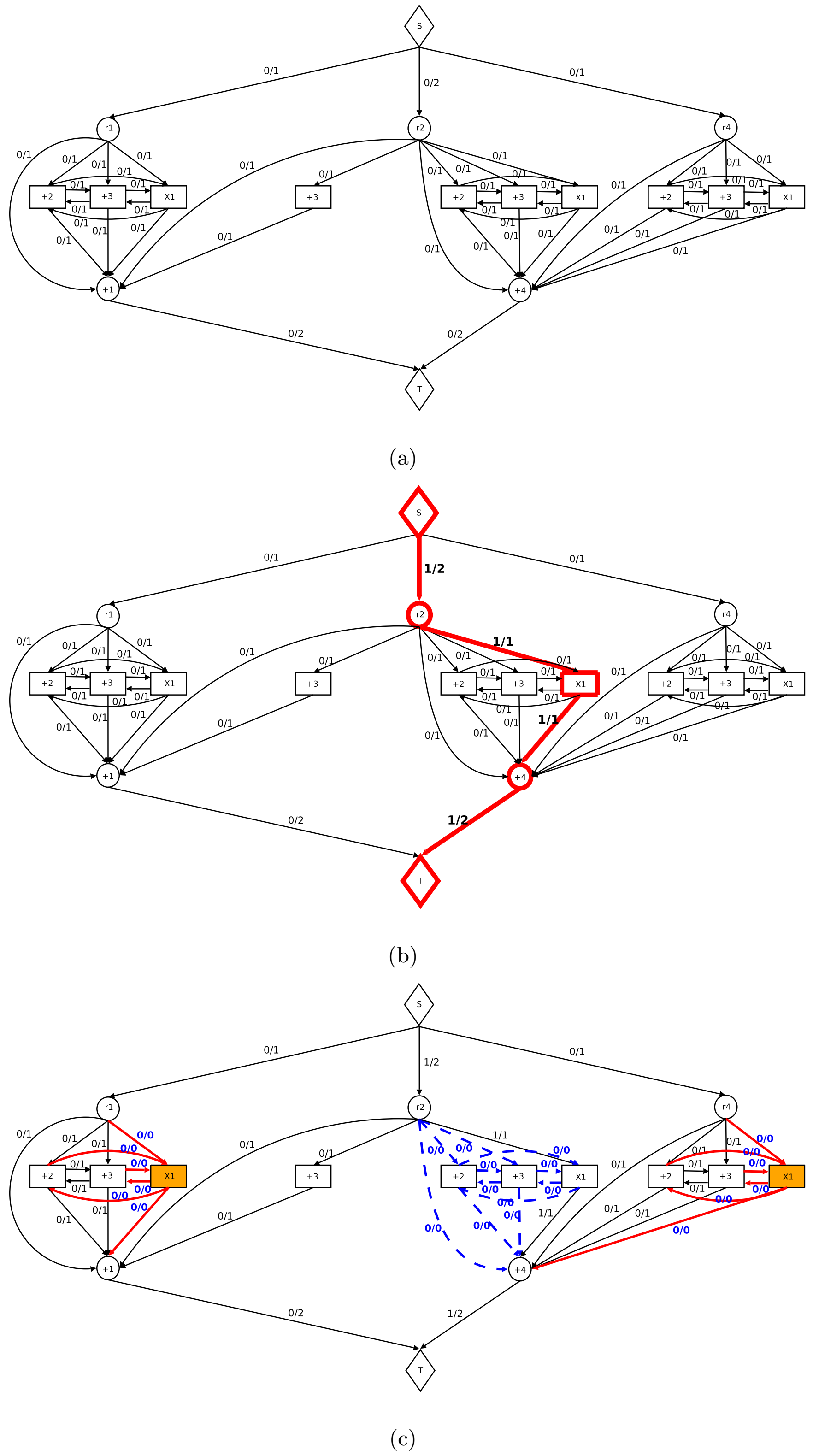

Figure 3.4: Process of interconnect binding (a) Constructed Network Flow (b) One of min-cost max-flow paths (c) Updated Network Flow

3.5 Experimental Setup and Results

In this section, we present our experimental results. Our data based on layout analysis and gate-level circuit simulations demonstrate the effectiveness of the proposed scheme. In comparison to previous interconnect reduction schemes in HLS, our algorithm reduces total interconnect length by 8.5% on the average. The worst power degradation is 5.1%. On the average, the design power is even reduced by 4.8%.

3.5.1 Experimental Setup

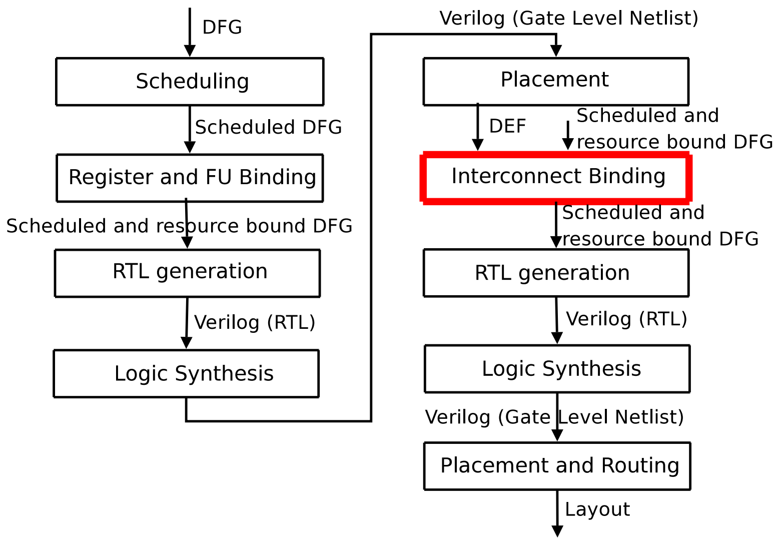

Figure 3.5: Experimental Flow

We have implemented our interconnect binding algorithm into a HLS software tool and tested it with the data intensive benchmarks in [43], which are common in the digital signal processing (DSP) field. All operations in these benchmarks are either addition or multiplication. Our proposed algorithm is applied during HLS step. However, in order to accurately estimate the real impact of our scheme on various circuit metrics, we have applied state-of-the-art EDA tools to generate the final layout for each benchmark. Figure 4.7 shows our design flow. Given a benchmark DFG, we perform the operation scheduling using the force directed scheduling algorithm, which is effective for resource usage. We then apply previous interconnect reduction schemes to complete the bind process. The goal is to produce the best result when idle FUs are not considered. We generate verilog RTL description from the scheduled and bound DFG. We use Synopsys Design Compiler^{TM} to perform circuit synthesis and generate the gate level netlists. We perform placement and routing using Cadence SoC Encounter^{TM} to get physical location information of registers and FUs.

With the placement information and the scheduled and bound DFG, we perform our interconnect binding denoted as the bold box in Figure 4.7 to generate the new RTL description. The same synthesis and physical design flow are next applied to produce the final layout.

To estimate power consumption of our circuits, we simulate their gate netlists with randomly generated inputs. The switching activity information of all nodes in the netlists are stored in the Switching Activity Interchange Format (SAIF). Design Compiler^{TM} is then used to derive power estimates using the SAIF files and the capacitance information from the layouts. Total wirelength is derived based on reports from SOC Encounter^{TM}.

3.5.2 Experimental Results

In order to assess whether there exist sufficient idle FUs, we examine the scheduled and FU-bound DFGs of all benchmarks. Tables 3.2 and 3.3 show how many idle adders and multipliers exist at each c-step on average. The first columns of the tables show benchmark names. The second columns show the number of available FUs. The third columns are the average numbers of idle FUs. The last columns present the percentages of idle FUs. As shown in the tables, 39.14% and 47.10% of adders and multipliers are not in operation at each c-step on average.

We compared our algorithm to compatibility path based register and FU binding algorithm [8] and weighted bipartite matching register binding algorithm [9]. Both algo-

Table 3.2: Average number of idle adders

Benchmarks	Required ADD	AVG IDLE ADD	RATIO (%)
chem	15	3.93	26.20
honda	7	3.47	49.57
dir	8	3.80	47.50
feig_dct	40	7.71	19.28
lee	8	4.78	59.75
mcm	13	5.00	38.46
u5ml	16	5.31	33.19
AVG			39.14

Table 3.3: Average number of idle mulipliers

Benchmarks	Required MUL	AVG IDLE MUL	RATIO (%)
chem	17	5.13	30.18
honda	7	3.40	48.57
dir	9	4.60	51.11
feig_dct	12	8.21	68.42
lee	4	1.56	39.00
mcm	9	5.00	55.56
u5ml	17	6.27	36.88
AVG			47.10

rithms optimize the total interconnect wirelength by minimizing the number of multiplexer inputs. We apply our scheme on the results generated by [8] and [9].

Table 3.4 shows the comparison of total wirelength. The first column of the table shows benchmark names. Columns 2 and 3 list the total wirelength in micrometer (*um*) with and without our optimization. The base designs are produced by [8]. Columns 4 and 5 list the similar results. The base designs are produced by [9]. The last two columns list the wirelength reduction. As can be seen, our scheme can further reduce interconnect length over [8] and [9]. The average improvements are 7.5% and 9.4%.

We also compare total power consumption to assess the power degradation due to the usage of idle FUs. In particular, when an idle FU is not used, we set its inputs to zeros to reduce its internal signal activities. We again shown the power dissipation of circuits after and before the application of our scheme.

Table 3.5 shows power consumption of each benchmark design. The unit of power consumption in the table is miliwatt (mW). As can be seen, on the average, the power dissipation of circuit does not increase but decrease by 4.8%. It indicates that the extra power due to the FU switching is offset by the power reduction because of interconnect reduction, although there are several circuits whose power increases after the application of our technique.

Table 3.4: Total wirelength after and before interconnect optimization using idle FUs

Benchmarks	ours	[8]	ours	[9]	Improvement over [8]	Improvement over [9]
chem	1.35×10^6	1.55×10^6	1.67×10^6	1.83×10^6	13.06%	5.81%
honda	3.22×10^5	3.42×10^5	3.47×10^5	3.84×10^5	5.68%	12.99%
dir	8.03×10^5	8.64×10^5	1.14×10^6	1.23×10^6	6.99%	7.32%
feig_dct	6.89×10^6	7.32×10^6	5.98×10^6	6.81×10^6	5.80%	12.08%
lee	3.26×10^5	3.37×10^5	2.87×10^5	3.15×10^5	3.36%	8.89%
mcm	8.81×10^5	9.82×10^5	1.09×10^6	1.17×10^6	10.28%	7.09%
u5ml	1.92×10^6	2.07×10^6	1.96×10^6	2.15×10^6	7.44%	8.84%
AVG					7.52%	9.42%

Table 3.5: Power consumption after and before interconnect optimization using idle FUs

Benchmarks	ours	[8]	ours	[9]	Increase over [8]	Increase over [9]
chem	13.17	14.76	13.27	13.09	-10.74%	1.33%
honda	3.34	3.84	2.87	2.81	-13.16%	2.25%
dir	5.83	6.19	6.03	6.62	-5.73%	-8.9%
feig_dct	36.84	37.51	26.72	28.48	-1.77%	-6.17%
lee	2.90	3.22	2.19	2.36	-10.02%	-7.20%
mcm	7.95	8.51	5.36	5.10	-6.61%	5.10%
u5ml	12.69	13.86	11.03	10.79	-8.44%	2.22%
AVG					-8.07%	-1.63%

3.6 Conclusion

In this chapter, we present a novel interconnect binding algorithm. It uses idle functional units for data transfers, resulting in global interconnect reduction. Specifically, our algorithm identifies idle functional units for each data transfer from scheduling and functional unit binding result. It assigns each data transfer in a DFG to idle functional units and/or dedicated wires to reduce wirelength of global interconnects. We transfer the problem of interconnect minimization using idle functional units to the mini-cost max-flow problem and solve it using modified EDMONDS_KARP algorithm. Experimental results show that our algorithm reduces the total wirelength of global interconnects by 8.5% and the power consumption by 4.8% without introducing any timing violations.

Chapter 4

Register Reduction Algorithm Using Idle Pipelined Functional Units

4.1 Introduction

In this chapter, we propose a register reduction algorithm. Our scheme is applied during the register binding stage of high level synthesis (HLS). The center of our scheme is to use the internal registers of idle pipelined functional units (FUs) to replace dedicated registers. Traditional register binding algorithms implicitly assume that variables in a data flow graph (DFG) are stored in only dedicated registers. We have observed that not all FUs are in operation all the time. If the FUs are pipelined to be used in high performance digital circuits, their internal registers can be used to store the variables during their idle time. Consequently, less number of dedicated registers are needed. The challenge is to identify the right FUs for the variables with minimal increase of power and interconnect complexity.

Our algorithm reduces the register count and minimizes the extra interconnects used due to the usage of idle FUs. It proceeds in four steps. First, it identifies idle pipelined

FUs throughout the entire execution time. Second, it matches variables with the internal registers in idle pipelined FUs using the bipartite matching algorithm. The matching is performed between all variables in the given DFG and internal registers of idle pipelined FUs with the objective of maximizing the number of matchings and minimizing the interconnect cost. If not all variables are mapped into the internal registers of idle pipelined FUs, dedicated registers are needed to store unmapped variables. Our algorithm calculates the minimal dedicated register count required. It then adjusts the mapping results to further reduce interconnect cost and limit power increase without increasing the number of dedicated registers. Namely, some variables mapped to internal registers of idle pipelined FUs are released and become unmapped. Finally, our algorithm assign all unmapped variables to dedicated registers.

We have applied our register reduction algorithm to a suite of benchmark designs. Experimental results have shown that our scheme reduces register counts by 26% in comparison to the Left Edge Algorithm in [11], which is considered as the optimal register reduction algorithm without using idle functional units. In addition, to assess the impact of our algorithm on global interconnect length and power consumption, we have generated the layout of each benchmark circuit using industrial EDA tools. Based on data reported by these tools, our approach achieves an average reduction of 4% and 4% of total wirelength and power, respectively, in comparison to previous register binding algorithms that target wirelength reduction [9]. Our contributions are as follows.

- A complete register binding framework that uses idle pipelined FUs and dedicated registers to store variables
- Four methods to use pipelined FUs as variable storages and formulation of the register binding problem as bipartite matching problem
- Full analysis of the effect of using pipelined FUs as variable storages with complete experimental results

The rest of this chapter is organized as follows. Section 4.2 briefly introduces previous research on register binding in HLS. Section 4.3 explains preliminaries for register binding and formulates our problem. Section 4.4 gives details about our register reduction algorithm, followed by experimental results in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.2 Related Works

During the past two decades, the register binding problem has been researched extensively with various design objectives including power reduction [18, 52, 53], interconnect reduction [19, 9, 34, 54], and timing improvement [55]. Our research focuses on register reduction since a small register count leads to small clock skews, low power dissipation, and small circuit area. Several researchers have addressed this problem. Specifically, Tseng et al. [56] proposed a clique partitioning algorithm to minimize the number of registers. Since clique partitioning is NP-complete, they presented a heuristic which gave a near-optimal solution in polynomial time. Kurdahi and Parker [11] presented the Left Edge Algorithm (LEA) for register binding. They have proved that LEA can derive a solution with the minimal number of registers. As a result, register reduction during register binding was often considered a solved problem. However, the proof in [11] was based on two assumptions. First, all variables are assumed to have the same bit width. Second, only dedicated registers are used to store variables. When either assumption does not hold, the LEA scheme is not optimal. Recently, Chabini and Wolf have presented a binding algorithm that minimizes the total number of registers without the first assumption, namely when variables have different bit widths [57]. Our work targets the scenario without the second assumption. In particular, we observe that there may exist several idle pipelined FUs in a circuit. These idle FUs can be used to store variables temporarily and therefore reduce the demand for dedicated registers.

4.3 Problem Formulation

In this section, we first describe register binding problem. We then show how idle pipelined FUs can be used for temporary data storage. We formulate our problem at the end.

4.3.1 Traditional Register Binding

The input of a register binding problem includes a scheduled DFG and the FU binding result. Figure 4.1 shows a sample input. The operations are scheduled in five time slots, called *c-steps*. The vertices are operations mapped to FUs. The strings within the vertices represent the functionalities and indexes of the corresponding FUs. Symbols '+', 'X' and '<<' denote adder, multiplier and shifter, respectively. The integers after the symbols are used to distinguish FUs of the same type. In Figure 4.1, there are two adders, '+1' and '+2', one multiplier 'X1', and one shifter '<<1'. Pipelines are represented by straight lines within vertices. In Figure 4.1, only the multiplier is pipelined by two stages.

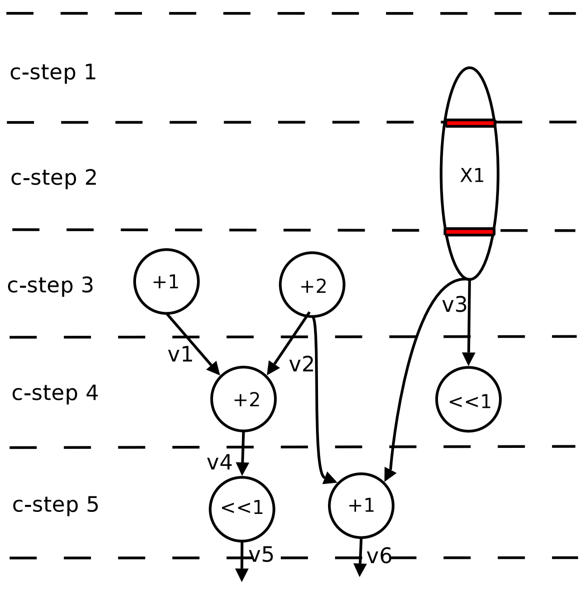

Figure 4.1: A scheduled DFG with FU binding result

Directed edges in a DFG represent variables. The starting vertex of an edge is the variable producer and the ending vertex is the consumer. The lifetime of a variable is the time duration from the c-step when the variable is produced to the c-step when it is used. Variables must be stored in registers during their lifetimes. Two variables cannot be stored in the same register if their lifetimes overlap.

If only dedicated registers are used to store variables, it is required to assign as many variables as possible to the same register so that the register count is minimized. Moreover, interconnect minimization during HLS is critical in today's VLSI design. Since no placement information is available, the interconnect cost is computed based on the connectivity among registers and FUs. Particularly, multiplexers are needed when multiple FUs store their results in the same register and multiple registers provide data to the same FU. Therefore, the interconnect cost is often represented by the multiplexer input count, which needs to be reduced during register binding.

4.3.2 Variable Storage using Idle Pipelined FUs

There are four ways to store a variable in a pipelined-FU. In the first way, registers within idle pipelined FUs are accessed only through the FU interfaces. As shown in Figure 4.2(a), the producer of the variable sends it to the input port of the idle FU. The variable is stored in the first-stage pipeline registers. It propagates one stage down after every clock cycle until the last pipeline stage. The variable will be sent to its consumer through the output port of the idle FU. In order to store variables in such a way, the pipelined FU must not initiate operation at the exact time when the variable is computed. In addition, the span of the variable lifetime must be equal to the pipeline depth of the FU minus one since the variable has to pass through the FUs to be used.

The second way to store variables in pipelined-FUs is to store the variables in the internal registers directly as shown in Figure 4.2(b). This approach does not have the restriction on the idle time of FU or the lifetime of variable as that in the first way. A

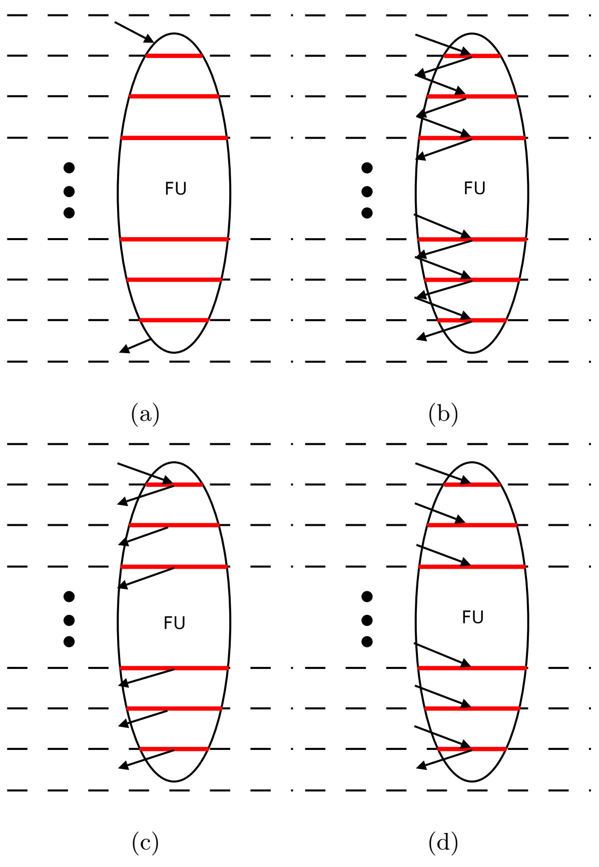

Figure 4.2: (a) Store and use a variable via FU interface (Method 1) (b) via register input/output pins (Method2) (c) via FU input port and register output pin (Method 3) (d) via register input pin and FU output port (Method 4)

variable can be stored to any pipeline register as long as the register is not in use. The variable stored can be accessed all the time. Unfortunately, the direct access of pipeline registers may lead to severe circuit performance degradation. Specifically, multiplexers must be added before the input pins of all pipeline registers, resulting in extra signal delays. Furthermore, additional fanouts are required for all registers for variable access, increasing delay and power.

The third and fourth ways are in between of the first and second. In the third way, the variables are stored via only the input ports of pipelined-FUs. However, the variables can be used via the output pins of internal registers at any pipeline stages. Since variables are stored via input ports, the variables should be birth from the c-step when a pipelined-FU starts to be idle. However, the death time can be anytime until the last pipeline stage of the FU. Thus, this method provides more flexibility in the sense of variable accommodation. As shown in Figure 4.2(c), if the third way is used, one of variables as many as the number of pipeline stages can be stored in the pipelined FU, which gives more chances to reduce register count than Method 1. However, note that this method also causes additional multiplexer and fanout overheads which is in between of that of the Method 1 and Method 2.

The fourth way in Figure 4.2(d) is opposite to the Method 3 in storing variables. Namely, variables can be stored via input pins of internal registers at any pipeline stages, but they are used via only output ports of the pipelined FU. The benefit of this way is similar to that of the third way. However, the lifetime span of variables which the FU can accommodate is different. The birth time of variables can be any c-steps within pipeline stages. The death time is only the c-step of last pipeline stage, since variables can be used via only the output ports of the FU. This method also causes additional input multiplexers of pipeline registers and fanouts of output ports of the FU.

In this paper, we explain our algorithm based on Method 2. Since it provides the full flexibility for variable storage, it is also superset of other methods. Thus, our algorithm based on Method 2 is applied to all other methods with slight modification. We examine all the four methods described above to see how much benefit they provide in terms of register counts and clock skew reduction and how much they impact to the circuit quality in terms of power and interconnect.

4.3.3 Problem Formulation

Since we access all the internal pipeline registers of a FU in general, the registers within the FU can be utilized independently. The storage of variables in those registers can, therefore, be modeled as mapping of variable lifetime to the timeframes in which the FU can retain a value. Consequently, our register binding problem can be formulated as follows:

PROBLEM: Given a scheduled DFG with a FU binding result, assign variables in the DFG to the set of internal registers of pipelined FUs and dedicated registers with the objective of minimizing dedicated registers and the global interconnect cost, which is calculated as the total multiplexer inputs of the FUs and dedicated registers.

4.4 The Proposed Approach

Figure 4.3 shows the overall flow of our register binding algorithm which consists of four steps. We first identify all pipelined FUs that are idle for each c-step. We then assign variables to these idle FUs with the objective of reducing dedicated registers and interconnect cost. To the end, we construct a bipartite graph whose one partition consists of variables and the other contains idle pipelined FUs. An edge is added between a variable and a FU, if the variable can be assigned to the FU, with the edge cost equal to the interconnect cost of the assignment. We assign as many variables as possible to FUs with the minimal total edge cost. In the third step, we first compute the number of dedicated registers needed based on the previous assignment result. we perform post processing to remove some assignments as long as the removal will not lead to the increase of dedicated registers. Finally, we perform bipartite-matching register binding algorithm [9] to assign unassigned variables to dedicated registers and minimize the interconnect cost, i.e., the total number of multiplexer inputs. In the following sections, we present the details of our algorithm.

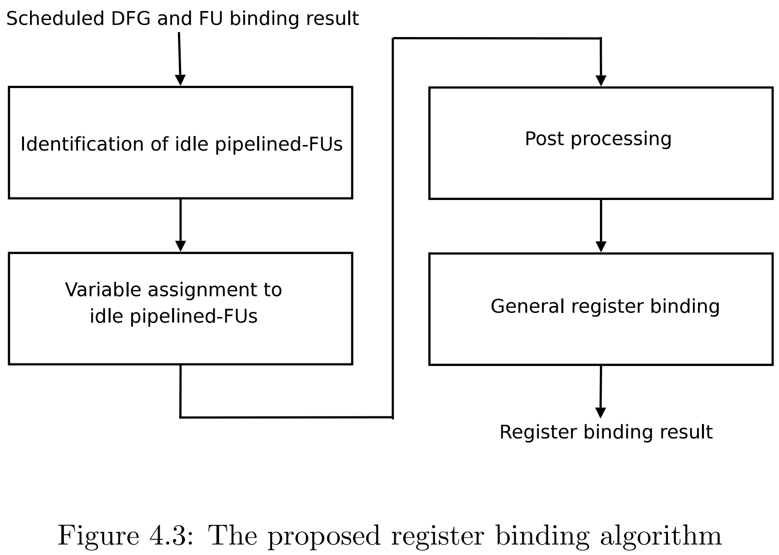
Scheduled DFG and FU binding result
Identification of idle pipelined-FUs
Variable assignment to idle pipelined-FUs
Post processing
General register binding
Register binding result

Figure 4.3: The proposed register binding algorithm

4.4.1 Identification of Idle Pipelined FUs

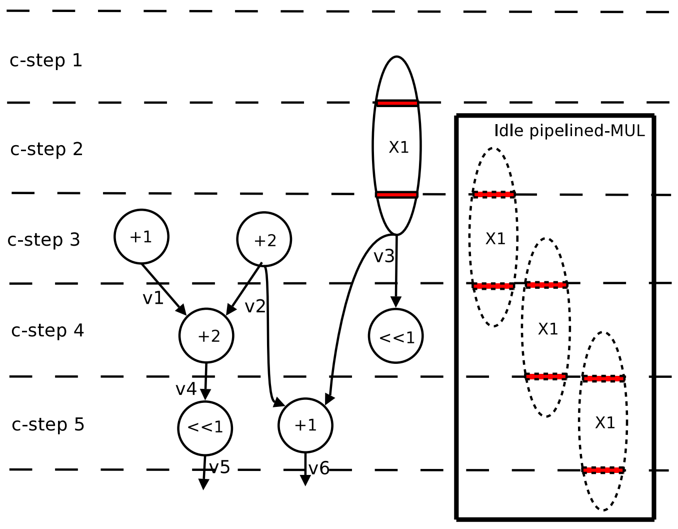

Figure 4.4: Identification of idle pipelined FUs

If a pipelined FU does not initiate computation at a c-step i, it is considered to be *idle* from c-step i to c-step i+p-1 where p is the number of pipeline stages of the FU. With the given DFG and FU binding information, the identification of idle pipelined FUs is straightforward. Figure 4.4 shows the example from Figure 4.1, where only the multiplier is pipelined. Our algorithm examines each c-step of the DFG. Since multiplier X1 initiates multiplication only at c-step one, it is idle at all other c-steps. When a pipelined FUs is idle for consecutive c-steps, the idle time will not be combined but processed independently. As shown in Figure 4.4, three dotted ovals in the bold box show the time slots when multiplier 'X1' is idle.

4.4.2 Variable Assignment to Idle Pipelined-FUs

In the second step of our algorithm, we assign variables to the internal registers of pipelined FUs to reduce the need for dedicated registers. Before delving into the details of this step, we define *FU lifetime* as follows.

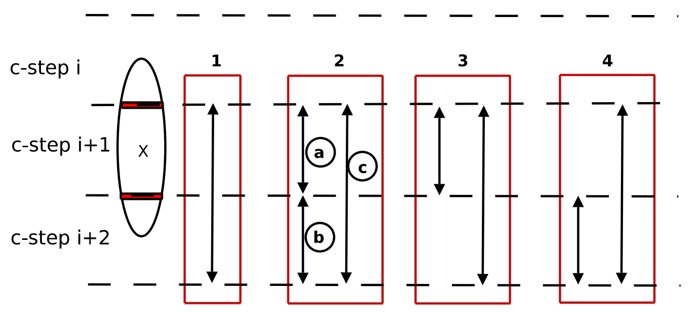

Figure 4.5: FU lifetimes of 3-stage pipelined multiplier for four storing methods

Definition 6 *FU lifetime is the time duration during which a pipelined FU can retain a value.*

Figure 4.5 shows *FU lifetimes* of a 3-stage multiplier for all possible storing methods described in Section 2.4. In case of Method 1 shown in *box 1*, it has only one FU lifetime, since it stores a value via only FU input/output ports. In other words, it can retain only one variable from the first pipeline register to the last one. On the other hand, the FU can have multiple FU lifetimes for other storing methods. Box 2, 3 and 4 in Figure 4.5 shows FU lifetimes for storing Method 2, 3 and 4, respectively. Note that variables cannot be assigned to overlapping FU lifetimes at the same time. In general, for k-stage pipelined FU, it can have as many FU lifetimes as follows.

$$\begin{cases} 1 & \text{for Method 1} \\ \frac{k\times(k-1)}{2} & \text{for Method 2} \\ k-1 & \text{for Method 3 and 4} \end{cases}$$

Therefore, in this step, we assign variables to the FU lifetimes. To that end, we recast the *variable-to-FU lifetime* assignment problem into a bipartite matching problem. Specifically, we construct a bipartite graph as follows. On one side of the graph, vertices are introduced for variables in the given DFG. If a variable is consumed at a single c-step, the variable is represented by one vertex, with the lifetime augmented. If a variable is consumed at multiple c-steps, one vertex is added for each consuming c-step. The lifetime of the vertex is between the producing c-step and the consuming c-step. Vertices on the other side of the bipartite graph represent FU lifetimes of idle pipelined FUs. Multiple vertices may be added for a FU. Particularly, for each c-step when a pipelined FU does not initiate computation, vertices are inserted. The number of vertices for a FU at a c-step depends on which storing method is used.

An edge is inserted between a variable vertex and a FU vertex if the variable can be assigned to the FU. We compare variable lifetimes to each FU lifetime to construct edges between two partitions. If variable lifetime in one partition is same with FU lifetime in the other, an edge between two vertices is created. Since there are multiple FU lifetimes for a pipelined FU, multiple variables can be potentially assigned to the idle pipelined FU during certain idle time. However, all those variables cannot be assigned to the FU, since there are overlapping FU lifetimes. For example, in Box 2 of Figure 4.5, there are three FU lifetimes such as a, b and c between c-step i and c-step $i+2$. Although three variables can be matched with the three FU lifetimes, all variables cannot be stored in the FU, since FU lifetime c overlaps those of a and b. Thus, after finding all matches between variables and FU lifetimes, those overlaps should be removed.

A weight is assigned to each edge, indicating the interconnect cost of assigning the corresponding variable, i.e., the starting vertex, to the ending FU of the edge. Since the potential increase of multiplexer inputs and fanouts is used to represent the extra interconnect, the weight of an edge between variable i and FU j is calculated by the equation:

$$C_{i,j} = NIN_j + N_{i,j} + NOUT_i \ , \tag{4.1}$$

where NIN$_j$ is the number of different input variables that FU j receives according to the given DFG. The item N$_{i,j}$ shows how many new interconnects are connected to the FU j by assigning variable i. If variable i is to be connected to the input ports of FU j and is among the input variables of input ports of FU j, N$_{i,j}$ is 0. If it is newly added variable to be stored via input ports, N$_{i,j}$ is constant α which is greater than one. We set α two. If it is added to the inputs of internal registers, N$_{i,j}$ is one. The reason α is greater than one is that input ports of a FU are already crowded due to existing input variables in a DFG. If we add more variables to the ports, it makes the congestion around the ports worse than putting a variable into internal register directly. Thus, we model the negative effect on congestion around input ports of a FU as α. The sum $\mathrm{NIN}_j + \mathrm{N}_{i,j}$ represents the multiplexer inputs at the input interface of FU j and internal pipeline registers. Intuitively, if a FU consumes many variables, it is likely that the FU needs a multiplexer with many inputs, which increases the global interconnect complexity. The parameter NOUT_i is the number of fanouts of variable i. It represents the increase of fanouts of FU j when it is used to store variable i. Large increase of the fanout count increases the overall interconnect complexity. After construction of the weighted bipartite graph, the assignment of variables to idle pipelined FUs with minimal interconnect cost can be solved by computing the minimum cost bipartite matching. In our scheme, we apply the Hungarian algorithm [58] to derive the optimal matching solution.

Figure 4.6 illustrates the procedure of bipartite graph construction using the example in Figure 4.4. Figure 4.6(a) presents the lifetimes of all variables. It also indicates that the multiplier does not initiate any computation, i.e., is idle, at c-step 2, c-step 3 and c-step 4. Consequently, six variable vertices and nine FU lifetime vertices are placed in the bipartite graph in Figure 4.6(b). We assume that Method 2 is used for this example. Edges are inserted based on the variable lifetimes and FU idle situations with edge weights calculated according to Equation (4.1). For example, an edge is added between v_1 and $m2$, since v_1 can be assigned to the multiplier at the end of c-step 3. The multiplier has two different input variables in c-step 1. Therefore, NIN_{m2} is two. The parameter $\mathrm{N}_{v1,m2}$ is 1, since $v1$ is not connected to the input ports of the multiplier in the DFG and it may be stored in the internal register at second stage pipeline. The parameter NOUT_{v1} is one because $v1$ is used by one adder at c-step 4 which is not originally connected to the output of the multiplier. The edge weight is therefore equal to $2+1+1=4$. Once the bipartite graph is constructed as shown in Figure 4.6(b), we perform the min-cost bipartite matching algorithm. For this example, $v1$, $v3$, $v4$ and $v5$ are matched to $m2$, $m6$, $m5$ and $m8$, respectively, with the total cost of 12. However, since $m5$ and $m6$ overlap each other as shown in Figure 4.6(a), $v3$ and $v4$ cannot be assigned to the multiplier as the same time. Thus, a match with minimum cost among conflicted matches is selected. In this example, a match between $v4$ and $m5$ is selected. Finally, $v1$, $v4$ and $v5$ are assigned to the internal registers of the multiplier.

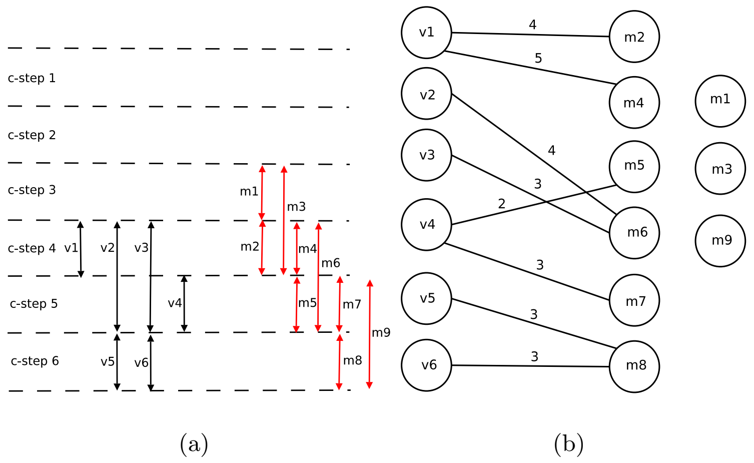

Figure 4.6: (a) Lifetimes of variables and idle pipelined multiplier (b) Bipartite graph constructed

4.4.3 Post Processing

After the variable assignment described in Section 4.4.2, there may still exist variables that are unassigned to pipelined FUs. Dedicated registers must be introduced for these variables. The minimal number of registers needed is equal to the maximal unassigned variables in any c-step.

Our algorithm scans through all c-steps to count the numbers of unassigned variables and derive the minimal number of dedicated registers N_{reg}. It then unassigns variables from FUs as long as the unassignment does not increase N_{reg}. The rationale behind our post processing step is that storing a variable using a pipeline FU is often more costly in terms of power than using a dedicated register. As a result, the existing dedicated registers should be used before any pipelined FU is used.

Algorithm 4 Post processing for reduction of multiplexer inputs

Input: Minimum number of registers (N_{reg}), scheduled and FU-bound DFG (G)

Output: Refinement of variable-to-FU assignment

```
1:  foreach $Variable$ in G do
2:    if $Variable$ is assigned to a pipelined FU then
3:      foreach c-step in the lifetime of $Variable$ do
4:        if unassigned variable count is $N_{reg}$ then
5:          set flag
6:        end if
7:      end foreach
8:      if flag not set then
9:        unassign $Variable$
10:     end if
11:   end if
12: end foreach
```

Algorithm 4 shows the detailed steps for variable unassignment. The input contains a scheduled and FU-bound DFG and the number of dedicated registers N_{reg}. The output is the refined variable-to-FU assignment. Specifically, all variables assigned to pipelined FUs are selected first (**lines 1 and 2**). The unassigned variable count of each c-step during the lifetime of every selected variable is compared with N_{reg} (**lines 3 to 7**). If all the counts are less than N_{reg}, there are unused dedicated registers during the variable lifetime. Therefore, the variable is unassigned (**lines 8 to 10**). The post processing ends when all variables are examined.

The benefit of the variable unassignment can be illustrated using the example in Figure 4.6. According to our variable assignment solution in Section 4.4.2, $v1$, $v4$ and $v5$ are assigned to $m2$, $m5$ and $m8$, respectively. Thus, there are two unassigned variables, $v2$ and $v3$, from c-step 4 to c-step 5. There is one unassigned variable $v6$ at c-step 6. The minimal number of dedicated registers N_{reg} is two. Our algorithm checks $v1$ and $v4$. In case of $v1$, its lifetime includes c-step 4, which has two unassigned variables. Therefore, $v1$ cannot be unassigned, since unassignment of $v1$ increases the number of dedicated registers to three. On the other hand, the lifetime of $v5$ only spans over c-step 6. Since there is only one unassigned variable $v6$ at c-step 6, $v5$ can be unassigned from $m8$. Storing $v5$ into a dedicated register helps reducing power consumption. In addition, it also reduces the number of variables fed into the multiplier, leading to less multiplexer inputs and therefore lower interconnect complexity.

After the refinement of variable-to-FU assignment, we assign all unassigned variables to dedicated registers. The bipartite-matching register binding algorithm [9] is used that guarantees the minimal number of dedicated registers and reduces total multiplexer inputs.

4.5 Experimental Setup and Results

In this section, we present experimental results that illustrate the effectiveness of our approach in register reduction. We also describe the benefit of our scheme in clock skew minimization. In addition, we show overhead of interconnect and power consumption.

4.5.1 Experimental Setup

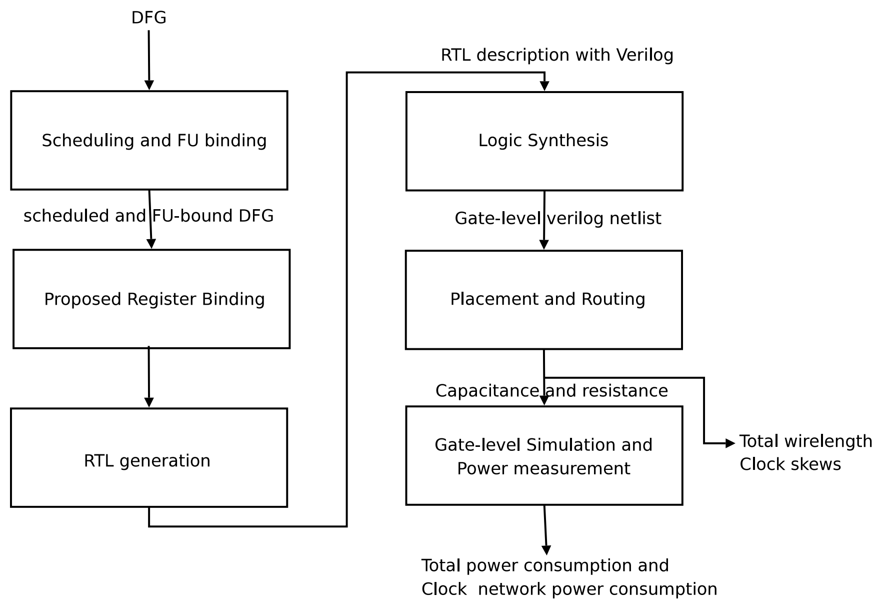
DFG
Scheduling and FU binding
scheduled and FU-bound DFG
Proposed Register Binding
RTL generation
RTL description with Verilog
Logic Synthesis
Gate-level verilog netlist
Placement and Routing
Capacitance and resistance
Gate-level Simulation and Power measurement
Total wirelength
Clock skews
Total power consumption and
Clock network power consumption

Figure 4.7: Design and Evaluation Flow

We have implemented our register binding algorithm into a HLS software tool and applied it to the data intensive benchmarks in [43]. All operations in these benchmarks are either addition or multiplication. Figure 4.7 shows the overall flow of our experiments. We first convert the benchmarks into DFGs. We then apply the Force-directed scheduling scheme [46] and Left Edge FU binding algorithm [11] to perform the operation scheduling and FU binding, respectively. We assume that only multipliers are pipelined, since delay of a non-pipelined multiplier is much longer than that of an adder. In addition, we experiment with 2-, 3- and 4-stage pipelined multipliers to examine how the benefit and overhead vary with respect to the number of pipeline stages. Our proposed algorithm is then used to complete the register binding with the utilization of idle FUs.

To assess the exact impact of our scheme on interconnect and power, we construct all benchmarks into layouts using industrial EDA tools. Specifically, an in-house tool converts the HLS results into Verilog RTL descriptions. Synopsys Design Compiler$^{\text{TM}}$ is then applied to synthesize the RTL descriptions into gate level netlists. Cadence SOC Encounter$^{\text{TM}}$ is next used to perform placement and routing to generate the layouts. The clock tree of each circuit is designed with skew minimization as the objective.

To estimate power consumption of our circuits, we simulate their gate netlists with randomly generated inputs. The switching activity information of all nodes in the netlists is stored in the Switching Activity Interchange Format (SAIF). Design Compiler$^{\text{TM}}$ is then used to derive power estimates using the SAIF files and the capacitance information from the layouts. Circuit statistics such as total wirelength and clock skews are reported by SOC Encounter$^{\text{TM}}$.

4.5.2 Experimental Results

Register Reduction

Table 4.1, 4.2 and 4.3 compare the register counts of circuits designed using LEA and our approach. We also test four storing methods described in Section 4.3.2. In case of 2-stage pipelined multiplier, we only test Method 1 which stores variables via FU interface, since it has only one stage pipeline register.

Table 4.1 shows register reduction for a DFG where multiplier is pipelined by 2-stage. Specifically, the first column lists the benchmark names. The second and third columns show the register counts for LEA and our algorithm, respectively. The fourth lists the register reduction of our scheme over LEA, which produces the optimal result without considering idle FUs. As Table **??** shows, our algorithm reduces registers for all benchmark circuits. The average reduction is 26%. The highest reduction is 100%. The

Table 4.1: Comparison of total number of registers (2-stage)

	2-stage		
Benchmarks	LEA	Ours	red
chem	576	512	11%
honda	240	224	7%
feig_dct	1408	1264	10%
lee	160	128	20%
mcm	400	272	32%
u5ml	624	576	8%
wang	176	112	36%
pr	160	96	40%
arai	160	128	20%
chendct	336	256	24%
chenidct	384	288	25%
fft	256	0	100%
fir11	48	32	33%
cftmdl	352	256	27%
KALMAN	64	48	25%
LowPass	176	112	36%
matmul	512	256	50%
idct	272	224	18%
jacob	384	320	17%
Wavelet	256	128	50%
AVG			26%

reason *fft* can get 100% reduction is that the absolute lifetimes of all variables are two c-steps. Thus, if there are sufficient number of idle multipliers, 2-stage pipelined multipliers can accommodate those short-lifetime variables.

The Table 4.2 and 4.3 show register reduction when 3- and 4-stage pipelined multipliers are used. The first column shows the benchmarks. The second column shows the number of registers when LEA performs register binding. The third, fourth, fifth and sixth columns show the register count when four different storing methods are used. The last four columns present the relative reduction over LEA. The average reduction for the four methods using 3-stage pipelined multiplier is 8%, 41%, 22% and 27%, respectively. In addition, When 4-stage pipelined multipliers are used, 2%, 44%, 28% and 27% reduction is

Table 4.2: Comparison of total number of registers (3-stage)

	3-stage								
Benchmarks	LEA	M1	M2	M3	M4	M1-red	M2-red	M3-red	M4-red
chem	576	544	464	576	512	6%	19%	0%	11%
honda	240	192	144	208	192	20%	40%	13%	20%
dir	272	256	240	272	256	6%	12%	0%	6%
feig_dct	1344	1248	1200	1328	1248	7%	11%	1%	7%
lee	176	176	160	176	144	0%	9%	0%	18%
mcm	400	400	224	304	272	0%	44%	24%	32%
u5ml	640	576	528	640	592	10%	18%	0%	8%
wang	160	144	128	128	112	10%	20%	20%	30%
pr	160	144	80	112	96	10%	50%	30%	40%
arai	128	112	112	112	112	13%	13%	13%	13%
chendct	288	256	240	256	240	11%	17%	11%	17%
chenidct	368	368	272	304	272	0%	26%	17%	26%
fft	256	256	0	0	128	0%	100%	100%	50%
fir11	64	48	32	48	32	25%	50%	25%	50%
cftmdl	288	288	144	176	192	0%	50%	39%	33%
KALMAN	64	64	32	48	48	0%	50%	25%	25%
LowPass	160	144	48	128	96	10%	70%	20%	40%
matmul	512	512	0	256	256	0%	100%	50%	50%
idct	272	224	192	224	208	18%	29%	18%	24%
jacob	352	304	208	320	288	14%	41%	9%	18%
Wavelet	256	256	0	128	128	0%	100%	50%	50%
AVG						8%	41%	22%	27%

Table 4.3: Comparison of total number of registers (4-stage)

	4-stage								
Benchmarks	LEA	M1	M2	M3	M4	M1-red	M2-red	M3-red	M4-red
chem	560	496	304	528	480	11%	46%	6%	14%
honda	224	224	112	208	192	0%	50%	7%	14%
dir	272	240	224	240	256	12%	18%	12%	6%
feig_dct	1296	1296	1184	1280	1200	0%	9%	1%	7%
lee	160	160	144	160	144	0%	10%	0%	10%
mcm	400	400	224	288	272	0%	44%	28%	32%
u5ml	608	608	480	608	544	0%	21%	0%	11%
wang	160	160	144	128	112	0%	10%	20%	30%
pr	160	144	96	112	96	10%	40%	30%	40%
arai	128	128	96	112	96	0%	25%	13%	25%
chendct	256	256	192	224	192	0%	25%	13%	25%
chenidct	368	368	288	304	304	0%	22%	17%	17%
fft	256	256	0	0	128	0%	100%	100%	50%
fir11	64	64	32	64	32	0%	50%	0%	50%
cftmdl	256	256	176	128	176	0%	31%	50%	31%
KALMAN	48	48	16	32	32	0%	67%	33%	33%
LowPass	144	144	32	80	80	0%	78%	44%	44%
matmul	512	512	0	0	256	0%	100%	100%	50%
idct	272	240	192	256	240	12%	29%	6%	12%
jacob	336	320	192	304	272	5%	43%	10%	19%
Wavelet	256	256	0	0	128	0%	100%	100%	50%
AVG						2%	44%	28%	27%

achieved. In case of Method 1, the effect of register reduction is relatively smaller than other methods. The reason is that the pipelined FUs can store variables which have exactly the same length of lifetime with FU-lifetime. For 3-stage pipelined multiplier, it can store only variables which have the lifetime length of three c-steps. Thus, the reduction depends on how many variables have the lifetime length of three c-steps. On the other hand, Method 2 achieves the most reduction, since it can accommodate any length of lifetime between the minimum and maximum FU-lifetimes of the FU.

Clock Skew Minimization

We investigate the impact of our scheme on clock skews by building min-skew clock trees using SOC Encounter^{TM}. We compare our approach with the bipartite matching register binding algorithm [9], which not only derives the same register counts as LEA but also optimizes multiplexer inputs and interconnect complexity. Table 4.4, 4.5 and 4.6 report the results. The unit of skews is pico-second. We measure rising and falling clock skews when 2-, 3- and 4-stage pipelined multipliers are used for each benchmark program. In addition, we apply four storing methods for each kind of multiplier except 2-stage pipelined multiplier.The Table 4.4 shows the comparison of clock skew when 2-stage pipelined multipliers are used. Method 1 is used to store variables in the internal registers of the multipliers. The second and third columns show the rising and falling skews for circuits designed using bipartite matching register binding algorithm. The fourth and fifth columns are the rising and falling skews for circuits designed by our algorithm. Last two columns show the percentage reduction. Our scheme delivers better results in most of the cases. The average skew reduction is 25% and 26% for rising and falling edge, respectively. The Table 4.5 and 4.6 present the clock skew comparison when 3- and 4-stage multipliers are used, respectively. Each table has the clock skew measure for each storing method. The second and third columns show clock skew measure for bipartite matching register binding algorithm. Columns from fourth to eleventh show clock skew for each storing methods.

The rest of columns shows the percentage reduction over bipartite matching algorithm. As Table 4.5 and 4.6 show, our algorithm reduces the rising (falling) clock skew by 28% (25%), 29% (27%), 28% (27%) and 33% (30%) on the average for M1, M2, M3 and M4, respectively. When 4-stage pipelined multipliers are used, the average reduction of rising (falling) clock skew is 7% (8%), 27% (19%), 22% (19%) and 24% (22%) for M1, M2, M3 and M4, respectively. By minimization of register counts, we achive the clock skew minimization, too. The reason is that the loading effect for clock signal line is reduced by reducing the number of registers.

Table 4.4: Comparison of clock skew (2-stage)

	2-stage					
Benchmarks	BIP-r	BIP-f	Ours-r	Ours-f	r-red	f-red
chem	53.70	57.10	48.90	50.8	9%	11%
honda	14.80	13.50	8.70	8.7	41%	36%
dir	77.70	78.00	21.20	23	73%	71%
feig_dct	66.30	62.20	52.30	47.1	21%	24%
lee	10.60	18.50	9.30	12.1	12%	35%
mcm	195.00	198.00	85.00	86	56%	57%
u5ml	90.80	93.20	79.00	85.3	13%	8%
wang	31.30	21.50	22.60	20.4	28%	5%
pr	49.10	36.00	46.40	46.4	5%	-29%
arai	81.50	82.40	20.80	20.8	74%	75%
chendct	28.30	29.10	17.90	18.3	37%	37%
chenidct	38.00	38.40	23.30	24.4	39%	36%
fft	21.70	23.40	18.20	18.9	16%	19%
fir11	12.90	13.10	7.50	5	42%	62%
cftmdl	67.30	66.10	75.60	44.9	-12%	32%
KALMAN	22.90	22.90	35.40	35.6	-55%	-55%
LowPass	35.60	35.50	30.60	30.7	14%	14%
matmul	72.20	61.50	45.00	45.7	38%	26%
idct	28.00	28.50	20.50	21	27%	26%
jacob	14.60	15.00	14.10	14.8	3%	1%
Wavelet	43.70	46.40	23.00	21.7	47%	53%
AVG					25%	26%

Table 4.5: Comparison of clock skew (3-stage)

	3-stage																	
Bench	BIPr	BIPf	M1r	M1f	M2r	M2f	M3r	M3f	M4r	M4f	M1r-red	M1f-red	M2r-red	M2f-red	M3r-red	M3f-red	M4r-red	M4f-red
chem	41.90	37.70	32.20	33.00	13.90	15.70	27.60	28.00	40.7	56.1	23%	12%	67%	58%	34%	26%	3%	-49%
honda	33.20	119.80	15.00	15.80	17.60	19.50	20.90	19.40	18.2	19.3	55%	87%	47%	84%	37%	84%	45%	84%
dir	28.80	22.10	19.70	19.90	21.90	22.10	16.40	17.30	35.9	22.9	32%	10%	24%	0%	43%	22%	-25%	-4%
feig_dct	42.60	40.20	55.00	52.70	72.60	89.60	40.80	34.00	52.4	51	-29%	-31%	-70%	-123%	4%	15%	-23%	-27%
lee	14.40	14.30	14.40	14.30	9.70	10.40	12.50	12.40	9.6	10.7	0%	0%	33%	27%	13%	13%	33%	25%
mcm	36.00	36.20	13.90	14.10	15.80	16.10	25.90	27.50	15.2	15.6	61%	61%	56%	56%	28%	24%	58%	57%
u5ml	33.70	30.60	41.40	41.30	44.50	40.40	31.20	32.50	38.8	33.1	-23%	-35%	-32%	-32%	7%	-6%	-15%	-8%
wang	41.60	42.70	14.20	14.50	15.10	15.90	32.50	33.00	21	21	66%	66%	64%	63%	22%	23%	50%	51%
pr	65.90	65.90	17.10	17.00	13.40	14.10	13.80	14.80	17.6	18.4	74%	74%	80%	79%	79%	78%	73%	72%
arai	41.00	41.00	13.20	12.70	23.40	19.30	19.20	19.10	13.7	11.3	68%	69%	43%	53%	53%	53%	67%	72%
chendct	39.20	33.40	40.00	41.10	27.30	28.00	34.70	30.80	38.5	40.9	-2%	-23%	30%	16%	11%	8%	2%	-22%
chenidct	89.20	91.60	31.60	30.80	46.90	46.70	66.70	67.40	14.5	17.1	65%	66%	47%	49%	25%	26%	84%	81%
fft	56.60	60.10	56.60	60.10	44.10	44.20	73.40	77.20	30.9	31.4	0%	0%	22%	26%	-30%	-28%	45%	48%
fir11	51.80	52.40	17.00	16.60	15.60	15.20	7.40	9.20	44.6	44.7	67%	68%	70%	71%	86%	82%	14%	15%
cftmdl	29.70	30.50	29.70	30.50	35.90	35.70	22.80	23.60	40.1	42.3	0%	0%	-21%	-17%	23%	23%	-35%	-39%
KALMAN	28.10	28.20	28.10	28.20	15.10	10.90	13.30	13.30	10.7	9.6	0%	0%	46%	61%	53%	53%	62%	66%
LowPass	13.00	13.70	10.50	12.00	10.30	10.50	13.20	14.30	10.7	10.9	19%	12%	21%	23%	-2%	-4%	18%	20%
matmul	51.30	36.20	51.30	36.20	33.20	32.10	35.60	32.90	37	43	25%	-8%	35%	11%	31%	9%	28%	-19%
idct	57.60	60.40	38.40	39.00	46.80	46.80	45.90	49.10	18	18.1	33%	35%	19%	23%	20%	19%	69%	70%
jacob	54.90	56.00	26.50	27.10	39.30	40.20	40.70	43.70	12.6	14.1	52%	52%	28%	28%	26%	22%	77%	75%
Wavelet	58.50	62.50	58.50	62.50	38.20	41.40	46.80	47.20	19.2	20.4	0%	0%	35%	34%	20%	24%	67%	67%
AVG											28%	25%	29%	27%	28%	27%	33%	30%

Table 4.6: Comparison of clock skew (4-stage)

	4-stage																	
Bench	BIPr	BIPf	M1r	M1f	M2r	M2f	M3r	M3f	M4r	M4f	M1r-red	M1f-red	M2r-red	M2f-red	M3r-red	M3f-red	M4r-red	M4f-red
chem	73.30	66.90	28.30	28.20	26.70	26.00	46.40	40.20	37.1	34	61%	58%	64%	61%	37%	40%	49%	49%
honda	33.60	34.60	33.60	34.60	44.30	44.80	15.90	18.10	30.1	30.6	0%	0%	-32%	-29%	53%	48%	10%	12%
dir	33.80	36.00	23.30	25.40	30.30	30.50	24.70	25.10	21.7	22.4	31%	29%	10%	15%	27%	30%	36%	38%
feig_dct	36.40	33.80	36.40	33.80	31.80	42.20	35.20	33.00	46.9	38.2	0%	0%	13%	-25%	3%	2%	-29%	-13%
lee	16.20	15.60	16.20	15.60	12.10	11.70	16.20	15.60	9.6	10.3	0%	0%	25%	25%	0%	0%	41%	34%
mcm	29.20	30.40	29.20	30.40	14.10	15.50	15.10	14.50	24.7	26.1	0%	0%	52%	49%	48%	52%	15%	14%
u5ml	31.10	29.90	31.10	29.90	45.50	40.30	31.10	29.90	42.1	42.2	0%	0%	-46%	-35%	0%	0%	-35%	-41%
wang	44.10	44.60	44.10	44.60	35.90	37.80	48.60	48.70	15.9	17.1	0%	0%	19%	15%	-10%	-9%	64%	62%
pr	36.00	45.80	40.30	40.70	22.80	24.50	27.40	24.40	25.6	27.8	-12%	11%	37%	47%	24%	47%	29%	39%
arai	12.70	12.70	12.70	12.70	16.50	17.90	12.70	12.70	15.5	13.4	0%	0%	-30%	-41%	0%	0%	-22%	-6%
chendct	77.90	79.80	77.90	79.80	29.10	30.00	52.70	53.60	21.8	23.4	0%	0%	63%	62%	32%	33%	72%	71%
chenidct	52.60	56.60	52.60	56.60	18.40	20.00	32.70	33.60	34.1	36.2	0%	0%	65%	65%	38%	41%	35%	36%
fft	36.80	28.50	36.80	28.50	20.60	21.80	31.80	34.80	35.6	35.7	0%	0%	44%	24%	14%	-22%	3%	-25%
fir11	15.00	13.10	15.00	13.10	14.60	14.60	15.00	13.10	7.3	6.9	0%	0%	3%	-11%	0%	0%	51%	47%
cftmdl	34.50	28.80	34.50	28.80	31.90	30.50	25.70	28.80	50	49.9	0%	0%	8%	-6%	26%	0%	-45%	-73%
KALMAN	30.40	30.50	30.40	30.50	25.80	25.80	34.60	34.70	12.9	12.2	0%	0%	15%	15%	-14%	-14%	58%	60%
LowPass	28.10	28.80	28.10	28.80	11.60	12.60	17.10	18.10	22.9	23.3	0%	0%	59%	56%	39%	37%	19%	19%
matmul	95.80	84.80	95.80	84.80	32.70	29.70	42.90	40.10	55.6	45.5	0%	0%	66%	65%	55%	53%	42%	46%
idct	64.50	65.70	45.50	45.50	40.70	41.10	19.70	20.90	46.6	49.4	29%	31%	37%	37%	69%	68%	28%	25%
jacob	58.60	58.90	30.90	31.30	27.10	28.70	58.80	58.50	46.1	46.2	47%	47%	54%	51%	0%	1%	21%	22%
Wavelet	74.00	63.60	74.00	63.60	59.60	64.20	64.70	65.70	34.5	30	0%	0%	19%	-1%	13%	-3%	53%	53%
AVG											7%	8%	27%	19%	22%	19%	24%	22%

Overhead of Interconnect and Power Consumption

Since we use FUs to store variables, there are possibility to increase interconnect by redirecting variables to the pipelined FUs. In addition, variables stored in the internal registers of pipelined FUs propagate down to the next stage of pipeline registers in the FUS. Thus, it increases the power consumption to store the variables. However, we also limit the increase of interconnect by incorporating multiplexer inputs into cost function. Thus, we expect that the overhead of interconnect and power consumption is handled within small increase. The experimental results confirm the argument.

First of all, we examine how much interconnect overhead is incurred by using pipelined FUs as variable storages. We compare our algorithm with the bipartite matching register binding algorithm in terms of total wirelength, estimated based on the layouts from SOC Encounter^{TM}. Table 4.7, 4.8 and 4.9 shows the results. We measure the total wirelength for 2-, 3- and 4-stage pipelined multipliers. Moreover, for the case using 3- and 4-stage pipelined multipliers, we also examine four different storing methods. The unit of wirelength is micro-meter. The Table 4.7 shows the total wirelength when 2-stage pipelined multiplier is used in a given DFG. The second and third columns list the total wirelength of circuits obtained using bipartite matching register binding algorithm and our algorithm, respectively. Our scheme is comparable in comparison to the bipartite matching algorithm. It reduces total wirelength slightly, by 4% on average. The Table 4.8 presents total wirelength comparison among bipartite matching algorithm and our algorithm with four storing methods when 3-stage pipelined multipliers are used. The columns from second to fifth are total wirelength for each method. As indicated in the Table 4.8, our algorithm increases the total wirelength by 0%, 10%, 1% and 2% on average for four storing methods. The increase is negligible except for the M2. Since M2 stores the most variables, it also increases the interconnect complexity the most. However, when 4-stage pipelined multiplier is used, the increase of total wirelength is more than 3-stage one. The increase is 0%, 9%, 3% and 10% for each storing method. The reason that 4-stage multiplier worsen the interconnect is that it has more FU lifetimes than 3-stage one. In other words, since it can accommodate more variables in it, more FU outputs generating the variables are connected to the 4-stage pipelined multipliers. However, note that the increase of Method 1 is smaller than 3-stage case, sinc it reduce less less number of registers, which results in less interconnect complexity.

Our algorithm increases the number of multiplexer inputs slightly. As shown in Table **??**, the average increase of multiplexer input count is 3% for 2-stage multiplier in comparison to the bipartite matching register binding algorithm. When 3-stage multiplier is used, the multiplexer inputs increase by 1%, 6%, 1% and 6% on average for Method 1, 2, 3 and 4, respectively. In case of 4-stage multiplier, the average increase of multiplexer is 1%, 8%, -1% and 7%, respectively. In case of Method 2 and 4, the increase is more than Method 1 and 3. The reason is that Method 2 and 4 store variables via register input. If a FU or a register is connected to multiple internal registers of a pipelined FU, the multiplexer input count is the number of the internal registers. However, if it is connected to only input ports of the pipelined FU, the multiplexer input count is just one.

Table 4.7: Comparison of total wirelength (2-stage)

	2-stage		
Benchmarks	BIP	Ours	inc
chem	6.93×10^5	7.18×10^5	4%
honda	2.98×10^5	2.28×10^5	-23%
dir	3.53×10^5	3.37×10^5	-4%
feig_dct	1.92×10^6	1.87×10^6	-3%
lee	1.59×10^5	1.44×10^5	-10%
mcm	3.20×10^5	3.42×10^5	7%
u5ml	1.08×10^6	1.09×10^6	1%
wang	2.76×10^5	2.48×10^5	-10%
pr	2.28×10^5	2.28×10^5	0%
arai	1.32×10^5	9.41×10^4	-29%
chendct	3.27×10^5	3.16×10^5	-4%
chenidct	3.37×10^5	3.24×10^5	-4%
fft	4.59×10^5	3.24×10^5	-29%
fir11	5.88×10^4	4.40×10^4	-25%
cftmdl	4.48×10^5	4.24×10^5	-5%
KALMAN	5.46×10^4	6.12×10^4	12%
LowPass	1.29×10^5	1.53×10^5	18%
matmul	7.55×10^5	7.44×10^5	-1%
idct	3.14×10^5	3.62×10^5	15%
jacob	3.57×10^5	3.67×10^5	3%
Wavelet	4.45×10^5	4.27×10^5	-4%
AVG			-4%

Table 4.8: Comparison of total wirelength (3-stage)

	3-stage								
Benchmarks	BIP	M1	M2	M3	M4	M1-inc	M2-inc	M3-inc	M4-inc
chem	5.99×10^5	6.55×10^5	8.69×10^5	5.99×10^5	7.04×10^5	9%	45%	0%	17%
honda	2.49×10^5	2.29×10^5	2.93×10^5	2.50×10^5	2.40×10^5	-8%	18%	0%	-3%
dir	3.87×10^5	3.60×10^5	4.00×10^5	3.74×10^5	3.77×10^5	-7%	3%	-3%	-3%
feig_dct	2.00×10^6	1.86×10^6	2.16×10^6	1.77×10^6	1.76×10^6	-7%	8%	-12%	-12%
lee	1.96×10^5	1.96×10^5	1.99×10^5	2.01×10^5	1.99×10^5	0%	2%	3%	2%
mcm	3.53×10^5	3.53×10^5	4.30×10^5	3.71×10^5	4.04×10^5	0%	22%	5%	14%
u5ml	1.12×10^6	1.12×10^6	1.35×10^6	1.16×10^6	1.16×10^6	1%	21%	4%	4%
wang	3.01×10^5	2.83×10^5	3.09×10^5	3.07×10^5	2.96×10^5	-6%	3%	2%	-2%
pr	2.63×10^5	2.42×10^5	2.85×10^5	2.75×10^5	2.59×10^5	-8%	8%	4%	-2%
arai	9.12×10^4	8.95×10^4	1.05×10^5	9.06×10^4	9.46×10^4	-2%	15%	-1%	4%
chendct	3.08×10^5	3.53×10^5	3.19×10^5	2.88×10^5	3.18×10^5	15%	4%	-6%	4%
chenidct	3.65×10^5	3.65×10^5	3.88×10^5	3.94×10^5	3.61×10^5	0%	6%	8%	-1%
fft	4.48×10^5	4.48×10^5	4.38×10^5	4.47×10^5	4.61×10^5	0%	-2%	0%	3%
fir11	6.31×10^4	6.08×10^4	6.76×10^4	6.82×10^4	6.42×10^4	-4%	7%	8%	2%
cftmdl	4.99×10^5	4.99×10^5	5.18×10^5	4.75×10^5	4.79×10^5	0%	4%	-5%	-4%
KALMAN	6.78×10^4	6.78×10^4	6.17×10^4	5.36×10^4	6.08×10^4	0%	-9%	-21%	-10%
LowPass	1.63×10^5	1.56×10^5	1.95×10^5	1.83×10^5	1.85×10^5	-4%	20%	12%	14%
matmul	7.36×10^5	7.36×10^5	7.80×10^5	8.45×10^5	7.96×10^5	0%	6%	15%	8%
idct	3.03×10^5	3.30×10^5	3.74×10^5	3.07×10^5	3.22×10^5	9%	24%	1%	6%
jacob	3.83×10^5	3.95×10^5	4.56×10^5	3.96×10^5	4.14×10^5	3%	19%	3%	8%
Wavelet	4.22×10^5	4.22×10^5	3.97×10^5	4.47×10^5	4.56×10^5	0%	-6%	6%	8%
AVG						0%	10%	1%	3%

Table 4.9: Comparison of total wirelength (4-stage)

	4-stage								
Benchmarks	BIP	M1	M2	M3	M4	M1-inc	M2-inc	M3-inc	M4-inc
chem	6.22×10^5	6.73×10^5	9.45×10^5	6.53×10^5	6.96×10^5	8%	52%	5%	12%
honda	2.85×10^5	2.85×10^5	3.08×10^5	2.58×10^5	2.65×10^5	0%	8%	-9%	-7%
dir	3.57×10^5	3.29×10^5	3.91×10^5	3.51×10^5	3.42×10^5	-8%	10%	-2%	-4%
feig_dct	2.02×10^6	2.02×10^6	2.02×10^6	2.04×10^6	2.04×10^6	0%	0%	1%	1%
lee	1.70×10^5	1.70×10^5	1.64×10^5	1.70×10^5	1.72×10^5	0%	-4%	0%	1%
mcm	3.67×10^5	3.67×10^5	4.21×10^5	3.87×10^5	3.93×10^5	0%	15%	5%	7%
u5ml	9.93×10^5	9.93×10^5	1.32×10^6	9.93×10^5	1.05×10^6	0%	33%	0%	6%
wang	3.05×10^5	3.05×10^5	3.21×10^5	2.95×10^5	2.89×10^5	0%	5%	-3%	-5%
pr	2.35×10^5	2.44×10^5	2.65×10^5	2.66×10^5	2.57×10^5	4%	13%	13%	9%
arai	1.32×10^5	1.32×10^5	1.19×10^5	1.32×10^5	1.29×10^5	0%	-10%	0%	-2%
chendct	2.87×10^5	2.87×10^5	3.21×10^5	2.90×10^5	2.95×10^5	0%	12%	1%	3%
chenidct	3.47×10^5	3.47×10^5	4.23×10^5	3.60×10^5	3.63×10^5	0%	22%	4%	5%
fft	4.87×10^5	4.87×10^5	4.36×10^5	4.60×10^5	4.81×10^5	0%	-11%	-6%	-1%
fir11	5.87×10^4	5.87×10^4	6.28×10^4	5.87×10^4	5.36×10^4	0%	7%	0%	-9%
cftmdl	4.82×10^5	4.82×10^5	5.09×10^5	5.14×10^5	5.42×10^5	0%	5%	6%	12%
KALMAN	5.20×10^4	5.20×10^4	5.95×10^4	5.76×10^4	5.43×10^4	0%	14%	11%	4%
LowPass	1.41×10^5	1.41×10^5	1.82×10^5	1.61×10^5	1.58×10^5	0%	29%	14%	12%
matmul	7.27×10^5	7.27×10^5	7.30×10^5	8.63×10^5	7.75×10^5	0%	0%	19%	7%
idct	3.24×10^5	3.21×10^5	3.69×10^5	3.43×10^5	3.57×10^5	-1%	14%	6%	10%
jacob	3.97×10^5	3.89×10^5	4.67×10^5	4.16×10^5	3.68×10^5	-2%	17%	5%	-7%
Wavelet	4.49×10^5	4.49×10^5	4.03×10^5	4.24×10^5	4.62×10^5	0%	-10%	-6%	3%
AVG						0%	11%	3%	3%

Table 4.10: Comparison of total multiplexer inputs (2-stage)

	2-stage		
Benchmarks	BIP	Ours	inc
chem	574	621	8%
dir	228	234	3%
feig_dct	1097	1107	1%
lee	101	100	-1%
mcm	241	263	9%
u5ml	817	842	3%
wang	127	131	3%
pr	117	124	6%
arai	91	91	0%
chendct	228	243	7%
chenidct	275	278	1%
KALMAN	34	33	-3%
fft	169	153	-9%
jacob	229	234	2%
LowPass	170	195	15%
cftmdl	239	229	-4%
fir11	37	37	0%
matmul	361	390	8%
Wavelet	175	187	7%
idct	247	247	0%
AVG			3%

The usage of idle FUs is likely to increase the circuit power dissipation. Particularly, in case of FUs pipelined more than two stages, the power consumption problem is more severe than a two-stage pipelined FU, since the value stored in it switches every part of the FU by propagating down to the stage where it is used. On the other hand, the reduction of registers lowers the circuit power. To assess the overall power impact, we compare the total power of circuits generated by the bipartite matching register binding algorithm and our algorithm. Table 3.5 shows the results, derived using Design Compiler^{TM} in conjunction with signal activities based on gate level simulations. Like other experimental results, the table is divided into three sub-tables to show the power consumption for 2-, 3- and 4-stage pipelined multipliers. In case of 2-stage multiplier, it is shown that the power degradation due to the usage of FUs is mostly offset by the power reduction due to a small register

Table 4.11: Comparison of total multiplexer inputs (3-stage)

	3-stage								
Benchmarks	BIP	M1	M2	M3	M4	M1-inc	M2-inc	M3-inc	M4-inc
chem	556	573	611	560	587	3%	10%	1%	6%
dir	234	239	256	237	242	2%	9%	1%	3%
feig_dct	1050	1091	1060	1049	1061	4%	1%	0%	1%
lee	107	107	110	110	113	0%	3%	3%	6%
mcm	241	241	288	246	272	0%	20%	2%	13%
u5ml	825	835	923	828	851	1%	12%	0%	3%
wang	133	133	132	134	145	0%	-1%	1%	9%
pr	120	118	129	118	125	-2%	8%	-2%	4%
arai	83	83	83	80	85	0%	0%	-4%	2%
chendct	195	207	208	204	205	6%	7%	5%	5%
chenidct	260	264	271	267	280	2%	4%	3%	8%
KALMAN	33	33	37	34	33	0%	12%	3%	0%
fft	169	169	170	153	169	0%	1%	-9%	0%
jacob	225	225	261	226	233	0%	16%	0%	4%
LowPass	159	162	176	171	186	2%	11%	8%	17%
cftmdl	231	233	240	227	234	1%	4%	-2%	1%
fir11	44	45	46	45	46	2%	5%	2%	5%
matmul	361	361	348	356	433	0%	-4%	-1%	20%
Wavelet	175	175	169	172	207	0%	-3%	-2%	18%
idct	244	245	268	248	253	0%	10%	2%	4%
AVG						1%	6%	1%	6%

Table 4.12: Comparison of total multiplexer inputs (4-stage)

	4-stage								
Benchmarks	BIP	M1	M2	M3	M4	M1-red	M2-red	M3-red	M4-red
chem	545	552	741	546	594	1%	36%	0%	9%
dir	228	230	247	230	229	1%	8%	1%	0%
feig_dct	1009	1049	1063	1009	1047	4%	5%	0%	4%
lee	101	101	103	101	108	0%	2%	0%	7%
mcm	241	241	288	246	272	0%	20%	2%	13%
u5ml	794	794	910	794	820	0%	15%	0%	3%
wang	131	131	128	134	144	0%	-2%	2%	10%
pr	114	115	117	114	122	1%	3%	0%	7%
arai	84	84	86	84	86	0%	2%	0%	2%
chendct	172	176	190	175	187	2%	10%	2%	9%
chenidct	247	256	261	245	249	4%	6%	-1%	1%
KALMAN	31	32	35	32	34	3%	13%	3%	10%
fft	169	169	173	153	169	0%	2%	-9%	0%
jacob	226	226	270	227	229	0%	19%	0%	1%
LowPass	150	150	167	165	174	0%	11%	10%	16%
cftmdl	214	212	218	204	226	-1%	2%	-5%	6%
fir11	43	43	46	43	45	0%	7%	0%	5%
matmul	361	361	345	338	425	0%	-4%	-6%	18%
Wavelet	175	175	167	151	207	0%	-5%	-14%	18%
idct	239	235	261	236	247	-2%	9%	-1%	3%
AVG						1%	8%	-1%	7%

Table 4.13: Comparison of total power consumption (2-stage)

	2-stage		
Benchmarks	BIP	Ours	inc
chem	6.00	6.13	2%
honda	2.51	1.79	-29%
dir	2.16	2.25	4%
feig_dct	10.87	10.62	-2%
lee	1.33	1.11	-17%
mcm	2.51	2.84	13%
u5ml	6.29	6.34	1%
wang	1.25	1.34	7%
pr	1.80	1.80	0%
arai	0.84	0.71	-16%
chendct	1.80	1.84	2%
chenidct	1.57	1.70	8%
fft	2.91	2.33	-20%
fir11	0.51	0.42	-18%
cftmdl	2.68	3.10	15%
KALMAN	0.55	0.46	-16%
LowPass	1.28	1.18	-8%
matmul	9.29	6.35	-32%
idct	2.28	2.56	12%
jacob	1.73	1.83	6%
Wavelet	3.10	3.20	3%
AVG			-4%

count. On the average, the overall circuit power is even decreased by 4%. However, in case of 3- and 4-stage multiplier, the effect is a little bit different. When Method 1 is used, there is no overhead of power consumption shown in the seventh column of the second and third sub-tables. However, when Method 2, 3 and 4 are used, power consumption cannot be negligible. Thus, when deeply pipelined FUs are used in conjunction with aggressive register saving techniques such as Method 2, 3 and 4, the saving is achieved at the expense of power consumption overhead.

Table 4.14: Comparison of total power consumption (3-stage)

	3-stage								
Benchmarks	BIP	M1	M2	M3	M4	M1-inc	M2-inc	M3-inc	M4-inc
chem	5.90	5.82	7.31	5.68	5.96	1%	-24%	4%	-1%
honda	1.65	1.51	1.82	1.64	1.82	8%	-10%	1%	-10%
dir	2.58	2.37	2.43	2.37	2.56	8%	6%	8%	1%
feig_dct	11.01	12.40	11.04	10.18	9.98	-13%	0%	8%	9%
lee	1.16	1.16	0.95	1.21	0.97	0%	18%	-4%	17%
mcm	2.48	2.48	3.65	2.69	2.63	0%	-47%	-8%	-6%
u5ml	6.32	6.41	7.42	6.20	6.44	-1%	-17%	2%	-2%
wang	1.20	1.11	1.41	1.22	1.33	8%	-17%	-2%	-11%
pr	1.38	1.29	1.64	1.54	1.30	6%	-19%	-12%	6%
arai	0.64	0.64	0.73	0.77	0.65	0%	-14%	-19%	-1%
chendct	1.63	1.72	1.52	1.39	1.53	-6%	7%	14%	6%
chenidct	1.75	1.75	1.95	2.21	1.50	0%	-11%	-26%	14%
fft	1.91	1.91	2.60	3.73	1.96	0%	-36%	-95%	-2%
fir11	0.56	0.62	0.61	0.56	0.57	-10%	-9%	1%	-1%
cftmdl	2.28	2.28	3.98	2.62	3.07	0%	-75%	-15%	-34%
KALMAN	0.43	0.43	0.55	0.43	0.42	0%	-28%	1%	4%
LowPass	1.35	1.39	2.01	1.38	1.59	-2%	-48%	-2%	-17%
matmul	6.88	6.88	8.52	8.57	8.59	0%	-24%	-25%	-25%
idct	2.14	2.12	2.59	2.09	2.09	1%	-21%	2%	2%
jacob	1.67	1.70	1.99	2.04	1.91	-2%	-19%	-22%	-14%
Wavelet	3.36	3.36	4.50	3.81	3.81	0%	-34%	-13%	-13%
AVG						0%	20%	10%	4%

Table 4.15: Comparison of total power consumption (4-stage)

	4-stage								
Benchmarks	BIP	M1	M2	M3	M4	M1-inc	M2-inc	M3-inc	M4-inc
chem	5.55	5.73	7.75	6.03	6.24	-3%	40%	-9%	-13%
honda	1.82	1.82	2.07	1.68	1.72	0%	-14%	8%	6%
dir	2.21	2.23	2.46	2.21	2.18	-1%	-11%	0%	1%
feig_dct	9.15	9.15	10.07	9.21	9.05	0%	-10%	-1%	1%
lee	0.90	0.90	0.94	0.90	0.93	0%	-5%	0%	-4%
mcm	2.31	2.31	2.77	2.57	2.41	0%	-20%	-11%	-4%
u5ml	5.98	5.98	7.29	5.98	6.26	0%	-22%	0%	-5%
wang	1.14	1.14	1.22	1.08	1.19	0%	-8%	5%	-5%
pr	1.16	1.18	1.30	1.30	1.29	-2%	-12%	-12%	-11%
arai	0.80	0.80	0.68	0.80	0.77	0%	15%	0%	5%
chendct	1.48	1.48	1.47	1.42	1.31	0%	1%	4%	12%
chenidct	1.52	1.52	1.76	1.63	1.53	0%	-15%	-7%	-1%
fft	1.89	1.89	2.69	3.47	2.31	0%	-42%	-83%	-22%
fir11	0.52	0.52	0.57	0.52	0.52	0%	-9%	0%	0%
cftmdl	2.84	2.84	2.78	4.23	3.56	0%	2%	-49%	-25%
KALMAN	0.41	0.41	0.56	0.52	0.47	0%	-34%	-26%	-13%
LowPass	1.22	1.22	1.65	1.65	1.58	0%	-35%	-35%	-29%
matmul	5.51	5.51	5.75	7.91	7.17	0%	-4%	-44%	-30%
idct	2.15	2.14	2.43	2.26	2.31	0%	-13%	-5%	-8%
jacob	1.67	1.69	2.16	1.71	1.70	-1%	-29%	-2%	-2%
Wavelet	2.67	2.67	2.74	3.93	2.90	0%	-2%	-47%	-8%
AVG						0%	15%	15%	7%

4.6 Conclusion

In this chapter, we present a new register binding algorithm for register reduction. Our scheme uses the internal registers of pipelined FUs when the FUs are idle, reducing the number of dedicated registers. In addition, our scheme takes into account the interconnect topology among FUs and dedicated registers and limit the increase of the global interconnect length. We proposed four storing methods based on whether variables are stored and used via FU input/output ports or input/output of internal registers. Our experimental results have shown that our algorithm reduces the number of registers by the amount ranging from 2% to 44% on average, depending on which storing methods are used. In addition, the experimental results show that our scheme is also effective to reduce clock skew. The average reduction of rising (falling) clock skew is ranging from 7% (8%) to 33% (30%), respectively. We also have shown the overhead our scheme causes. The average increase of wirelength and power consumption is ranging from -4% to 10% and from -4% to 20%, respectively, depending on the storing methods.

Chapter 5

Conclusion

Optimization techniques during high level synthesis procedure are often preferred since design decisions at early stages of a design flow are believed to have a large impact on design quality. In this dissertation, we present three high-level synthesis schemes to improve the power, speed and reliability of deep submicron VLSI systems.

In this dissertation, we present two global interconnect optimization algorithms and a register reduction algorithm during high level synthesis. Specifically, we first propose simultaneous functional unit and register binding algorithm for global interconnect optimization. Our main goal is to maximize physical interconnect sharing among data transfers. We observed that flow dependencies and common inputs can be used to measure the interconnect sharing. Based on the observation, we formulate the functional unit and register binding problem as the longest path problem in the compatibility graph. Our experimental results have shown that our algorithm reduces the number of multiplexer inputs by more than 20% on average in comparison to the previously proposed binding algorithms for interconnect optimization. Moreover, our scheme achieves a total wirelength reduction by 17.29% on average at the cost of slight FU and register increases.

We then propose interconnect assignment algorithm to minimize total wirelength of global interconnects. In general, data transfer is realized by using interconnects, i.e., metal wires. Other forms of signal propagation paths can also be used, which results in the reduction of total interconnect wirelength. We observed that not all functional units are in operation for every clock cycle and they have circuit path from their input ports to output ports. Thus, if a functional unit is idle in a certain clock cycle, it can be used for data transfer during the cycle. The challenges are how to find idle functional units that reduce global interconnects without inducing power and timing overhead. We formulate the interconnect assignment problem as maximum-flow-minimum-cost network flow problem. In addition, we propose a modified shortest path algorithm to solve the problem. Experimental results show that our algorithm reduces the total wirelength of global interconnects by 8.5% and the power consumption by 4.8% without introducing any timing violations.

Finally, we present a register binding algorithm for register reduction. Our scheme uses internal registers in pipelined functional units. Since not all functional units operate in every clock cycle, the internal registers of functional units can be used for storage of variables, when the functional units are idle. We discuss four storing methods using idle pipelined functional units and analyze the impact on register cost and overhead of each method. The register binding problem is formulated as the minimum-cost bipartite matching problem. We match the lifetimes of variables to time durations during which idle functional units can maintain values of variables. We limit the increase of multiplexer inputs for global interconnect optimization. Our experimental results have shown that our algorithm reduces the number of registers by the amount ranging from 2% to 44% on average, depending on which storing methods are used.

Bibliography

[1] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI.* Addison Wesley, 1990.

[2] Jeffrey A. Davis, Raguraman Venkatesan, Alain Kaloyeros, Michael Beylansky, Shukri J. Souri, Kaustav Banerjee, Krishna C. Saraswat, Arifur Rahman, Rafael Reif, and James D. Meindl. Interconnect Limits on Gigascale Integration (GSI) in the 21st Century. *Proceedings of the IEEE*, 2001.

[3] Nir Magen, Avinoam Kolodny, Uri Weiser, and Nachum Shamir. Interconnect-Power Dissipation in a Microprocessor. In *Proceedings of International Workshop on System-Level Interconnect Prediction*, February 2004.

[4] Kaustav Banerjee and Amit Mehrotra. Global (Interconnect) Warming. *IEEE Circuit & Devices*, 2001.

[5] Jan M. Rabaey, Anantha Chandrakasan, and Borivoje Nikolić. *DIGITAL INTEGRATED CIRCUITS.* Prentice Hall, 2003.

[6] Amir H. Farrahi, Gustavo E. Téllez, and Majid Sarrafzadeh. Memory Segmentation to Exploit Sleep Mode Operation. In *Proceedings of DAC*, June 1995.

[7] Paul E. Gronowski, William J. Bowhill, Ronald P. Preston, Michael K. Gowan, and Randy L. Allmon. High-Performance Microprocessor Design. *Journal of Solid-State Circuits*, 1998.

[8] Taemin Kim and Xun Liu. Compatibility Path Based Binding Algorithm for Interconnect Reduction in High Level Synthesis. In *Proceedings of ICCAD*, November 2007.

[9] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, and Yu-Chin Hsu. Data Path Allocation Based on Bipartite Weighted Matching. In *Proceedings of DAC*, June 1990.

[10] Taemin Kim and Xun Liu. Better Than Optimum? Register Reduction Using Idle Pipelined Functional Units. In *Proceedings of GLSVLSI*, May 2009.

[11] Fadi J. Kurdahi and Alice C. Parker. REAL:A Program for REgister ALlocation. In *Proceedings of DAC*, June 1987.

[12] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on High-Level Synthesis. In *Proceedings of DAC*, June 1988.

[13] Michael C. McFarland, Alice C. Parker, and Raul Camposano. The High-Level Synthesis of Digital Systems. *Proceedings of the IEEE*, 1990.

[14] Raul Camposano. From behavior to structure: High-level synthesis. *IEEE Design & Test of Computers*, 1990.

[15] R. Camposano, L.F. Saunders, and R.M. Tabet. VHDL as Input for High-Level Synthesis. *IEEE Design & Test of Computers*, 1991.

[16] Daniel Gajski, Allen Wu, Nikil Dutt, and Steve Lin. *HIGH-LEVEL SYNTHESIS:Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[17] Philippe Coussy and Adam Morawiec. *High-Level Synthesis from Algorithm to Digital Circuit.* Springer, 2008.

[18] Deming Chen, Jason Cong, and Yiping Fan. Low-Power High-Level Synthesis for FPGA Architectures. In *Proceedings of ISLPED*, August 2003.

[19] Deming Chen and Jason Cong. Register Binding and Port Assignment for Multiplexer Optimization. In *Proceedings of ASPDAC*, January 2004.

[20] Jason Cong, Yiping Fan, and Wei Jian. Platform-Based Resource Binding Using a Distributed Register-File Microarchitecture. In *Proceedings of ICCAD*, November 2006.

[21] Jason Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang. Architecture and Synthesis for On-Chip Multicycle Communication. *IEEE Transactions on CAD of integrated circuits and systems*, 2004.

[22] Jason Cong, Yiping Fan, Xun Yang, and Zhiru Zhang. Architecture and Synthesis for Multi-Cycle Communication. In *Proceedings of ISPD*, April 2003.

[23] Jason Cong and Junjuan Xu. Simultaneous FU and Register Binding Based on Network Flow Method. In *Proceedings of DATE*, March 2008.

[24] Renu Mehra, Lisa M. Guerra, and Jan M. Rabaey. Low-Power Architectural Synthesis and the Impact of Exploiting Locality. *Journal of VLSI Signal Processing*, 1996.

[25] Renu Mehra, Lisa M. Guerra, and Jan M. Rabaey. A Partitioning Scheme for Optimizing Interconnect Power. *IEEE Journal of Solid-State Circuits*, 1997.

[26] Xun Liu and Marios C. Papaefthymiou. Design of a 20-Mb/s 256-State Viterbi Decoder. *IEEE Transactions on VLSI Systems*, 2003.

[27] Jen-Pin Weng and Alice C. Parker. 3D Scheduling:High-Level Synthesis with Floorplanning. In *Proceedings of DAC*, June 1991.

[28] Daehong Kim, Jinyong Jung, Sunghyun Lee, Jinhwan Jeon, and Kiyoung Choi. Behavior-to-Placed RTL Synthesis with Performance-Driven Placement. In *Proceedings of ICCAD*, November 2001.

[29] Junhyung Um, Jae hoon Kim, and Taewhan Kim. Layout-Driven Resource Sharing in High-Level Synthesis. In *Proceedings of ICCAD*, November 2002.

[30] Adam Kaplan, Philip Brisk, and Ryan Kastner. Data Communication Estimation and Reduction for Reconfigurable Systems. In *Proceedings of DAC*, June 2003.

[31] Ryan Kastner, Wenrui Gong, Xin Hao, Forrest Brewer, Adam Kaplan, Philip Brisk, and Majid Sarrafzadeh. Layout Driven Data Communication Optimization for High Level Synthesis. In *Proceedings of DATE*, March 2006.

[32] L. Stok. INTERCONNECT OPTIMIZATION DURING DATA PATH ALLOCATION. In *Proceedings of EURO-DAC*, September 1990.

[33] C. A. Papachristou and H.Konuk. A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm. In *Proceedings of DAC*, June 1990.

[34] Minjoong Rim, Rajiv Jain, and Renato De Leone. Optimal Allocation and Binding in High-Level Synthesis. In *Proceedings of DAC*, June 1992.

[35] Taewhan Kim and C. L. Liu. An Integrated Data Path Synthesis Algorithm Based on Network Flow Method. In *Proceedings of CICC*, May 1995.

[36] Annie Avakian and Iyad Ouaiss. Optimizing Register Binding in FPGAs Using Simulated Annealing. In *Proceedings of ReConFig*, September 2005.

[37] Imtiaz Ahmad and C. Y. Roger Chen. Post-Processor For Data Path Synthesis Using Multiport Memories. In *Proceedings of ICCAD*, November 1991.

[38] Taewhan Kim and C. L. Liu. Utilization of Multiport Memories in Data Path Synthesis. In *Proceedings of DAC*, June 1993.

[39] Hassan Al Atat and Iyad Ouaiss. Register Binding for FPGAs with Embedded Memory. In *Proceedings of FCCM*, April 2004.

[40] Barry Pangrle. On the Complexity of Connectivity Binding. *IEEE Transactions on Computer Aided Design*, 1991.

[41] Giovanni De Micheli. *SYNTHESIS AND OPTIMIZATION OF DIGITAL CIRCUITS*. McGraw Hill, 1994.

[42] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *INTRODUCTION TO ALGORITHMS,2nd Edition*. The MIT Press, 2001.

[43] M. B. Srivastava and M. Potkonjak. Optimum and Heuristic Transformation Techniques for Simultaneous Optimization of Latency and Throughput. *IEEE Transactions on VLSI Systems*, 1995.

[44] N. Dutt and C. Ramchandran. Benchmarks for the 1992 High Level Synthesis Workshop. *Technical Report 92-107, University of California, Irvine*, 1992.

[45] P. Panda and N. Dutt. 1995 high level synthesis design repository. In *Proceedings of International Symposium on System Synthesis*, September 1995.

[46] Pierre G. Paulin and John P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 1989.

[47] Miodrag Potkonjak and Sjit Dey. Optimizing Resource Utilization and Testability Using Hot Potato Techniques. In *Proceedings of DAC*, June 1994.

[48] Jennifer L. Wong, Miodrag Potkonjak, and Sujit Dey. Optimizing Designs Using the Addition of Deflection Operations. *IEEE Transactions on Computer-Aided Design*, 2004.

[49] Dirk Herrmann and Rolf Ernst. Improved Interconnect Sharing by Identity Operation Insertion. In *Proceedings of ICCAD*, November 1999.

[50] Hyuk-Jae Jang and Barry M. Pangrle. A Grid-Based Approach for Connectivity Binding with Geometric Costs. In *Proceedings of ICCAD*, November 1993.

[51] Lin Zhong and Niraj K. Jha. Interconnect-aware High-level Synthesis for Low Power. In *Proceedings of ICCAD*, November 2002.

[52] Lin Zhong, Jiong Luo, Yunsi Fei, and Niraj K. Jha. Register Binding based Power Management for High-level Synthesis of Control Flow Intensive Behviors. In *Proceedings of ICCD*, September 2002.

[53] Jiong Luo, Lin Zhong, Yunsi Fei, and Niraj K. Jha. Register Binding-Based RTL Power Management for Control-Flow Intensive Designs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 2004.

[54] Deniz Dal and Nazanin Mansouri. A High-Level Register Optimization Technique for Minimizing Leakage and Dynamic Power. In *Proceedings of GLSVLSI*, March 2007.

[55] Shih-Hsu Huang, Chun-Hua Cheng, Yow-Tyng Nieh, and Wei-Chieh Yu. Register Binding for Clock Period Minimization. In *Proceedings of DAC*, July 2006.

[56] C. J. Tseng and D. P. Siewiorek. Automated Synthesis of Data paths on Digital Systems. *IEEE Transactions on CAD of integrated circuits and systems*, 1986.

[57] Noureddine Chabini and Wayne Wolf. Register Binding Guided By The Size Of Variables. In *Proceedings of ICCD*, October 2007.

[58] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.