IPSJ Transactions on System LSI Design Methodology Vol. 1 78–90 (Aug. 2008)

Regular Paper

Floorplan-Driven High-Level Synthesis for Distributed/Shared-Register Architectures

Akira Ohchi,[†1] Shunitsu Kohara,[†1]
Nozomu Togawa,[†1] Masao Yanagisawa[†1]
and Tatsuo Ohtsuki[†1]

In this paper, we propose a high-level synthesis method targeting distributed/shared-register architectures. Our method repeats (1) scheduling/FU binding, (2) register allocation, (3) register binding, and (4) module placement. By feeding back floorplan information from (4) to (1), our method obtains a distributed/shared-register architecture where its scheduling/binding as well as floorplaning are simultaneously optimized. Experimental results show that the area is decreased by 13.2% while maintaining the performance of the circuit equal with that using distributed-register architectures.

1. Introduction

It is very effective to make use of high-level synthesis methodologies to improve the productivity of LSI design, in which we can automatically synthesize a hardware architecture from an abstract behavior description. Since conventional high-level synthesis deals with floorplan as its postprocessing, informations on placement and interconnection among modules (i.e., functional unit, register, controller, and multiplexer) are not able to be considered in a high-level synthesis stage. In recent years, as device feature size decreases, interconnection delay becomes the dominant factor of total delay, and it is predicted that this trend will continue over the next few years. This means that it is necessary to deal with floorplan informations such as placement and interconnection delay even in a high-level synthesis stage.

Several researchers have considered floorplaning in high-level synthesis [2),3),12),15)]. These approaches reduce clock period by decreasing wire delay on a critical path by considering module floorplaning. In Refs. 4), 7), 11), 16), both wiring length and power consumption are decreased. But execution time of a given application running on a synthesized hardware is not referred.

In Refs. 5), 6), they consider the situation in which wire delay becomes a bottleneck, and propose a distributed-register architecture. This architecture places local registers only for the functional unit close to it. The wire length between the functional unit and the register is reduced. The clock period is able to be occupied almost entirely by the delay of the functional unit. Clock period is shortened and the execution time of a given application using this architecture can be decreased. Furthermore, if there is data transfer between two functional units placed apart, register-to-register data transfer can be used. However, the number of registers increases because this architecture requires local registers placed for each functional unit.

Another architecture, RDR (Regular Distributed Register) is proposed in Ref. 1). This architecture divides a chip into a uniform size, and arranges functional units, a register file, and a controller in one island. Because a chip is divided into a constant size, wire delay may become smaller than a conventional shared-register architecture, and its design is simplified. However area overhead is increased.

In Ref. 13), a high-level synthesis algorithm for distributed-register architectures is proposed in which it can deal with a control-data flow graph with condition structures and also it gives a detailed algorithm for register binding.

Although a distributed-register architecture has an advantage of reducing critical path delays by placing local registers close to a functional unit, the number of local registers will be increased since local registers cannot be shared by multiple functional units. It is desirable that “shared registers” will be used in a non-critical path between functional units. A mixture of distributed-register architecture and shared-register architecture is strongly required in next-generation high-level synthesis.

Based on the above discussions, this paper proposes a high-level synthesis method for distributed/shared-register architecture, which decreases area while maintaining performance of a distributed-register architecture. The proposed method can reflect floorplan information in scheduling by using feedback synthe-

[†1] School of Fundamental Science and Engineering, Waseda University

sis flow. In addition to that, it can automatically select local register or shared register for each functional unit based on a signal flow between functional units. Thus it finally synthesizes an RT-level description with its floorplaning assuming a distributed/shared-register architecture.

This paper is organized as follows. Section 2 introduces a distributed/shared-register architecture. Section 3 proposes a new high-level synthesis flow. Section 4 proposes details of a scheduling/FU binding algorithm. In Section 5, we propose a register allocation algorithm. In Section 6, we propose a register binding algorithm. Section 8 presents and discusses experimental results. We conclude this paper in the last section.

2. Distributed/Shared-Register Architecture

Our high-level synthesis targets a distributed/shared-register architecture which has a combination of a distributed-register architecture and a shared-register architecture.

A distributed-register architecture is an architecture where each FU (functional unit) has local registers. The local register is placed near its FU, and then interconnection delay is reduced because the wire length from an FU to a local register is reduced. The FU uses register-to-register data transfer when data is transferred to distant FUs. In this case, since all the clock cycle time can be used for data transfer, a clock cycle can go up. However, using complete distributed-register architecture would increase local registers significantly, which is far away from realistic design.

In order to reduce the number of registers without degrading performance, we introduce shared registers. In this paper, we define our architecture model which uses local register and a single shared-registers group. A distributed/shared-register architecture has a single shared-register group and several local registers as shown in **Fig. 1**. A shared-register group has its own controller in it and FUs also have their own controller. If "FU delay" + "wire delay between the FU and the shared-register group" exceeded a given clock period constraint, the FU needs local registers. Otherwise the FU use shared-registers. This architecture can reduce the number of registers while maintaining the performance of a synthesized hardware equal to that using a distributed-register architecture.

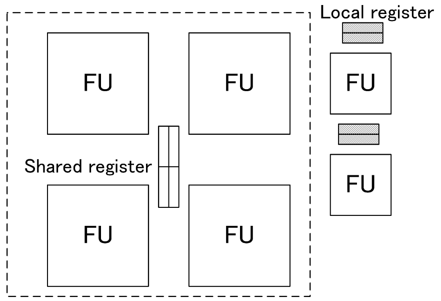


Fig. 1 Distributed/shared-register architecture.

3. Synthesis Flow

In this section, we first define our high-level synthesis problem with floorplaning, and then we propose a new synthesis flow targeting a distributed/shared-register architecture.

3.1 Problem Definition

A control-data flow graph (CDFG) $G(V, E)$ is a directed graph, where a node set V is an operation node set N_o and a branching control node set N_c (beginning and termination of condition branches), and an edge set E is a data-flow edge set E_d and a control-flow edge E_c set. **Figure 2** shows an example of our CDFG. A circle node represents operation, such as '+' and '<'. A fork node, depicted by an upward triangle node, indicates the beginning of a condition branch. A join node, depicted by a downward triangle node, indicates the termination of a condition branch. A solid line represents a data-flow. A dotted line represents a control dependency. Operation nodes are associated with their CV (Condition Vector)[14]. CVs show the execution conditions of resources.

Our high-level synthesis problem is, for a given CDFG and the number of FUs, to assign each operation node to a Control Step and an FU, to assign each variable to a register, and to arrange modules, under clock period constraint. The first objective is to minimize the execution time of the application given by the CDFG. The second objective is to minimize the chip area of a synthesized hardware.

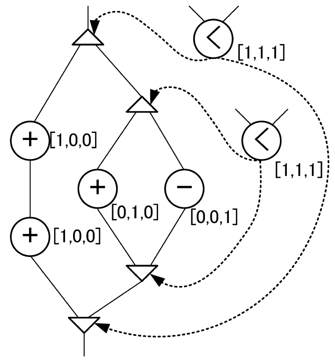


Fig. 2 An example of CDFG.

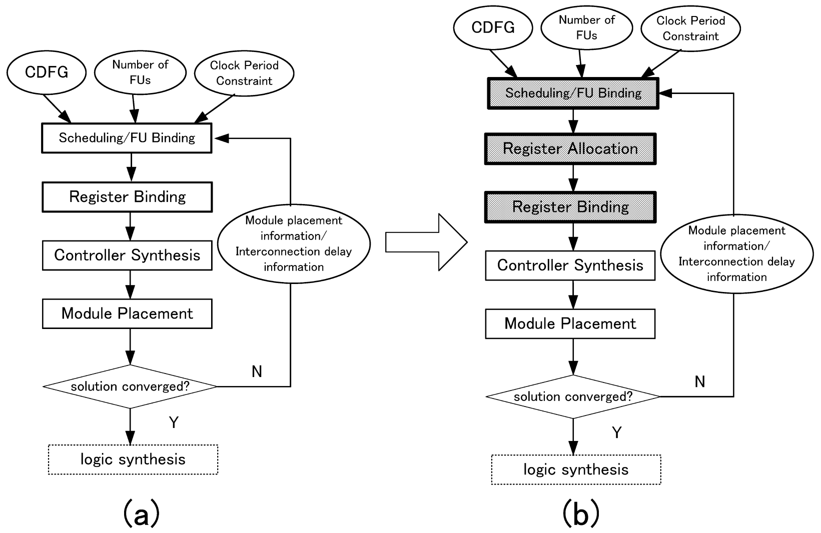


Fig. 3 High-level synthesis flow. (a) The flow proposed in Ref. 13). (b) The improved flow proposed in this paper.

3.2 Synthesis Flow

Figure 3 (a) shows the synthesis flow proposed in Ref. 13) which targets distributed-register architecture. The method proposed in Ref. 13) uses register-to-register data transfer and interconnection delay information is needed at the scheduling process. Thus, it feeds back the interconnection delay and module placement information to its next iteration.

However, if we consider a distributed/shared-register architecture, we must determine which register type (local register or shared register) is used for each functional unit. We improved the synthesis flow and showed it in Fig. 3 (b). We determine a register type in “register allocation” based on placement information, for each FU after the scheduling process. Then we determine which variable is bound to which register according to the register allocation process.

Moreover, we extend the scheduling/FU binding algorithm so that it can deal with local registers and shared registers and also we propose a new register binding algorithm which takes into account both shared registers and local registers.

We roughly summarize each step in the flow as follows:

In the scheduling/FU binding process, we use an algorithm based on CVLS (Condition Vector based List Scheduling) [14] and it executes scheduling and FU binding simultaneously.

In the register allocation process, register type that each FU uses is determined referring to the placement information from the floorplan process done in the previous iteration to satisfy clock period constraint as in the scheduling/FU binding stage. At the initial step of the synthesis flow, all FUs are assumed to use shared registers because interconnect delay information cannot be referred.

In the register binding process, to minimize the number of total registers, variables extracted from a scheduled CDFG are bound to shared registers or local registers.

In the controller synthesis process, we use Synopsys Design Compiler to obtain control circuits.

In the floorplan process, module placement is optimized by using simulated annealing with Sequence-pair [9] representation. The cost function is

$$cost = \alpha A + \beta W + \gamma V$$

where A is the rectangle area which includes all the modules (dead space may be included), W is the wire length, V is the sum of violation of clock period constraint, and α, β, γ are parameters. Initial temperature T_i in floorplan at the i-th iteration of the synthesis flow is computed as

$T_i = T_{i-1}/K$
where K is also a parameter and set to be $K \geq 1$. The initial solution of floorplan at each iteration is the floorplan solution represented by its Sequence-pair of the last iteration. Note that in this floorplanning, we have FU modules, a shared-register module, and a controller module, where each FU module includes local registers, MUXs, and an FU itself. Thus if the number of local registers changes in each iteration, we can use Sequence-pair representation of the last iteration as the initial solution of the current iteration. By repeating this iteration, module placement becomes fixed and the solution will converge⋆1.

We repeat the above mentioned process until the convergence condition is met. The convergence condition is that the solution of floorplan and its area are equal to these of the last iteration. If the count of iterations exceeds M, we consider that the solution doesn't converge and the output best solution⋆2.

In the rest of the paper, we propose a scheduling/FU binding algorithm in Section 4, a register allocation algorithm in Section 5, and a register binding algorithm in Section 6.

4. Scheduling/FU Binding

We first define out scheduling/FU binding problem. For given a CDFG, the number of FUs, and interconnection delays between modules, a scheduling/FU binding problem is to assign each operation node to an FU and a Control Step (CS) under clock period constraint. The objective is to minimize the number of Control Steps.

4.1 The Basic Algorithm

We extend the scheduling/FU binding algorithm proposed in Ref. 13) so that it can deal with a distributed/shared-register architecture. This original algorithm is based on CVLS (Condition Vector based List Scheduling) and considers multi-cycle interconnection delay. First, it calculates minimum clock cycle counts to transfer data between FUs. Second, it performs scheduling/FU binding from start nodes to end nodes, based on the priority function which is calculated by the critical path length. Since the original algorithm in Ref. 13) is based on CVLS, it can deal with control flow including conditional branches as well as data flow.

⋆1 In the experiment in Section 8, we set $\alpha = 1$, $\beta = 1$, $\gamma = 10{,}000$, and $K = 10$.
⋆2 M is the constant number. In the experiment, we set $M = 30$.

4.2 The Proposed Scheduling/FU Binding Algorithm for a Distributed/Shared-Register Architecture

In order to extend the scheduling/FU binding algorithm proposed in Ref. 13) to a distributed/shared-register architecture, we first propose a *data transfer table*. Then we propose our algorithm for a distributed/shared-register architecture. Note that, in the beginning of our synthesis flow, we assume that all the FUs use shared registers. In a scheduling/FU binding process in a subsequent iteration, we use shared register or local register for each FU which is determined by the previous iteration.

4.2.1 Data Transfer Table

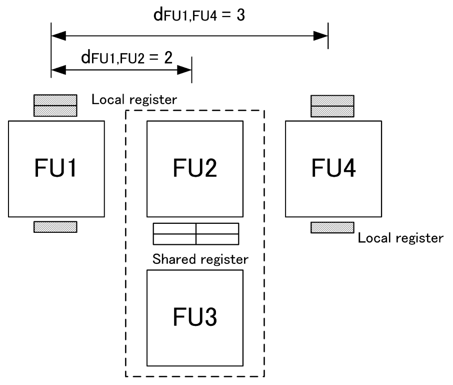
The number of clock cycles to transfer data between FUs is calculated based on the placement result of the previous iteration.

Let f_i and f_j be two functional units. Let $d_{i,j}$ be an interconnection delay between f_i and f_j. Let $rd_{i,j}$ be an interconnection delay between the register which f_i uses and the register which f_j uses. An interconnection delay is assumed to be proportional to the square of the Manhattan distance between two FUs/registers. Let t_{reg} and t_{CLK} be a register read/write time and clock period, respectively. When d_{f_i} is delay of FU f_i, $Slack_i$ can be defined as $(t_{CLK} - t_{reg} - d_{f_i})$.

Further, a required clock cycle $id_{i,j}$ for the data transfer from an FU f_i to an FU f_j can be expressed as follows: if both f_i and f_j use shared registers, $id_{i,j}$ is 0. Otherwise $id_{i,j}$ is defined as

$$id_{i,j} = \begin{cases} 0 & (Slack_i \geq d_{i,j}) \\ \lceil (rd_{i,j} + t_{reg})/t_{CLK} \rceil & (Slack_i < d_{i,j}) \end{cases}$$

For example, if an FU f_i uses a shared register and an FU f_j uses a local register, $rd_{i,j}$ is $c \times l_{share,j} \times l_{share,j}$, where c is a constant and $l_{share,j}$ is the Manhattan distance between a shared register and f_j. The distance between modules can be calculated by the positions of modules, which are determined in the module placement process in the previous iteration. $id_{i,j}$ constructs a data transfer table for any two FUs f_i and f_j.



(a) Placement of (i−1)-th iteration

	FU1	FU2	FU3	FU4
FU1	0	1	1	2
FU2	1	0	0	1
FU3	1	0	0	1
FU4	2	1	1	0

(b) Data transfer table of i-th iteration

Fig. 4 An example of a data transfer table. We assume that $t_{CLK} = 2$, $t_{reg} = 0$, $d_{FU_X} = 1$ $(X = 1, \cdots, 4)$. When we obtain a placement result as shown in (a) in $(i-1)$-th iteration, then we can obtain a data transfer table as shown in (b) in i-th iteration.

Figure 4 shows an example of a data transfer table. The first column in Fig. 4 (b) shows source FUs. The first row in Fig. 4 (b) shows destination FUs. We assume that $t_{CLK} = 2$, $t_{reg} = 0$, $d_{FU_X} = 1$ $(X = 1, \cdots, 4)$. When we obtain a placement result as shown in Fig. 4 (a) in $(i-1)$-th iteration, then we can obtain a data transfer table as shown in Fig. 4 (b) in i-th iteration. Since the FU2 and FU3 use shared registers, the required clock cycle for data transfer between these two FUs, $id_{FU2,FU3}$ and $id_{FU3,FU2}$, become 0. As in Fig. 4, we assume that the interconnection delay between FU1 and FU4 $d_{FU1,FU4}$ is 3. We also assume that the interconnection delay between register used FU1 and register used FU4 $rd_{FU1,FU4}$ is 3. Thus the required clock cycle for data transfer from $FU1$ to $FU4$, $id_{FU1,FU4}$ is computed as $\lceil 3/2 \rceil = 2$, which means that we need two clock cycles to transfer data from $FU1$ to $FU4$. In the same way, $id_{FU1,FU2}$ is computed as $\lceil 2/2 \rceil = 1$.

As mentioned above, we set $id_{i,j}$ to be 0 if both an FU f_i and an FU f_j use shared registers. This means that we assume that f_i, f_j, and shared registers are close enough and then data transfer from f_i to f_j can be done within one clock cycle. If this assumption is violated, we change the register type of f_i or f_j in the register allocation process. By introducing this strategy, FUs on the critical path tend to use local registers and those not on the critical path tend to use shared registers.

Note that, at the initial step of the synthesis flow, interconnect delay information cannot be referred. Interconnect delay between any two modules assumed to be 0.

4.2.2 The Algorithm

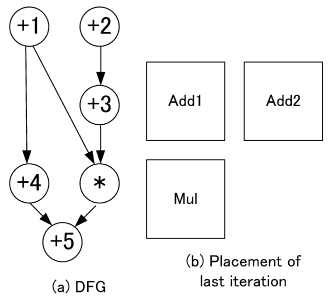
The algorithm is based on CVLS and employs the strategy that it picks up the most "critical node" one by one and then assigns it to a Control Step and a functional unit so that the total number of expected Control Steps is minimized.

Critical path length of a CDFG is defined as follows. Critical path length $cp(n_i, f_j)$ refers to the longest path length from each node n_i to any end node if n_i is assigned to an FU f_j. Let n_e be one of the end nodes in a given CDFG. If n_e is bound to an FU f_j, $cp(n_e, f_j)$ is defined as c_{f_j}, where c_{f_j} is the number of clock cycles to execute FU f_j. In recursive manner from an end node to a start node, we can calculate $cp(n_i, f_j)$ for each node n_i and each FU f_j as:

$$cp(n_i, f_j) = c_{f_j} + \max_{n_k \in succ(n_i)} \left[\min_{f_l \in FUs} (id_{j,l} + cp(n_k, f_l)) \right]$$

where $succ(n_i)$ is a set of immediate successors of n_i. In the above expression, $id_{j,l}$ is given by our data transfer table.

We show how to calculate critical path length using the example DFG as shown in **Fig. 5** (a). Assume that placement of the last iteration and its data transfer table are given as shown in Figs. 5 (b) and (c) and all FUs are able to be executed in one cycle. Then we start the scheduling/FU binding of the current iteration as follows: Since "+5" is an end node, both $cp(+5, Add1)$ and $cp(+5, Add2)$ are 1.



	Add1	Add2	Mul
Add1	0	1	1
Add2	1	0	2
Mul	1	2	0

(c) Data transfer table

Fig. 5 An example of scheduling/FU binding.

As for $cp(*, Mul)$, if the node "+5" is bound to Add1, $cp(*, Mul)$ is calculated as $c_{Mul} + id_{Mul,Add1} + cp(+5, Add1) = 1 + 1 + 1 = 3$. If the node "+5" is bound to Add2, $cp(*, Mul)$ is calculated as $c_{Mul} + id_{Mul,Add2} + cp(+5, Add2) = 1+2+1 = 4$. Since the number of Control Steps from the node "$*$" to the end node is minimized when the node "+5" is bound to Add1, $cp(*, Mul)$ becomes 3. As for the nodes "+2", "+3" and "+4", their critical path lengths are similarly calculated as in **Table 1**. As for $cp(+1, Add1)$, the path length of path("+1"→"+4"→"+5") is calculated as $c_{Add1} + id_{Add1,Add1} + cp(+4, Add1) = 3$ and the path length of path("+1"→"$*$"→"+5") is calculated as $c_{Add1} + id_{Add1,Mul} + cp(*, Mul) = 5$. The minimum number of Control Steps to execute from the node "+1" to the end node is expected to be 5 when the node"+1" is bound to Add1. Then $cp(+1, Add1)$ is calculated as 5.

We first calculate $cp(n_i, f_j)$ for each node n_i and each FU f_j from an end node to a start node. For each node n_i, $priority(n_i)$ is set to be

$$priority(n_i) = \min_{k \in FUs} cp(n_i, f_k)$$

as the priority function of CVLS.

Table 1 An example of critical path lengths. Each value in this Table shows $cp(n_i, f_j)$ for node n_i and FU f_j. Each underlined value shows $priority(n_i)$ for node n_i.

FU / node	Add1	Add2	Mul
+1	$\underline{5}$	6	–
+2	$\underline{6}$	7	–
+3	$\underline{5}$	6	–
+4	$\underline{2}$	2	–
+5	$\underline{1}$	1	–
$*$	–	–	$\underline{3}$

Step 1. Calculate $cp(n_i, f_j)$ for each node n_i and each FU f_j.
Step 2. Calculate $priority(n_i)$ for each node n_i. Control Step $k \leftarrow 0$.
Step 3. If all the nodes are scheduled, finish.
Step 4. $k \leftarrow k + 1$, and make the ready list L.
Step 5. Pick up the node $n_i \in L$ whose $priority(n_i)$ is maximum. Let f_j be the FU whose $l(n_i, f_j)$ is the minimum. If f_j is available at Control Step k, then assign n_i to Control Step k, bind it to FU f_j, and remove n_i from L. Otherwise, just remove n_i from L.
Step 6. If $L \neq \phi$, go to **Step 5**. Otherwise, go to **Step 3**.

Fig. 6 A scheduling/FU binding algorithm.

Then from a start node to an end node, we pick up a node n_i whose $priority(n_i)$ is the maximum. After that, we bind it to the FU f_j so that estimated latency $l(n_i, f_j)$ is minimized. Estimated latency $l(n_i, f_j)$ is calculated as:

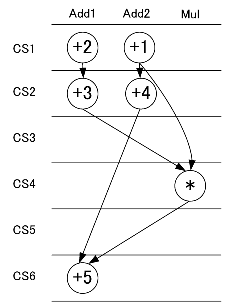
$$l(n_i, f_j) = c\_step(n_i, f_j) + cp(n_i, f_j)$$

where $c\_step(n_i, f_j)$ is the earliest Control Step where node n_i can be bound to the FU f_j. The estimated latency $l(n_i, f_j)$ means the maximum number of Control Steps to execute CDFG when a node n_i is bound to an FU f_j

Figure 6 shows our proposed scheduling/FU binding algorithm. We show our algorithm using the example as shown in Fig. 5 (a). First of all, the priority of each node is set to the underlined value in Table 1. Then, the nodes "+1" and "+2" are put into the ready list. We pick up the node "+2" because its priority is the highest. We calculate estimated latency of node "+2". $c\_step(+2, Add1)$ and $c\_step(+2, Add2)$ are 1 because "Add1" and "Add2" is unused in CS1. $l(+2, Add1)$ and $l(+2, Add2)$ can be calculated as $1 + 6 = 7$ and $1+7 = 8$, respectively. Then, we bind node "+2" to "Add1" because $l(+2, Add1)$ is lower than $l(+2, Add2)$. The node "+2" is removed from the ready list. Next, we pick up node "+1". $c\_step(+1, Add1)$ is 2 because "Add1" is already used in CS1. $l(+1, Add1)$ and $l(+1, Add2)$ can be calculated as $2 + 5 = 7$ and $1 + 6 = 7$, respectively. Since "Add1" is not able to be bound in CS1, we bind the node "+1" to "Add2". We finally have a scheduling/FU binding result as shown in **Fig. 7** (b) to repeat this procedure. Note that in CS3, we pick up node "$*$" from

Control Step	Ready list (priority)	Selected node	Estimated latency (Add1,Add2,Mul)	Binding
1	+1(5) , +2(6)	+2	7 , 8 , –	Add1
	+1(5)	+1	7 , 7 , –	Add2
2	+3(5) , +4(2)	+3	7 , 8 , –	Add1
	+4(2)	+4	5 , 4 , –	Add2
3	∗(3)	∗	– , – , 7	–
4	∗(3)	∗	– , – , 7	Mul
5	+5(1)	+5	7 , 7 , –	–
6	+5(1)	+5	7 , 7 , –	Add1

(a) Scheduling/FU binding procedure.



(b) Scheduling/FU binding result.

Fig. 7 An example of scheduling/FU binding result.

the ready list. However, $c\_step(*, Mul)$ becomes 4, i.e., we can use the "Mul" in CS4 and cannot use it in CS3. This is because data transfer from "Add1" (which is assigned to node "+3" in CS2) to "Mul" requires one clock cycle as in Fig. 5 (c) and them we cannot use "Mul" right now in CS3.

By introducing the data transfer table which deals with shared registers, we can successfully perform our extended scheduling/FU binding.

Note that, as in the original algorithm [13)], our extended algorithm can deal with control flow including conditional branches as well as data flow.

5. Register Allocation

It is necessary to decide for each FU whether to use a local register or a shared register in a distributed/shared-register architecture. In this section, we propose a register allocation algorithm.

A register allocation problem is, for given a scheduled CDFG, interconnection delay between modules, and the number of FUs, to determine whether to use shared registers or to add local registers for each FU under clock period constraint. The objective is to minimize the number of local registers.

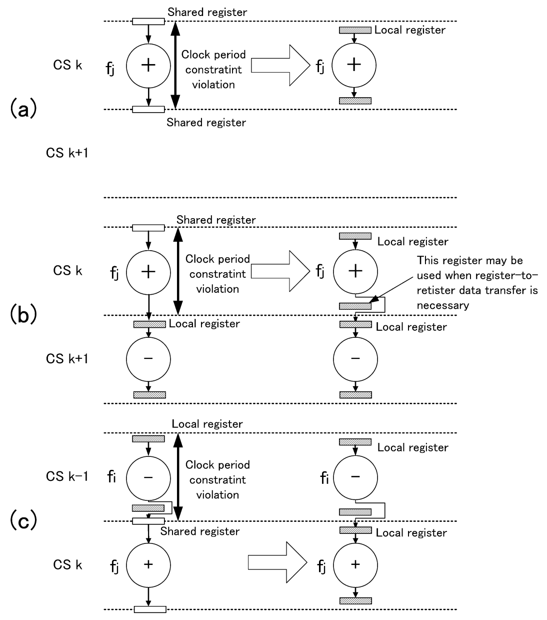
In the proposed method, the number of registers is minimized by using shared registers as much as possible. However, when the operation execution time (total delay time required from a source register to a destination register) violates clock period constraint, it is necessary to add local registers.

Let us pick up a functional unit f_j and assume that f_j is used in Control Step (CS) k. We add local registers in the following three cases:

Case (a): Assume that the source registers and the destination registers of f_j are shared registers as shown in **Fig. 8** (a). If the operation execution time of f_j exceeds clock period constraint in Control Step k, we add local registers for f_j for its input ports and output ports.

Case (b): Assume that the source registers of f_j are shared registers and the destination registers of f_j are local registers as shown in Fig. 8 (b). If the operation execution time of f_j exceeds the clock period constraint in Control Step k, we add local registers for f_j for its input ports and output ports.

Case (c): Assume that the source registers of f_i used in Control Step $k - 1$ are local registers and the destination registers of f_i are shared registers as



(a)
CS k
CS k+1
Shared register
Clock period constratint violation
Local register
(b)
This register may be used when register-to-retister data transfer is necessary
(c)
CS k−1
fi
fj

Fig. 8 Register allocation.

shown in Fig. 8 (c). If the operation execution time of f_i exceeds the clock period constraint in Control Step $k-1$ and the distance from FU f_i to shared registers is longer than the distance from FU f_i to FU f_j, we add local registers for f_j for its input ports and output ports.

In case (a) and (b), since the type of input registers of f_j is changed from "shared" to "local", the distance from the input registers to the FU f_j can be reduced. In case (c), since the type of input registers of f_j is changed from "shared" to "local", the distance from the FU f_i to the input registers of f_j can be reduced. We can expect that violation of clock period constraint can be relaxed by this register allocation strategy.

The algorithm is as follows: At first, we assume that every functional unit uses shared registers at the register allocation process in each iteration. (Step 1) For each FU f_l in a scheduled CDFG, if Case (a) holds true for f_l, we actually change the register type from shared registers to local registers for f_l. (Step 2) After that, for each FU f_l in a scheduled CDFG, if Case (b) and (c) holds true for f_l, we actually change the register type from shared registers to local registers for f_l. (Step 3) We repeat Step 2 until there are no FUs satisfying Case (b) and (c) or all the FUs have local registers.

6. Register Binding

Our architecture can use register-to-register data transfer. It is necessary to determine when and where data transfer should be done.

A register binding problem is, for given a scheduled CDFG and a register type of each FU, to assign each variable to a particular register in order to minimize the number of registers. In register binding, CVs are used for conditional branches.

In the proposed algorithm, variables are stored in its source registers[⋆1] as late as possible when the variables are bound to local registers. When collisions occur, we use shared registers temporarily. But we try to move variables from shared registers to local registers as soon as possible when the variables are bound to shared registers. We expect that, by increasing utilization of shared registers, the total number of required registers can be reduced. Note that, the variables which are allocated to shared registers are finally bound to actual shared registers by means of the left edge algorithm.

We show our algorithm using the example as shown in **Fig. 9**. Assume that module placement is given as in Fig. 9 (a) where the FU enclosed with the dotted line uses shared registers, and the other FUs use local registers.

Before starting register binding, we assume that each FU which uses local registers has one local register for each of its input ports and outputs ports.

In the first step in our algorithm, we allocate the "beginning" and "end" of the lifetime of all variables so that we can satisfy the scheduling result. Otherwise, if lifetime of a variable is less than 1 clock cycle, it is allocated to the register that their destination FU uses ($v4$ in Fig. 9 (c)). We allocate the "beginning" of the lifetime of all variables to the register which their source FU uses. The "end" of the lifetime of all variables is allocated to the register which their destination FU

⋆1 The source register of the variable v is the local register of the functional unit that generates the variable v.

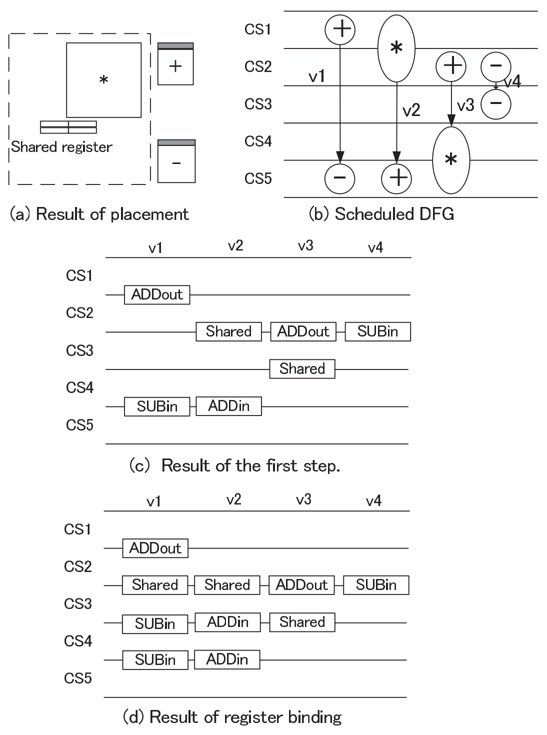


Fig. 9 Register binding example. XXin shows the input local registers and XXout shows the output local registers. Shared shows the shared registers.

uses. Figure 9 (c) shows the result of the first step. Here, `ADDin` and `ADDout` are the input local register and output local register of the adder "+", respectively, and `SUBin` is the input local register of the subtractor "−" ⋆1.

Second, we allocate each variable to registers in each control step from beginning to end of its lifetime. As for variable $v1$, it is first bound to the output local register `ADDout` of the adder "+" in CS1-2 (CS1-2 means the boundary between CS1 and CS2). Since the output local register of the adder is used in CS2-3 by variable $v3$, then we try to bind $v1$ to the input local register `SUBin` of the subtractor "−". However, since `SUBin` is also used in CS2-3 by $v4$, $v1$ is bound to a shared register in CS2-3. Since `SUBin` is unused in CS3-4, then we bind $v1$ to `SUBin` in CS3-4.

As for variable $v2$, it is first bound to a shared register in CS2-3. In CS3-4, since the input local register `ADDin` of the adder is unused, $v2$ is bound to it. In CS4-5, $v2$ is still bound to `ADDin`.

The same discussion can be applied to the variable $v3$ and $v4$ as in Fig. 9 (d).

If the register binding mentioned above fails, we increase the number of local registers by one for the FUs which causes the register collision and re-start the above register binding again.

7. Computational Complexity

Because our method repeats scheduling/FU binding, register allocation, register binding, controller synthesis and placement until a solution converges, the number of iterations changes depending on a target application. So we calculate the computational complexity of each of the proposed algorithms (scheduling/FU binding, register allocation and register binding) in our method.

First we calculate the computational complexity of the scheduling/FU binding algorithm. Let N_n be the number of nodes in a given CDFG and N_f be the number of functional units. In each control step, we require $O(N_f N_n + N_n \log N_n)$ time. The number of control steps is at most $O(N_n)$. Therefore, the computational complexity can be calculated as $O(N_f {N_n}^2 + {N_n}^2 \log N_n)$.

Next we calculate the computational complexity of the register allocation algorithm. In this algorithm, we require $O({N_n}^2)$ time to search clock period constraint violation for a CDFG. When this algorithm starts, we need this search first. Furthermore, we need this search every time we change the register type from "shared" to "local". Since the number of searches is at most N_f, the computational complexity of this algorithm can be calculated as $O(N_f {N_n}^2)$.

Finally we calculate the computational complexity of the register binding algorithm. Let N_v be the number of variables and N_s be the number of control steps. Since we require $O(N_s)$ time to assign each variable to registers, we require $O(N_s N_v)$ time for all variable assignments. When the result of the register

⋆1 For simplicity, the adder "+" and the subtractor "−" have one local register for its input port and one local register for its output port in the example of Fig. 9. Our FUs have two or more input/output ports in the experiments.

Table 2 Experimental results.

App.	FUs	Architecture	Area [μm^2]	Execution time [ns]	#Total Regs.	#Local Regs.	#Shared Regs.	#MUXs	CPU time[s]
DCT		Ours	22,950	32.4	18	9	9	60	307
	+2,*2	Distributed [13)]	23,655	32.4	23	23	-	57	100
		Shared	25,116	57.6	18	-	18	74	145
		Ours	29,336	25.2	21	15	6	72	356
	+3,*3	Distributed [13)]	36,750	25.2	26	26	-	67	100
		Shared	29,915	30.6	17	-	17	77	155
FIR		Ours	17,125	73.8	6	0	6	35	413
	+2,*2	Distributed [13)]	18,576	73.8	18	18	-	27	350
		Shared	17,125	73.8	6	-	6	35	180
		Ours	24,603	59.4	12	6	6	40	352
	+3,*3	Distributed [13)]	27,072	59.4	23	23	-	33	142
		Shared	24,705	64.8	7	-	7	52	436
EWF		Ours	12,110	50.4	10	0	10	32	97
	+1,*1	Distributed [13)]	13,244	50.4	13	13	-	31	78
		Shared	12,110	50.4	10	-	10	32	73
		Ours	11,625	37.8	6	0	6	31	67
	+2,*1	Distributed [13)]	13,493	37.8	13	13	-	29	77
		Shared	11,625	37.8	6	-	6	31	67
EWF3		Ours	15,982	106.2	14	9	5	48	204
	+2,*1	Distributed [13)]	17,136	106.2	15	15	-	45	327
		Shared	16,848	165.6	10	-	10	64	304
		Ours	24,235	93.6	17	12	5	66	724
	+3,*2	Distributed [13)]	26,640	95.4	22	22	-	67	310
		Shared	25,440	151.2	10	-	10	86	976
Parker		Ours	4,048	18.0	6	0	6	14	88
	+1,−1,<1	Distributed [13)]	5,499	18.0	11	11	-	14	69
		Shared	4,048	18.0	6	-	6	14	73
		Ours	5,320	12.6	7	0	7	18	57
	+2,−2,<1	Distributed [13)]	7,448	12.6	17	17	-	18	52
		Shared	5,320	12.6	7	-	7	18	56
COPY	+3,−1,<1,	Ours	112,797	257.4	119	47	72	354	1,008
	AND1,*5,	Distributed [13)]	127,260	266.4	145	145	-	381	1,798
	Shifter2	Shared	114,975	396.0	82	-	82	514	703

binding cannot satisfy the clock period constraint, this algorithm re-assigns variables to registers after increasing local registers. Since this re-assignment occurs at most N_v times, its computational complexity becomes $O(N_s {N_v}^2)$. After local register binding, the proposed algorithm executes the left edge algorithm, which requires $O({N_v}^2)$ time. Therefore, the computational complexity of the register binding algorithm becomes $O(N_s {N_v}^2)$.

Note that, our experimental results in Section 8 shows that the number of iterations is up to 25 times.

8. Experimental Results

We have implemented the proposed method in C++ on UNIX 3.0 GHz with 2 GB of memory. The method has been applied to an FIR filter (75 nodes),

Table 3 The area and the delay of functional units.

Functional unit	Area [μm^2]	Delay [ns]
Adder	287	1.36
Multiplier	4,507	2.93
Comparator	148	0.88
AND	68	0.03
Shifter	270	0.48

DCT (48 nodes), EWF (34 nodes), EWF3 (102 nodes), Parker [10)] (22 nodes, including conditional branches) and COPY (378 nodes, including conditional branches) where COPY is a real application. The proposed method targeting a distributed/shared-register architecture was compared with a distributed-register architecture ("Distributed" in **Table 2**) and a shared-register architecture ("Shared" in Table 2). Table 2 summarizes the experimental results. "Distributed" in Table 2 means the approach proposed in Ref. 13) which uses the synthesis flow shown in Fig. 3 (a). The number of Control Steps of "Distributed" is almost the same as "Ours". This is because our scheduling/FU binding algorithm is the extension of the one proposed in Ref. 13). "Shared" in Table 2 also uses the synthesis flow shown in Fig. 3 (a). Scheduling/FU binding of "Shared" uses CVLS proposed in Ref. 14). Register binding of "Shared" uses the left edge algorithm. In "Shared", after module placement, we decide required cycles to execute an operation for each FU based on the placement result.

All the functional units were assumed to have a bit width of 16, and the areas and the delays were obtained by synthesizing them beforehand based on the CMOS 90 nm technology. **Table 3** summarizes the area and the delay of functional units.

The area and the delay of one bit register were assumed to be 13 [μm^2] and 0.09 [ns], and those of 2-1 multiplexer were assumed to be 7 [μm^2] and 0.04 [ns]. The area of the controllers was synthesized by Synopsys Design Compiler in each iteration of Fig. 3. The wire delay was assumed to be a proportion to square of the wiring length, and set wire delay as 1 [ns] when wiring length is 250 [μm] [⋆1].

⋆1 Estimate of wire delay in high-level synthesis cannot be the same as that in physical design. However, even in physical design, especially in placement, wire delay is generally estimated by a proportion to the square of the distance between modules (for example, Ref. 8)). In our method, we use the same estimation for wire delay.

The I/O port of each module was assumed to be at the center of the module, and the clock period constraint was given to be 1.8 [ns].

The area of the experimental results is the minimum rectangle which includes the FUs, registers, MUXs, and the controller. The execution time shows the execution time of the application ("clock period" × "the number of Control Steps"). In our experiment, the convergence condition is met within 30 iterations for all given applications.

The experimental results show that the execution time of the proposed method is almost equal to that for distributed-register architectures and is reduced by a maximum of 44.7% and an average of 16.2% compared with shared-register architectures. The area of the proposed method is reduced by a maximum of 28.5% and an average of 13.2% compared with distributed-register architectures. These results demonstrate that the proposed method can reduce the area while maintaining the execution time equal to the distributed-register architectures.

9. Conclusions

In this paper, we proposed a new high-level synthesis method for distributed/shared-register architectures. The proposed method reduced by an average of 16.2% execution time compared with shared-register architectures and reduced by an average of 13.2% area compared with distributed-register architectures.

In the future, we will consider an architecture that has a distributed arrangement of shared registers and develop its dedicated high-level synthesis algorithms. Further, we will develop the method to reduce the number of MUXs for our high level synthesis system.

References

1) Cong, J., Fan, Y., Han, G., Yang, X. and Zhang, Z.: Architectural synthesis integrated with global placement for multi-cycle communication, *Proc. ICCAD'03*, pp.536–543 (2003).
2) Dougherty, W.E. and Thomas, D.E.: Unifying behavioral synthesis and physical design, *Proc. DAC'00*, pp.756–761 (2000).
3) Fang, Y.M. and Wong, D.F.: Simultaneous functional-unit binding and floorplanning, *Proc. ICCAD'94*, pp.317–321 (1994).

4) Gu, Z., Wang, J., Dick, R.P. and Zhou, H.: Unified incremental physical-level and high-level synthesis, *IEEE Trans. CAD*, Vol.26, No.9, pp.1576–1588 (2007).
5) Jeon, J., Kim, D., Shin, D. and Choi, K.: High-level synthesis under multi-cycle interconnect delay, *Proc. ASP-DAC'01*, pp.662–667 (2001).
6) Kim, D., Jung, J., Lee, S., Jeon, J. and Choi, K.: Behavior-to-placed RTL synthesis with performance-driven placement, *Proc. ICCAD'01*, pp.320–325 (2001).
7) Liu, Z., Bian, J., Zhou, Q. and Dai, H.: Interconnect delay and power optimization by module duplication for integration of high level synthesis and floorplan, *Proc. ISVLSI'07*, pp.279–284 (2007).
8) Ma, Y., Hong, X., Dong, S., Chen, S. and Cheng, C.K.: Perfomance constrainted floorplanning base on partial clustering, *Proc. ISCAS'05*, pp.1863–1866 (2005).
9) Murata, H., Fujiyoshi, K. and Nakatake, S.: Rectangle-packing-based module placement, *Proc. ICCAD'95*, pp.472–479 (1995).
10) NCSU CBL: www.cbl.ncsu.edu/benchmarks/.
11) Stammermann, A., Helms, D., Schulte, M., Schulz, A. and Nebel, W.: Binding, allocation and floorplanning in low power high-level synthesis, *Proc. ICCAD'03*, pp.544–550 (2003).
12) Sundaresan, V. and Vemuri, R.: A novel approach to performance-oriented datapath allocation and floorplanning, *Proc. ISVLSI'06*, pp.323–328 (2006).
13) Tanaka, A., Uchida, J., Miyaoka, Y., Togawa, N. and Ohtsuki, T.: High-level synthesis with floorplan and timing constraint for distributed-register architecture, *IPSJ Journal*, Vol.46, No.6, pp.1383–1394 (2005).
14) Wakabayashi, K. and Yoshimura, T.: A resource sharing and control synthesis method for conditional branches, *Proc. ICCAD'89*, pp.62–65 (1989).
15) Weng, J.P. and Parker, A.C.: 3D scheduling high-level synthesis with floorplanning, *Proc. DAC'91*, pp.668–673 (1991).
16) Zhong, L. and Jha, N.K.: Interconnect-aware high-level synthesis for low power, *Proc. ICCAD'02*, pp.110–117 (2002).

(Received December 25, 2007)
(Revised March 17, 2008)
(Accepted May 1, 2008)
(Released August 27, 2008)

(Recommended by Associate Editor: *Takashi Kambe*)

Akria Ohchi received the B. Eng. and M. Eng. degrees from Waseda University in 2005 and 2007 respectively, all in electronics, information and communication engineering. He is currently working towards the Dr. Eng. degree. His research interest is design and verification of VLSI, especially high-level synthesis. He is a student member of IEEE and IEICE.

Shunitsu Kohara received the B. Eng. and M. Eng. degrees from Waseda University in 2002 and 2004 respectively, all in electronics, information and communication engineering. He is currently working towards the Dr. Eng. degree. His research interest is design and verification of VLSI, especially hardware/software cosynthesis for system VLSIs. He is a student member of IEEE and IEICE.

Nozomu Togawa received the B. Eng., M. Eng., and Dr. Eng. degrees from Waseda University in 1992, 1994, and 1997, respectively, all in electrical engineering. He is presently an Associate Professor in the Department of Computer Science and Engineering, Waseda University. His research interests are VLSI design, graph theory, and computational geometry. He is a member of IEEE and IEICE.

Masao Yanagisawa received the B. Eng., M. Eng., and Dr. Eng. degrees from Waseda University in 1981, 1983, and 1986, respectively, all in electrical engineering. He was with University of California, Berkeley from 1986 through 1987. In 1987, he joined Takushoku University. In 1991, he left Takushoku University and joined Waseda University, where he is presently a Professor in the Department of Electronic and Photonic Systems. His research interests are combinatorics and graph theory, computational geometry, VLSI design and verification, and network analysis and design. He is a member of IEEE, ACM, and IEICE.

Tatsuo Ohtsuki received the B. Eng., M. Eng., and Dr. Eng. degrees from Waseda University in 1963, 1965, and 1970, respectively, all in electrical engineering. In 1965, he joined the NEC Corporation Ltd., Tokyo, Japan. From 1978 to 1980, he served as Research Manager, Application System Research Laboratory, at Central Research Laboratories. In 1980, he left NEC and joined Waseda University, where he is presently a Professor in the Department of Computer Science and Engineering. His research interests are algorithms and hardware engines for VLSI design and verification, computer algorithms for combinatorial problems, and network analysis/design. He is a Fellow of IEEE and IEICE.