IPSJ Transactions on System LSI Design Methodology Vol. 2 64–79 (Feb. 2009)

Regular Paper

A Behavioral Synthesis System for Asynchronous Circuits with Bundled-data Implementation

Naohiro Hamada,[†1] Yuki Shiga,[†1] Takao Konishi,[†1]
Hiroshi Saito,[†1] Tomohiro Yoneda,[†2] Chris Myers[†3]
and Takashi Nanya[†4]

This paper proposes a behavioral synthesis system for asynchronous circuits with bundled-data implementation. The proposed system is based on a behavioral synthesis method for synchronous circuits and extended on operation scheduling and control synthesis for bundled-data implementation. The proposed system synthesizes an RTL model and a simulation model from a behavioral description specified by a restricted C language, a resource library, and a set of design constraints. This paper shows the effectiveness of the proposed system in terms of area and latency through comparisons among bundled-data implementations synthesized by the proposed system, synchronous counterparts, and bundled-data implementations synthesized by using a behavioral synthesis method for synchronous circuits directly.

1. Introduction

Asynchronous circuits have several advantages such as average-case performance, low power consumption, and so on. However, the design of asynchronous circuits is difficult because designers must decide on a proper delay model, data encoding scheme, and control protocol according to a given application. Moreover, a hazard-free circuit must be realized because the propagation of a hazard results in circuit malfunction. Nevertheless, only a limited number of CAD tools are available.

Behavioral synthesis synthesizes an RTL model from a behavioral description specified by a programming language or its extension (e.g., C or SystemC [1]). A behavioral synthesis method generates an optimum RTL model through operation scheduling, resource allocation, and control synthesis while exploring the design space under design constraints. In the domain of synchronous circuits, many behavioral synthesis methods and their support tools have been developed because of the requirement for system-level design [2),3)].

This paper presents a behavioral synthesis system for asynchronous circuits with bundled-data implementation. The proposed system is based on a behavioral synthesis method for synchronous circuits. Operation scheduling and control synthesis are extended for bundled-data implementation because the execution of bundled-data implementation is different from synchronous circuits.

The proposed system synthesizes an RTL model from a behavioral description specified by a restricted C language, a resource library, and a set of design constraints. This paper presents the effectiveness of the proposed system in terms of area and latency through comparisons among bundled-data implementations synthesized by the proposed system, synchronous counterparts, and bundled-data implementations synthesized by using a behavioral synthesis method for synchronous circuits directly.

The rest of this paper is organized as follows. Section 2 gives related work. Section 3 gives background used in this paper. Sections 4 and 5 give the proposed system and its evaluation. Finally, Section 6 gives conclusions.

2. Related Work

There exist many behavioral synthesis methods for synchronous circuits [2),3)]. These methods schedule operations to a control step which implies a clock cycle. However, the direct application of these methods for asynchronous circuits may not synthesize optimum circuits. Let us explain the reason. Suppose we use a behavioral synthesis method for synchronous circuits to asynchronous circuits. In such a case, all operations are scheduled to a clock cycle. This ignores a characteristic of asynchronous circuits in which operations are executed immediately after the completion of previous operations. It may result in a performance loss or an extra use of resources in synthesized circuits. Even if we adjust operations so that each control step has an independent time interval using asynchronous control circuits, it does not change the execution order of operations. It means

†1 The University of Aizu, Japan
†2 National Institute of Informatics, Japan
†3 The University of Utah, USA
†4 The University of Tokyo, Japan

that a performance loss or an extra use of resources is not essentially solved just by changing the control scheme. This motivates us to develop a scheduling method dedicated for asynchronous circuits.

Several behavioral synthesis methods for asynchronous circuits have been also proposed. The method described in Ref. 4) is based on a heuristic list scheduling algorithm [2] which determines the start time of operations under resource constraints observing the availability of resources and the completion of direct predecessor operations. Compared to this method, the proposed system approximates a set of start time candidates for each operation and determine the start time of operations from the candidates considering design optimization. The method described in Ref. 5) explores resource sharing between operations by introducing additional dependence. However, it has the huge computational complexity (i.e., $O(3^{n(n-1)/2})$, where n is the number of operations).

On the other hand, the methods described in Refs. 6), 7) do not propose a new scheduling and/or allocation algorithm. Instead, these methods use templates for the control of registers derived from a synchronous or asynchronous behavioral synthesis method. Different from these methods, the proposed system extends operation scheduling and control synthesis.

The methods proposed by Venkataramani, et al. [8] and Bardsley and Edwards [9] generate asynchronous circuits from a high-level language such as C language or communicating sequential processes (CSP) [10]. Different from the proposed system where the design space is explored to generate an optimum circuit, these methods generate a circuit by a direct translation from a given specification without design space exploration.

We proposed a behavioral synthesis method for bundled-data implementation in our former work [11]. This paper extends our previous method so that control constructs such as branches and loops in a given description can be synthesized.

3. Background

3.1 Control Data Flow Graph

The Control Data Flow Graph (CDFG) is a directed graph which represents the data and control flow of an application. The CDFG is used as an intermediate representation in the proposed system. The CDFG G is defined as follows.

$$G = \langle N, BB, E \rangle$$

N, BB, E are sets of nodes, basic blocks, and direct edges, respectively.

The node set N ($N = \{n_i | i = 1, \ldots, \gamma\}$) consists of operation nodes, variable nodes, fork nodes, join nodes, the source node, and the sink node. γ represents the number of nodes. An operation node represents a data operation labeled by an operation type (e.g., addition), a variable node represents a variable (a primary input, a primary output, or the result of an operation), a fork node represents a branch, and a join node represents a merge of branched control flows, respectively. The source and sink nodes represent the start and end of the application, respectively.

A given behavioral description can be partitioned into a set of basic blocks BB ($BB = \{bb_k | k = 1, \ldots, \delta\}$). δ represents the number of basic blocks in a CDFG. A basic block bb_k is a sequence of consecutive statements in which control flow enters at the beginning and leaves at the end without halt or branch except at the end.

The edge set E consists of directed edges $e_{i,j}$ which represent dependencies between nodes n_i and n_j. If nodes n_i and n_j are an operation node and a variable node, the edge $e_{i,j}$ represents a data dependency. If either node n_i or n_j is a fork node, a join node, the source node, or the sink node, the edge $e_{i,j}$ represents a control dependency.

Figure 1 shows a CDFG. In Fig. 1, the rectangle nodes, circle nodes, triangle node, and inverted triangle node are operation nodes, variable nodes, a fork node, and a join node, respectively. For convenience, this paper denotes an operation node, a variable node, a set of operation nodes, and a set of variable nodes as o_i, v_i, O, and V ($(O \cup V) \subset N$).

3.2 Bundled-data Implementation

Bundled-data implementation is one of data encoding schemes for asynchronous circuits. In bundled-data implementation, N bit data transfer is represented by N + 2 signal wires. One bit data is represented by one signal wire. The two comes from the handshake signals, a request signal *req* and an acknowledge signal *ack*. Operations during a data transfer are initiated by a request signal *req* while the completion of operations is acknowledged by an acknowledge signal *ack*. To guarantee the completion of operations, a delay element is put on the *req* signal

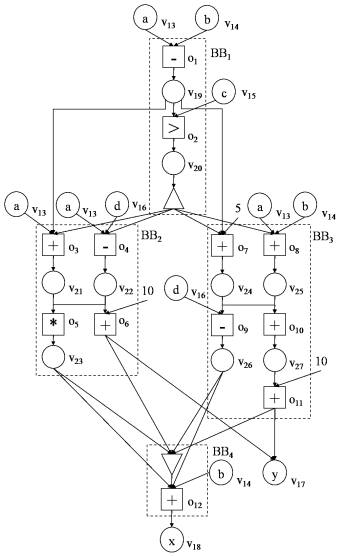


Fig. 1 Control data flow graph.

wire. The delay of a delay element is the sum of the maximum execution delays of resources used to execute operations.

Figure 2 shows a circuit model of bundled-data implementation used in this paper. The model consists of a data-path circuit and a control circuit. The data-path circuit consists of functional units which execute an operation, registers which store an input data or operation result, and multiplexers which select an appropriate input for a functional unit or register. The control circuit consists of Q-modules [12], glue logics, and delay elements which guarantee the control timing to write data into registers. In the proposed system, a Q-module is mapped to each state $s_h (h = 1, \ldots, \beta)$ which is determined by the operation scheduling result. β represents the number of states. The execution time of each state is equal to the delay of a delay element on the corresponding request signal.

The control of bundled-data implementation is explained as follows. Q-module q_h for state s_h is activated by a rising edge of input signal in_h which comes from the previous Q-module q_{h-1}. When an operation is executed at a shared

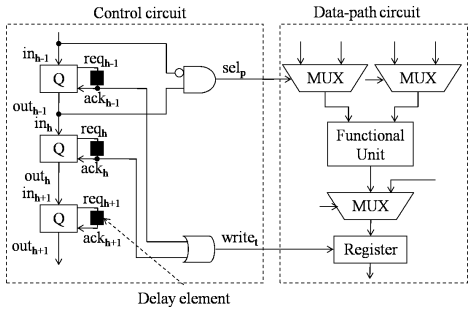


Fig. 2 A circuit model of bundled-data implementation.

functional unit or the result of an operation is written into a shared register in state s_h, multiplexers for such shared resources are controlled by in_h of Q-module q_h. As resources are shared at different states, the select signal sel_p for the p-th multiplexer is generated from a combination of several in_h signals. Then, Q-module q_h asserts req_h. req_h returns to Q-module q_h as ack_h through the corresponding delay element. A register write signal $write_t$ for the t-th register is generated from ack_h. When the register is shared one at different states, $write_t$ is generated from several ack_h via an OR gate. As states are sequentially ordered, no Q-modules control a shared register at the same time. It implies that an OR gate is enough to control a shared register. A data is written into a register by a falling edge of ack_h. After ack_h is deasserted, Q-module q_h passes the control to the next Q-module q_{h+1} with a rising edge of output signal out_h.

4. The Proposed Behavioral Synthesis System

4.1 Synthesis Flow

The synthesis flow of the proposed system is shown in **Fig. 3**. The inputs of the proposed system are a behavioral description of an application, a resource library, and a set of design constraints as inputs.

The front-end analyzes a behavioral description of an application and generates the CDFG of the application. After the CDFG is generated, the bit-width of operations and variables is analyzed. After bit-width analysis, the initial allocation is carried out. In the initial allocation, a functional unit is allocated for each

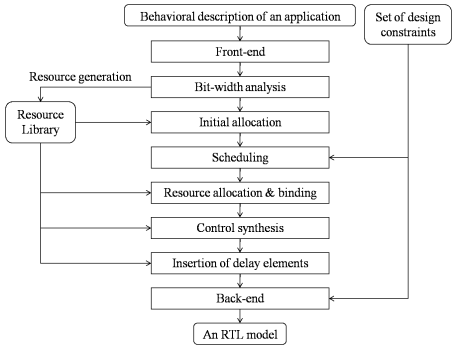


Fig. 3 The flow of the proposed system.

operation to determine the execution time used in operation scheduling. Next, operation scheduling is applied to determine the start time of operations. After operation scheduling, a functional unit or register is allocated for each operation and variable. If shared resources exist, multiplexers are allocated. A data-path circuit is synthesized after resource allocation and binding. Before control synthesis, the state space is decided from the operation scheduling result. In control synthesis, mapping of Q-modules and generation of delay elements and glue logics are carried out to synthesize a control circuit. Finally, the proposed system generates a synthesizable RTL model and a simulation model in Verilog HDL.

The behavioral synthesis method in the proposed system is based on a behavioral synthesis method in synchronous circuits as shown in Ref. 2). For bundled-data implementation, we extend operation scheduling and control synthesis. In the following sub-sections, this paper describes the detail of the behavioral synthesis method in the proposed system. It is based on our former work [11]. This paper describes how behavioral descriptions not only data operations but also control constructs such as branches and loops are synthesized.

4.2 Inputs and Output of the Proposed System

The inputs and output of the proposed system are listed below.

- Inputs
  - A behavioral description of an application
  - A resource library
  - A set of design constraints
- Output
  - An RTL model and a simulation model of bundled-data implementation

Table 1 The C language syntax supported in the proposed system.

Integer type constants and variables
Assignments
if
switch
for
while
do-while

Table 2 Parameters in a resource library.

Parameters in a resource library
Area
The maximum execution delay
Executable operations
The bit-width of inputs and output

The behavioral description of an application must be written by the C language syntax shown in **Table 1**. Otherwise, the proposed system terminates the synthesis process with an error. Input signals and output signals can be explicitly specified in the behavioral description using "pragma". Each resource in a resource library is parameterized with parameters shown in **Table 2**. Resource parameters must be specified in an XML format and resources must be prepared as synthesizable RTL models in Verilog HDL. A set of design constraints may have a time constraint or a set of resource constraints used for operation scheduling, a delay margin to generate delay elements. A set of design constraints must be specified in an XML format.

Note that the proposed system is not restricted on to the C language. If we can provide a proper front-end, the proposed system can synthesize bundled-data implementation from other languages as well.

The proposed system generates two circuit models. One is a synthesizable RTL model for implementation and the other is a simulation model for functional verification. In the simulation model, arbitrary short delays are inserted for all feedback loops. This is because logic simulators cannot generate correct values if

there is no time difference between input signals and feedback loops. In addition, the delays of delay elements are explicitly represented by exact times even though delay elements in the synthesizable RTL model are represented by logic gates. Designers can synthesize and simulate these models using a conventional logic synthesis tool or an HDL simulator.

4.3 Front-end

The front-end analyzes and optimizes a given behavioral description using COmpiler INfra Structure (COINS) [13]. COINS supports optimization in compliers such as common sub-expression elimination and generates an intermediate format called High-level Intermediate Representation (HIR) which looks like a syntax tree. The front-end generates a CDFG from a generated HIR.

4.4 Bit-width Analysis

The bit-width of operations and variables is analyzed using the method in Ref. 11). In bundled-data implementation, bit-width analysis is one of the important processes for optimization. Delay elements in bundled-data implementation are generated so that the delays of delay elements are larger than the maximum delays of used resources. Therefore, if we can use resources with short delays, the performance of bundled-data implementation can be improved. In general, the bit-width of a resource is shorter, the delay of the resource is shorter.

4.5 Initial Allocation

A functional unit is allocated for each operation node o_i to decide the maximum execution delay used in operation scheduling. The maximum execution delay of operation o_i is represented by $d(o_i)$.

In time constraint scheduling, the main objective is to minimize the number of resources used in a data-path circuit. To maximize resource sharing, for the same type of operations, the proposed system allocates the resource which can execute the operation with the maximum bit-width. On the other hand, in resource constraint scheduling, the main objective is to minimize the latency of a data-path circuit. Therefore, for each operation, the resource whose bit-width matches to the operation is allocated. If there is no suitable resource in a given resource library, the proposed system asks designers to prepare such a resource before synthesis.

As the use of only functional unit delays may violate a given time constraint after synthesis due to the allocation of registers and multiplexers, the proposed system adds one register delay and two multiplexer delays to each $d(o_i)$. Two multiplexer delays correspond to a multiplexer used for a functional unit and a multiplexer used for a register. A multiplexer delay is estimated from the average number of functional unit sharing or register sharing in the As Late As Possible (ALAP) schedule [2] where operations o_i are scheduled to the latest start time $alaps_i$ under a given time constraint.

4.6 Operation Scheduling

Operation scheduling determines the start time of each operation. The proposed system supports time constraint scheduling and resource constraint scheduling. The objective of time constraint scheduling is to decide start times minimizing the number of resources while the objective of resource constraint scheduling is to decide start times minimizing the latency. The proposed system uses the Asynchronous Force-directed Scheduling (AFDS) algorithm [15] as a time constraint scheduling algorithm while the Asynchronous Force-directed List Scheduling (AFDLS) algorithm as a resource constraint scheduling algorithm. Both algorithms are based on the FDS and FDLS algorithm [14] developed for synchronous circuits.

In the FDS and FDLS algorithm, operations are assigned to one of control steps which have a uniform time interval. These control steps represent clock cycles. On the other hand, in the AFDS and AFDLS algorithms, control steps are determined from sets of approximated start time candidates of operations. In such a case, the time intervals among control steps are not uniform. The reason why control steps are decided so is that operations in bundled-data implementation are executed immediately after the completion of a previous operation. Note except the decision of control steps there is no big difference between the FDS (FDLS) and the AFDS (AFDLS) algorithms. Therefore, other scheduling algorithms which use control steps can also be extended in the similar way.

This sub-section describes the overview to determine control steps from sets of approximated start time candidates of operations, the AFDS algorithm, and the AFDLS algorithm.

4.6.1 Approximation of Start Times

Figure 4 shows the function ApproximateStep which is used in the proposed

```
ApproximateStep(CDFG G, TimeConstraint tc)
1   Asap = ComputeASAP(G);
2   Alap = ComputeALAP(G, tc);
3   Frame = ComputeTimeFrame(G, Asap, Alap);
4   for ∀o_i ∈ O do
5     Direct_i = GetPredecessorOp(o_i, G);
6     Share_i = GetShareOp(o_i, G);
7     Mutex_i = GetMutexOp(o_i, G);
8     Cand_i = ComputeStartTimeCandidate(o_i, Direct_i, Share_i, Mutex_i, Asap);
9     Cand = Cand ∪ Cand_i
10  end for
11  Step = ComputeStep(G, Cand);
12  return Step;
```

Fig. 4 The function ApproximateStep.

system to determine control steps from sets of start time candidates. The inputs of the function ApproximateStep are a CDFG G and a time constraint tc and the output of the function is a set of control steps denoted as $Step$ ($Step = \{step_w | w = 1, \ldots, \lambda\}$). $step_w$ is a positive integer and λ represents the number of control steps.

Before the approximation of start time candidates, the As Soon As Possible (ASAP) and ALAP schedules are calculated. The ASAP schedule determines the earliest start time $asaps_i$ for each operation o_i [2]. The completion times in the ASAP and ALAP schedules denoted as $asapc_i$ and $alapc_i$ are the sum of $asaps_i$ or $alaps_i$ and $d(o_i)$.

From $asaps_i$ and $alaps_i$ for each operation o_i, the time frame $Frame_i$ where operation o_i can be scheduled without violating a given time constraint is calculated. The time frame $Frame_i$ is defined as follows.

$$Frame_i = alaps_i - asaps_i$$

After time frames are calculated, a set of start time candidates for each operation o_i is approximated from the completion times of direct predecessor operations, concurrent operations, and mutually exclusive operations. This paper denotes a set of start time candidates, a set of direct predecessor operations, a set of concurrent operations, and a set of mutually exclusive operations for operation o_i as $Cand_i$, $Direct_i$, $Conc_i$, and $Mutex_i$, respectively. $Direct_i$ is defined as follows.

$$Direct_i = \{o_j \in O | e_{j,i} \in E\}$$

A direct predecessor operation o_j for operation o_i is an operation that has a directed edge $e_{j,i}$.

For $Mutex_i$ and $Conc_i$, a set of transitive predecessors and successors for operation o_i is calculated. We represent a transitive relation $i \rightarrow j$ if there is a path of edges $(e_{i,x}, \ldots, e_{y,j})$ from node n_i to node n_j. A set of transitive predecessors and successors for operation o_i is denoted as T_i and calculated as follows.

$$T_i = \{o_j \in O | i \rightarrow j \vee j \rightarrow i\}$$

$Mutex_i$ and $Conc_i$ are defined as follows.

$$Mutex_i = \{o_j \in O | o_j \notin T_i, o_i \in bb_k, o_j \in bb_l, bb_k, bb_l \in BB, k \neq l\}$$

An operation o_j in $Mutex_i$ is neither a transitive successor nor a transitive predecessor for operation o_i and belonging to a different basic block from the basic block of operation o_i.

$$Conc_i = \{o_j \in O | o_j \notin (T_i \cup Mutex_i), asaps_i \leq asapc_j \leq alaps_i\}$$

An operation in $Conc_i$ is neither a transitive successor, a transitive predecessor, nor a mutually exclusive operation for operation o_i and $asapc_j$ is a value between $asaps_i$ and $alaps_i$.

Finally, $Cand_i$ for operation o_i is calculated from $Direct_i$ and $Conc_i$ as follows.

$$Cand_i = \{st | \forall o_j \in (Direct_i \cup Conc_i), st = asapc_j\}$$

A start time candidate st for operation o_i is $asapc_j$ of $o_j \in Direct_i \cup Conc_i$ that satisfies $asaps_i \leq st \leq alaps_i$. Note we assume that st is a positive real number. This paper calls o_j as a previous operation. To calculate start time candidates more, $Cand_i$ for operation o_i is calculated recursively from the execution sequences of previous operations.

After the approximation of start time candidates, the union set $Cand$ ($Cand = st | st \in \bigcup Cand_i$) of all $Cand_i$ is calculated. Sorting start time candidates in $Cand$ in the ascending order, the proposed system decides $Step$ by translating each start time candidate in $Cand$ to a positive integer which corresponds to $step_w$. Similarly, $Cand_i$ for each operation o_i is translated into a set of schedulable steps $Step_i$ ($Step_i \subseteq Step$).

```
AFDS(CDFG G, TimeConstraint tc)
1   repeat until all operations are scheduled;
2     Step = ApproximateStep(G, tc);
3     Prob = ComputeProbability(G, Step);
4     DG = ComputeDistributionGraph(G,
        Step, Prob);
5     Sf = ComputeForce(G, Step, Prob, DG);
6     start_i = GetMinForceOpAndStep(Sf, Start);
7   end repeat
8   return Start
```

Fig. 5 The AFDS algorithm.

4.6.2 The AFDS algorithm

Figure 5 shows the AFDS algorithm. The inputs of the AFDS algorithm are a CDFG G and a time constraint tc. The output is a set of start times for operations denoted as $Start$.

The following processes are repeatedly executed until all operations are scheduled. At first, the AFDS algorithm calls the function ApproximateStep to decide control steps. Next, for each $step_w \in Step_i$, the probability $Prob(i, w)$ that an operation o_i is scheduled to $step_w$ is calculated. Then, for each operation type (e.g., addition), Distribution Graphs (DGs) which represent the resource utilization of each resource are calculated from $Prob(i, w)$. DGs are calculated for each basic block independently and then merged to generate the entire DGs for the CDFG. After the calculation of DGs, the cost function called self force $Sf(i, w)$ is calculated for each $step_w$. The self force $Sf(i, w)$ represents how resource utilization is balanced through control steps when an operation o_i is scheduled to a control step $step_w \in Step_i$. Finally, the operation o_i which has the minimum $Sf(i, w)$ is scheduled to the control step $step_w$. This paper represents the start time of operation o_i as $start_i (start_i = step_w, \exists step_w \in Step_i)$.

4.6.3 The AFDLS algorithm

Figure 6 shows the AFDLS algorithm. The inputs of the AFDLS algorithm are a CDFG G, a resource library R, and a set of resource constraints RC.

Initially, *currentstep* which represents the current referenced control step is set to 0. The following processes are repeatedly carried out until all operations are scheduled. For each resource $r \in R$ whose resource constraint rc_r is more than 0, a set of operations that can be scheduled to *currentstep* using resource r and a set of executed operations that are already scheduled to a previous step using resource r but not finished at *currentstep* are calculated. These sets are denoted as Op and $ExecutedOp$. If the number of Op plus the number of $ExecutedOp$ is less than resource constraint rc_r, all operations in Op are scheduled to *currentstep*. Otherwise, operations less than $rc_r - |ExecutedOp|$ are scheduled based on self forces. For the calculation of self forces, the same processes as the AFDS algorithm are carried out. Only difference is that although self forces in the AFDS algorithm are calculated for each schedulable step of all operations, self forces in the AFDLS algorithm are calculated only for *currentstep* of operations in Op.

```
AFDLS(CDFG G,
  ResourceLibrary R, ResourceConstraint RC)
1   currentstep = 0;
2   repeat until all operations are scheduled;
3     for ∀r ∈ R do
4       if (rc_r > 0)
5         Op = GetSchedulableOp(G, r, currentstep);
6         ExecutedOp = GetExecutedOps(G,
            Start, r, currentstep);
7         if(|Op| + |ExecutedOp| ≤ rc_r)
8           for ∀o_i ∈ Op
9             start_i = currentstep;
10        else
11          for i = 1 to rc_r − |ExecutedOp| do
12            Step = ApproximateStep(G, asaps_sink);
13            Prob = ComputeProbability(G, Step, currentstep);
14            DG = ComputeDistributionGraph(G, Step, Prob, currentstep);
15            Sf = ComputeForce(G, Step, Prob, DG, currentstep);
16            start_i = GetMinForceOp(Sf, currentstep);
17          end for
18        end if
19      end if
20    end for
21    currentstep = GetNextStep(G, Start);
22  end repeat
23  return Start
```

Fig. 6 The AFDLS algorithm.

After the scheduling at *currentstep*, the next control step is decided from the earliest completion time of scheduled operations.

4.7 Resource Allocation and Binding

The proposed system allocates a functional unit for each operation and then a register for each variable. For shared functional units and registers, the proposed system allocates multiplexers.

The proposed system uses an extension of the Left-Edge (LE) algorithm [2] called the Extended Left-Edge (ELE) algorithm [11]. The difference between the LE algorithm and the ELE algorithm is that the ELE algorithm uses a priority for resource allocation calculated from the bit-width, inputs, and output of operations. The objective of the ELE algorithm is to minimize the bit-width of allocated resources and the number of allocated multiplexers.

Figure 7 shows the ELE algorithm for functional unit allocation and **Fig. 8** shows the function AllocateResource called in the ELE algorithm. A set of operations that can be bounded by a functional unit $fu_c(c = 1, \ldots, \theta)$ is denoted as $Op \subseteq O$ and a set of functional units that can execute operations in Op is denoted as $Fu(Fu \subseteq R)$. θ represents the number of functional units which is determined by the scheduling result or a set of resource constraints. In the proposed system, resource allocation is carried out without increasing the number of resources. Another note is that the ELE algorithm is called for each resource type in functional unit allocation. A set LT represents the life time of operations. The life time $life_i$ of an operation o_i is defined from the start time to the completion time of o_i.

The ELE algorithm initially sorts operations in Op by the ascending order of start times. Op is set to a set $Unalloc$ which represents unallocated operations and 0 is set to $Alloc$. $Alloc_i$ represents the index of the allocated functional unit for operation $o_i \in Op$.

Then, the following processes are carried out until $Unalloc$ becomes the empty set. For each control step $step_w$, a subset of operations whose lifetime intersects to $step_w$ is calculated. It means that operations are executed at $step_w$. This subset is represented as $SubO \subseteq Op$. In addition, a set of available functional units at $step_w$ is calculated. This set is represented as $Avail \subseteq Fu$. Functional units $fu_c \in Avail$ are allocated to operations $o_i \in SubO$ in the function AllocateResource.

```
ELE(OperationSet Op, FunctionalUnitSet Fu,
    LifeTimeSet LT)
1   Sort(Op, LT);
2   Unalloc = {Op}; Alloc = 0;
3   while Unalloc ≠ ∅ do
4     ∀step_w ∈ Step do
5       SubO = GetSubSet(Op, LT, step_w);
6       Avail = GetAvail(Fu, LT step_w);
7       AllocateResource(Op, SubO, Avail, Unalloc, Alloc);
8     end for
9   end while
10  return Alloc;
```

Fig. 7 The ELE algorithm for functional unit allocation.

```
AllocateResource(OperationSet Op,
  OperationSet SubO, ResourceSet Avail
  UnallocatedSet Unalloc, AllocationIndex Alloc)
1   Io = ComptueIONum(Op, SubO, Avail);
2   Bit = ComptueBitDiff(Op, SubO, Avail);
3   while(SubO ≠ ∅ ∨ Avail ≠ ∅) do
4     fu_c = SelectResource(Io, Bit);
5     o_i = SelectOperation(Io, Bit);
6     Alloc_i = index(fu_c);
7     Unalloc = Unalloc \ {o_i};
8   end while
```

Fig. 8 The function AllocateResource for functional unit allocation.

In the function AllocateResource, Io and Bit are calculated for each pair of $o_i \in subO$ and $fu_c \in Avail$. $Io(i, c)$ represents the number of the same inputs and output among operation o_i and operations $o_j \in Op$ when the same functional unit fu_c as o_j is allocated to o_i.

$$Io(i, c) = \sum_{Alloc_j == index(fu_c)} IONum(o_i, o_j)$$

$IONum(o_i, o_j)$ represents the number of the same inputs and output between operations o_i and o_j. The same inputs mean that the sources of operations

are the same while the same output means that the destination of operations is the same. The outputs of operations become the same when mutually exclusive assignments for the same variable exist in branches (e.g., o_6 and o_{11} in Fig. 1). A higher value in $Io(i, c)$ implies that the number of inputs for the multiplexers used for the functional unit fu_c and a register to store the operation result is reduced more.

$Bit(i, c)$ represents the difference of the bit-width among operation o_i and operations $o_j \in Op$ when the same functional unit fu_c as o_j is allocated for o_i. Here, b_i and b_j represent the bit-width of operations o_i and o_j, respectively.

$$Bit(i, c) = \sum_{Alloc_j == index(fu_c), b_j - b_i \geq 0} (b_i - b_j)$$

If b_j minus b_i is more or equal to 0, the difference between b_j and b_i is accumulated to $Bit(i, c)$. A smaller value in $Bit(i, c)$ means that the difference of the bit-width among operation o_i and operations o_j is large. It implies that many bits in the functional unit fu_c are not utilized by operation o_i. Such an allocation is not suitable in the view of resource sharing.

After the calculation of Io and Bit, a functional unit is allocated for each operation in $SubO$ until $SubO$ or $Avail$ becomes the empty set. Resource allocation is carried out from the combination of operation $o_i \in SubO$ and functional unit $r_c \in Avail$ where Io has the maximum value. If there are more than two combinations, the combination that the value of Bit is 0 or the closest to 0 is selected.

Figures 9 and **10** show the ELE algorithm and the function AllocateResource for register allocation. Instead of Op and Fu in the ELE algorithm for functional unit allocation, the variable set V and the register set Reg are given as arguments. The procedure is mostly the same as functional unit allocation. For each control step $step_w$, a subset of variables ($SubV$) whose lifetime intersects to $step_w$ and a set of available registers ($Avail$) at $step_w$ are calculated. Then, registers reg_c are allocated based on the priority calculated from the number of the same input/output and the difference of the bit-width among variables.

After functional unit and register allocation, multiplexers are allocated for shared functional units and registers.

```
ELE(VariableSet V, RegisterSet Reg,
    LifeTimeSet LT)
1   Sort(V, LT);
2   Unalloc = {V}; Alloc = 0;
3   while Unalloc ≠ ∅ do
4     ∀step_w ∈ Step do
5       SubV = GetSubSet(V, LT, step_w);
6       Avail = GetAvail(Reg, LT step_w);
7       AllocateResource(V, SubV, Avail, Unalloc, Alloc);
8     end for
9   end while
10  return Alloc;
```

Fig. 9 The ELE algorithm for register allocation.

```
AllocateResource(VariableSet V,
  VariableSet SubV, ResourceSet Avail
  UnallocatedSet Unalloc, AllocationIndex Alloc)
1   Io = ComptueIONum(V, SubV, Avail);
2   Bit = ComptueBitDiff(V, SubV, Avail);
3   while(SubV ≠ ∅ ∨ Avail ≠ ∅) do
4     reg_c = SelectResource(Io, Bit);
5     v_i = SelectVariable(Io, Bit);
6     Alloc_i = index(reg_c);
7     Unalloc = Unalloc \ {v_i};
8   end while
```

Fig. 10 The function AllocateResource for reigster allocation.

Finally, resources in a given resource library are bound for allocated functional units, registers, and multiplexers. During resource binding, resources which have the enough bit-width are bound.

4.8 Control Synthesis

Before control synthesis, the proposed system calculates the state space of a synthesized circuit from the scheduling result. Then, a control circuit is synthesized through mapping of Q-modules and the generation of glue logics and delay elements.

4.8.1 State Allocation

The proposed system extends the state allocation method proposed by Tseng,

et al.[16] so that states are determined by the start times of operations. The proposed system supports the following slicing methods.

- The local slicing
- The global slicing simple
- The global slicing complex

In the local slicing, states are determined from the set *Start*. An interval between start times becomes a state s_h. Different states are allocated for mutually exclusive basic blocks. In the global slicing simple, several states in different basic blocks are merged if the interval for the states is equivalent. The global slicing complex is an extension of the global slicing simple in that not only start times but also completion times are used for state allocation.

4.8.2 Mapping of Q-modules and generation of glue logics.

The proposed system maps a Q-module q_h to each state s_h. Then, glue logics are generated. A multiplexer select signal sel_p for the p-th multiplexer is generated from a glue logic which comes from input signals in_h of Q-modules q_h. A register write signal $write_t$ for the t-th register is generated from a glue logic which comes from acknowledge signals ack_h of Q-modules q_h.

4.8.3 Insertion of Delay Elements

The delay sd_h for state s_h is the maximum path delay which is calculated from the sum of the delays of used resources in the state. The proposed system generates delay elements with buffers. As data are written into registers by a falling edge of ack_h, every delay element is passed to twice. The first is from a rising edge of req_h to a rising edge of ack_h and the second is from a falling edge of req_h to a falling edge of ack_h. It implies that the required delay of a delay element is a larger value than $sd_h/2$.

Usually, the delay in state s_h becomes long after the physical design due to wire delays. Moreover, the delay may be changed by technological or environmental variations. Therefore, the proposed system generates delay elements with a margin *margin* specified in a given constraint file. The number of buffers in a delay element is decided so that the delay of the delay element is larger than $margin * sd_h/2$.

5. Experimental Results

This section shows the effectiveness of the proposed system comparing the synthesized RTL models of bundled-data implementations using the proposed system with the synchronous counterparts and the bundled-data implementations using a behavioral synthesis method for synchronous circuits. This paper calls latter bundled-data implementations as direct implementations. For the experiments, the proposed system is implemented in Java. The FDS and FDLS algorithms and a finite state machine (FSM) generator are also implemented for the synthesis of synchronous circuits. Note that in the experiments optimization techniques such as pipelining and chaining are not concerned. They will be considered in our future work. The experiments are carried out on a Windows machine which has a dual-core processor (2.66 GHz) and a 2G memory.

Table 3 shows the statistics of benchmarks used in the experiments. These benchmarks are downloaded from Refs. 17), 18) and modified to satisfy supported syntaxes shown in Table 1. The columns in Table 3 represent the name, the number of operations, the number of basic blocks, the number of branches, and the number of loops in benchmarks, respectively.

Table 4 shows a part of a used resource library. Each resource is modeled by Verilog HDL and synthesized by using Xilinx ISE WebPACK 9.2i[19] targeting Virtex4 (xc4vlx15-12sfs623) FPGA. The columns of Table 4 represent the name, bit-width, area, and delay of each resource. The unit of area is slice. A slice consists of two flip-flops and two 4-to-1 look-up tables (LUTs). As multipliers can be implemented on not slices but embedded multipliers in Vertex4, the number

Table 3 Benchmarks.

name	# of ops	# of bbs	branch	loop
usqrt	16	6	1	1
fdct	35	19	0	6
SNR	93	35	7	6
decoder	36	27	10	1
fht	172	13	0	4
pred1	158	43	19	2
mdct	173	4	0	1
quant	45	23	8	1

Table 4 A part of resource library.

name	bit-width	area [slice]	delay [ns]
mux2	32	9	0.2
mux4	32	30	0.4
register	32	32	0.5
adder	16	8	0.8
adder	32	16	1.4
multiplier	16	0	3.5
multiplier	32	0	7.3

of slices for multipliers is set to 0.

In the proposed system, the AFDS or AFDLS algorithm, the ELE algorithm, and the global slicing simple are used for operation scheduling, resource allocation, and state allocation. No time margin is assigned to generate delay elements. For the synchronous counterparts, the FDS or FDLS algorithm, the ELE algorithm, and the global slicing simple are used. The control circuits in the synchronous counterparts are generated by using the FSM generator. The direct implementations are synthesized by using the FDS or FDLS algorithm, the ELE algorithm, and the global slicing simple for data-path circuits and Q-modules for control circuits.

The time interval of control steps when the FDS or FDLS algorithm is used is decided as follows. First, we find two operations from the initial allocation result. One has the minimum operation delay and the other has the maximum operation delay. We synthesize benchmarks by changing the time interval 0.1 by 0.1 from the minimum operation delay to the maximum operation delay. The time interval which synthesizes an optimum RTL model of the synchronous counterpart is selected.

The first comparison shows the number of resources and the number of slices for the synthesized RTL models under a time constraint. **Table 5** shows the experimental results. The rows in "async" represent the results in the proposed system, the rows in "sync" represent the synchronous counterparts, and the rows in "direct" represent the direct implementations. For each benchmark, behavioral synthesis is carried out for three time constraints. The first constraint corresponds to the critical path delay of each benchmark derived by the ASAP algorithm. The second and third constraints correspond to the critical path delay * 1.5 and the critical path delay * 2.0, respectively. The column "t_{step}" represents the time interval of control steps when the FDS algorithm is used. The columns "FUs", "Regs", and "Muxs" in "resource usage" represent the numbers of functional units, registers, and multiplexers in the synthesized RTL models, respectively. The column "states" represents the number of states in the synthesized RTL models. The column "area" represents the number of slices when logic synthesis is carried out for the synthesized RTL models using ISE. The columns "S", "RA", "CS", and "others" in "run-time" represent the times for scheduling, resource allocation, control synthesis, and other processing. The column "total" represents the total behavioral synthesis time.

Note the symbol "-" in area means that ISE cannot synthesize logic circuits because of their substantial large state space. Logic synthesis by ISE is frozen after the whole memory space on our environment is utilized. Another note, for benchmarks usqrt and fdct, we verify the functional correctness using the generated simulation models and an HDL simulator ModelSim [20].

The second comparison in **Table 6** shows the latency of the synthesized RTL models under a set of resource constraints. For each benchmark, we synthesize RTL models two times by changing the number of functional units arbitrary. The number of functional units is shown in the column "rc". Similar to the first comparison, the rows in "async" represent the results in the proposed system, the rows in "sync" represent the synchronous counterparts, and the rows in "direct" represent the direct implementations. The column "latency" represents the latency of the synthesized RTL models. The values in "async" and "direct" are the sum of the state delays while the values in the synchronous counterparts are the product of t_{step} by the number of states.

5.1 Discussion

Area. As the main objective of time constraint scheduling is to minimize area, we discuss the impact of area in the proposed system referring to Table 5.

Compared to the synchronous counterparts, the area of synthesized circuits using the proposed system is slightly large. As a buffer of delay elements is implemented by one slice in FPGAs, the large portion of the area overhead is occupied by delay elements. To reduce the area overhead, one may consider the optimization of delay elements. It can be realized by utilizing the delays

Table 5 Synthesis results under a time constraint.

name	circuit	tc	t_{step}	resource usage			states	area	run-time [s]				
		[ns]	[ns]	FUs	Regs	Muxs		[slice]	S	RA	CS	others	total
usqrt	async	43.0		8	10	17	12	881	0.16	0.05	0.03	0.14	0.38
		64.5		6	10	13	15	782	0.33	0.05	0.02	0.19	0.59
		86.0		6	10	14	14	788	0.31	0.05	0.03	0.14	0.53
	sync	43.0	1.0	8	10	17	43	765	0.08	0.27	0.03	0.19	0.57
		64.5	1.0	7	9	14	57	732	0.13	0.05	0.02	0.17	0.37
		86.0	1.0	7	10	15	49	778	0.13	0.05	0.03	0.19	0.40
	direct	43.0	1.0	8	10	17	13	843	0.06	0.06	0.03	0.94	1.09
		64.5	1.0	7	9	14	14	801	0.09	0.05	0.03	0.38	0.55
		86.0	1.0	7	10	15	13	850	0.11	0.05	0.03	0.39	0.58
fdct	async	76.5		5	16	17	20	888	0.36	0.08	0.03	0.16	0.63
		104.5		5	16	17	22	855	0.33	0.06	0.05	0.17	0.61
		140.4		4	17	14	29	875	0.53	0.06	0.05	0.17	0.81
	sync	76.5	4.5	5	16	18	32	875	0.20	0.08	0.03	0.22	0.53
		104.5	1.7	5	16	19	104	926	0.41	0.08	0.08	0.25	0.82
		140.4	2.7	5	17	19	76	873	0.38	0.06	0.05	0.17	0.66
	direct	76.5	4.5	5	16	18	19	960	0.19	0.06	0.05	0.50	0.80
		104.5	1.7	5	16	19	20	911	0.39	0.19	0.05	0.55	1.18
		140.4	2.7	5	17	19	23	952	0.39	0.11	0.06	0.59	1.15
SNR	async	1609.5		12	32	33	67	2993	13.55	0.61	0.25	1.78	16.19
		2280.0		12	32	30	63	2583	12.75	0.59	0.25	1.75	15.34
		3121.8		12	32	30	63	2583	12.83	0.59	0.25	1.73	15.40
	sync	1609.5	8.7	11	32	31	1018	2509	3.24	0.63	0.41	2.14	6.42
		2280.0	3.2	12	32	32	1947	–	6.50	0.66	0.74	2.34	10.24
		3121.8	4.3	12	32	36	2083	–	6.36	0.61	0.75	2.19	9.91
	direct	1609.5	8.7	11	32	31	66	2412	4.03	0.59	0.22	2.11	6.95
		2280.0	3.2	12	32	32	67	2791	8.05	0.66	0.23	2.19	11.13
		3121.8	4.3	12	32	36	62	2810	8.19	0.63	0.27	2.39	11.48
decoder	async	186.3		5	28	17	33	1155	1.38	0.56	0.23	8.52	10.69
		279.4		5	28	17	33	1155	1.31	0.55	0.23	8.38	10.47
		404.8		5	28	17	33	1155	1.33	0.55	0.24	8.42	10.54
	sync	186.3	2.7	6	28	16	71	975	1.08	0.61	0.06	8.92	10.67
		279.4	2.7	5	28	16	75	957	1.24	0.58	0.06	8.78	10.66
		404.8	4.6	5	28	17	49	957	1.30	0.61	0.06	8.75	10.72
	direct	186.3	2.7	6	28	16	30	1140	1.06	0.59	0.22	9.77	11.64
		279.4	2.7	5	28	16	31	1134	1.19	0.56	0.22	9.33	11.30
		404.8	4.6	5	28	17	32	1138	1.24	0.66	0.22	9.64	11.76
fht	async	170.1		21	48	76	94	6993	184.02	0.28	0.22	0.38	184.90
		263.2		18	48	71	103	4121	183.84	0.28	0.20	0.41	184.73
		347.6		18	46	70	106	4023	182.91	0.33	0.16	0.41	183.81
	sync	170.1	0.7	21	47	84	621	7095	19.03	0.30	0.38	0.59	20.30
		263.2	1.5	19	46	78	421	4178	28.78	0.33	0.31	0.55	29.97
		347.6	2.2	20	51	81	237	4165	28.50	0.31	0.23	0.47	29.51
	direct	170.1	0.7	21	47	84	70	6929	18.36	0.30	0.14	0.78	19.58
		263.2	1.5	19	46	78	76	3911	28.34	0.31	0.20	0.84	29.69
		347.6	2.2	20	51	81	63	4204	29.42	0.31	0.19	0.86	30.78
pred1	async	3135.6		22	64	48	107	4405	145.69	1.75	0.98	3.20	151.62
		4914.0		16	63	52	108	4120	163.55	1.52	0.80	3.20	169.07
		6286.8		15	64	52	110	4476	179.88	1.77	0.97	3.36	185.98
	sync	3135.6	2.6	28	65	59	1919	–	9.13	1.86	0.66	3.81	15.46
		4914.0	6.0	19	65	55	944	3961	19.92	1.67	0.34	3.47	25.40
		6286.8	3.1	21	64	59	1889	–	26.41	1.86	0.64	3.66	32.57
	direct	3135.6	2.6	28	65	59	89	4532	9.00	1.70	0.73	4.39	15.82
		4914.0	6.0	19	65	55	90	4484	19.56	1.86	0.66	4.22	26.30
		6286.8	3.1	21	64	59	98	4740	26.80	1.78	0.66	4.25	33.49
mdct	async	112.5		27	67	71	59	4008	176.47	0.38	0.16	0.28	177.29
		207.9		15	62	76	86	4240	183.52	0.36	0.16	0.33	184.37
		290.4		17	70	73	95	4327	184.16	0.38	0.17	0.30	185.01
	sync	112.5	2.5	26	66	74	63	4540	16.14	0.41	0.16	0.36	17.07
		207.9	2.1	24	69	74	79	4565	26.06	0.41	0.17	0.52	27.16
		290.4	2.2	25	71	71	91	4574	25.23	0.39	0.17	0.38	26.17
	direct	112.5	2.5	26	66	74	22	4058	15.44	0.39	0.13	0.69	16.65
		207.9	2.1	24	69	74	63	4215	25.44	0.38	0.17	0.81	26.80
		290.4	2.2	25	71	71	68	4401	26.05	0.47	0.24	0.83	27.59
quant	async	2059.2		8	26	23	38	1828	1.31	0.17	0.13	0.47	2.08
		3088.8		8	26	23	38	1829	1.50	0.16	0.13	0.47	2.26
		4118.4		8	26	23	38	1829	1.55	0.14	0.14	0.44	2.27
	sync	2059.2	7.8	8	26	24	399	1502	0.34	0.16	0.14	0.52	1.16
		3088.8	7.8	8	26	22	264	1354	1.25	0.19	0.09	0.66	2.19
		4118.4	7.8	8	26	22	264	1354	1.34	0.17	0.11	0.55	2.17
	direct	2059.2	7.8	8	26	24	32	1748	0.33	0.16	0.13	0.77	1.39
		3088.8	7.8	8	26	22	34	1765	1.25	0.17	0.14	0.83	2.39
		4118.4	7.8	8	26	22	34	1772	1.50	0.19	0.16	0.84	2.69

Table 6 Synthesis results under a set of resource constraints.

name	circuit	rc	t_{step}	resource usage		states	latency	area	run-time [s]				
			[ns]	Regs	Muxs		[ns]	[slice]	S	RA	CS	others	total
usqrt	async	10		10	12	11	41.4	663	0.06	0.05	0.02	0.20	0.33
		8		10	12	13	52.3	677	0.08	0.03	0.03	0.14	0.28
	sync	10	0.9	10	12	45	40.5	586	0.13	0.05	0.03	0.20	0.41
		8	1.8	10	11	30	54.0	552	0.06	0.09	0.02	0.14	0.31
	direct	10	0.9	10	12	11	31.0	668	0.11	0.06	0.02	0.41	0.60
		8	1.8	10	11	12	39.4	694	0.09	0.09	0.03	0.45	0.66
fdct	async	7		16	18	23	102.3	994	0.11	0.06	0.03	0.17	0.37
		6		17	20	26	111.4	972	0.13	0.06	0.05	0.16	0.40
	sync	7	2.7	16	19	44	118.8	901	0.13	0.08	0.03	0.24	0.48
		6	1.7	16	21	76	129.2	814	0.16	0.06	0.03	0.17	0.42
	direct	7	2.7	16	19	16	95.8	956	0.16	0.09	0.05	0.52	0.82
		6	1.7	16	21	21	119.6	864	0.25	0.08	0.06	0.56	0.95
SNR	async	15		32	35	46	1648.9	2497	1.09	0.61	0.19	1.77	3.66
		11		32	35	51	3009.3	2706	2.33	0.59	0.22	1.80	4.94
	sync	15	0.9	32	35	1789	1610.1	–	3.27	0.59	0.97	2.41	7.24
		11	0.9	32	34	3335	3001.5	–	168.09	0.59	1.44	2.64	172.76
	direct	15	0.9	32	35	45	1624.5	3031	3.80	0.70	0.25	2.39	7.14
		11	0.9	32	34	59	3128.7	2820	173.59	0.73	0.30	2.22	176.84
decoder	async	8		28	18	31	147.2	1176	1.00	0.56	0.19	8.36	10.11
		7		28	18	33	148.5	1171	1.00	0.55	0.22	8.37	10.14
	sync	8	1.0	28	17	190	190.0	1052	1.19	0.58	0.09	8.63	10.49
		7	1.6	28	17	124	198.4	983	1.08	0.56	0.06	8.41	10.11
	direct	8	1.0	28	17	31	152.8	1132	1.30	0.58	0.25	10.14	12.27
		7	1.6	28	17	32	156.1	1128	1.11	0.61	0.23	9.83	11.78
fht	async	24		47	83	57	191.1	4160	14.89	0.27	0.17	0.38	15.71
		21		46	73	65	243.5	3919	20.31	0.25	0.16	0.36	21.08
	sync	24	0.7	47	85	286	200.2	4364	1.17	0.33	0.34	0.52	2.36
		21	1.1	46	80	221	243.1	4008	1.17	0.31	0.30	0.45	2.23
	direct	24	0.7	47	85	58	250.3	4413	1.34	0.34	0.19	0.94	2.81
		21	1.1	46	80	68	281.2	4451	1.11	0.42	0.23	0.92	2.68
pred1	async	27		65	57	89	3175.3	4482	12.38	1.45	0.64	3.23	17.70
		23		65	59	100	4156.0	4327	15.88	1.53	0.70	3.19	21.30
	sync	27	3.3	66	69	1476	4870.8	–	26.00	1.92	0.52	3.66	32.10
		23	1.4	65	60	4715	6601.0	–	343.63	1.55	1.28	4.47	350.93
	direct	27	3.3	66	69	72	3228.8	4439	26.84	2.05	0.55	4.36	33.80
		23	1.4	65	60	93	3704.1	4248	342.70	1.70	0.73	4.35	349.48
mdct	async	30		76	94	49	242.8	7811	11.05	0.38	0.19	0.39	12.01
		27		75	87	49	275.0	7251	13.97	0.38	0.14	0.31	14.80
	sync	30	2.3	75	94	68	156.4	7993	0.49	0.52	0.20	0.41	1.62
		27	3.1	75	91	49	151.9	7241	0.45	0.48	0.20	0.34	1.47
	direct	30	2.3	75	94	37	287.1	7225	0.52	0.52	0.24	0.75	2.03
		27	3.1	75	91	33	327.3	6792	0.63	0.55	0.22	0.78	2.18
quant	async	11		26	21	34	1973.7	1563	0.28	0.16	0.13	0.45	1.02
		10		25	22	35	1977.0	1590	0.30	0.16	0.13	0.47	1.06
	sync	11	2.6	26	22	780	2028.0	1611	1.42	0.17	0.27	0.78	2.64
		10	2.6	25	22	780	2028.0	1589	1.44	0.16	0.23	0.69	2.52
	direct	11	2.6	26	22	34	1532.1	1530	1.45	0.16	0.16	0.92	2.69
		10	2.6	25	22	34	1536.7	1509	1.47	0.19	0.16	0.88	2.70

of control circuits and the wire delays of delay elements in addition to the logic delays of delay elements in the generation of delay elements. If delay elements are optimized well, we may obtain better circuits than the synchronous counterparts. Circuits where the number of resources is less than the synchronous counterpart are such candidates. For example, in the case of usqrt with time constraint 86.0 ns, the number of slices used in the delay elements is 42. If we can reduce more than 10 slices in the delay elements, the area of the synthesized circuits by the proposed system becomes less than the synchronous counterpart. This optimization will be considered in our future work.

Compared to the direct implementations, the experimental results may not show a large difference between the synthesized circuits by the proposed system and the direct implementations. The proposed system synthesizes better circuits or worse circuits which depends on the number of resources. This is because the heuristic nature of the proposed system. However, the proposed system has much more possibility to synthesize the best circuit. mdct is such a case. As a different data-path circuit is synthesized by using non-uniform control steps, the difference in the number of resources results in less area. On the other hand, in the direct implementations, it is difficult to synthesize a better circuit than the synchronous counterpart. This is because the data-path circuit is the same as the synchronous counterpart, but the control circuit has delay elements.

Latency. As the main objective of resource constraint scheduling is to minimize latency, we discuss the impact of latency in the proposed system referring to Table 6.

Compared to the synchronous counterparts and the direct implementations, the proposed system synthesizes the best circuits in many cases (e.g., all cases of fdct, decoder, and pred1). Operations are scheduled so that they are executed immediately after the completion of previous operations using non-uniform control steps. In addition, the use of non-uniform control steps results in different schedules compared to the synchronous counterparts. On the other hand, the latency improvement of the direct implementations for the synchronous counterparts is restricted. This is because the same scheduling results are utilized although the control schemes are different.

Synthesis time. In behavioral synthesis under time constraints, the proposed system takes more time for scheduling because control steps are updated whenever an operation is scheduled. On the other hand, there is no big difference in behavioral synthesis under resource constraints. This is because the proposed system approximates start time candidates at control steps where the number of available functional units is less than the number of schedulable operations.

From the experimental results, we can say that the proposed system is preferable for behavioral synthesis of bundled-data implementations in that in many cases the proposed system synthesizes better circuits in terms of area and latency than direct implementations. Moreover, in several cases, the proposed system synthesizes better circuits than synchronous counterparts not only in latency but also in area. It is the effect of non-uniform control steps used in the proposed system.

6. Conclusions

This paper proposes a behavioral synthesis system for asynchronous circuits with bundled-data implementation. The proposed system is implemented in Java and evaluated through the experiments. The experimental results show the effectiveness of the proposed system in that the synthesized bundled-data implementations are superior to the synchronous counterparts and the direct implementations in many cases.

As our future work, we are going to extend the proposed system to synthesize a behavioral description with arrays and floating point operations. Moreover, pipelining, chaining, and other optimization techniques will be implemented.

Acknowledgments This work is partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) 18700047.

References

1) Open SystemC Initiative: http://www.systemc.org/home/.
2) Micheli, G.D.: *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Higher Education (1994).
3) Gupta, S., Gupta, R., Dutt, N. and Nicolau, A.: *SPARK: A Parallelizing Approach to the High-level Synthesis of Digital Circuits*, Kluwer Academic Publishers (2004).
4) Badia, R. and Cortadella, J.: High-level synthesis of asynchronous systems:

scheduling and process synchronization, *Proc. European Conference on Design Automation*, pp.70–74 (1993).
5) Bachman, B., Zheng, H., and Myers, C.: Architectural synthesis of timed asynchronous systems, *Proc. International Conference on Computer Designs*, pp.354–363 (1999).
6) Nielsen, S.F., Sparso, J., and Madsen, J.: Towards behavioral synthesis of asynchronous circuits — an implementation template targeting syntax directed compilation, *Proc. EUROMICRO Symposium on Digital System Design*, pp.298–307 (2004).
7) Sacker, M., Brown, A.D., Rushton, A.J., Wilson, P.R.: A General Purpose Behavioral Architectural Synthesis System, *Proc. International Symposium on Asynchronous Circuits and Systems*, pp.125–134 (2004).
8) Venkataramani, G., Budiu, M., Chelcea, T. and Goldstein, S.: C to asynchronous dataflow circuits: An end-to-end toolflow (2004).
9) Edwards, D.A. and Bardsley, A.: Balsa: An Asynchronous Hardware Synthesis Language, *The Computer Journal*, Vol.45, No.1, pp.12–18 (2002).
10) Hoare, C.A.R.: *Communicating sequential processes*, Prentice-Hall (1985).
11) Hamada, N., Shiga, Y., Saito, H., Yoneda, T., Myers, C., and Nanya, T.: A Behavioral Synthesis Method for Asynchronous Circuits with Bundled-data Implementation, *Proc. International Conference on Application of Concurrency to System Design*, (2008) (to appear).
12) Rosenberger, F., Molnar, C., Chaney, T. and Fang, T.-P.: Q-Modules: Internally Clocked Delay-Insensitive Modules, *IEEE Transactions on Computers*, Vol.37, No.9, pp.1005–1018 (1988).
13) COINS-project: A compiler infra structure, http://www.coins-project.org/.
14) Paulin, P.G. and Knight, J.P.: Force-Directed Scheduling for the Behavioral Synthesis of ASIC's, *IEEE Transactions on Computer-Aided Design*, Vol.8, No.6, pp.661–679 (1989).
15) Saito, H., Hamada, N., Jindapetch, N., Yoneda, T., Myers, C., and Nanya, T.: Scheduling Methods for Asynchronous Circuits with Bundled-Data Implementations Based on the Approximation of Start Times, *IEICE Transaction*, Vol.E90-A, No.12, pp.2790–2799 (2007).
16) Tseng, C.-J., Wei, R.-S., Rothweiler, S.G., Tong, M.M. and Bose, A.K.: Bridge: a versatile behavioral synthesis system, *Proc. Design Automation Conference*, pp.415–420 (1988).
17) MiBench version 1.0. http://www.eecs.umich.edu/mibench/
18) MediaBenchII Benchmark. http://euler.slu.edu/fritts/mediabench/mb2/index.html/
19) Xilinx: ISE WebPACK 9.2i. http://www.xilinx.com/
20) Mentor Graphics: ModelSim 6.2g. http://www.mentor.com/

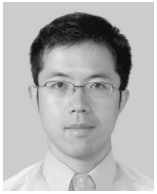
(Received May 23, 2008)
(Revised August 22, 2008)
(Accepted October 9, 2008)
(Released February 17, 2009)

(Recommended by Associate Editor: *Yusuke Matsunaga*)

Naohiro Hamada received the B.S. and M.S. degrees in Computer Science from the University of Aizu in 2006 and 2008, respectively. Currently, he is a Ph.D. student at the University of Aizu.

Yuki Shiga received the B.S. degree in Computer Science from the University of Aizu in 2007. Currently, he is a M.S. student at the University of Aizu.

Takao Konishi received the B.S. degree in Computer Science from the University of Aizu in 2006. Currently, he is a M.S. student at the University of Aizu.



Hiroshi Saito received the B.S. and M.S. degrees in Computer Science from the University of Aizu in 1998 and 2000, respectively. In 2003, he received the Ph.D. in Interdisciplinary Course on Advanced Science and Technology from the University of Tokyo. Currently, he is an assistant professor at the University of Aizu. His research interests include synthesis of asynchronous circuits and formal verification.

Tomohiro Yoneda received B.E., M.E., and Dr. Eng. degrees in Computer Science from the Tokyo Institute of Technology, Tokyo, Japan in 1980, 1982, and 1985, respectively. In 1985 he joined the staff of Tokyo Institute of Technology, and he moved to National Institute of Informatics in 2002, where he is currently a Professor. He was a visiting researcher of Carnegie Mellon University from 1990 to 1991. His research activities currently focus on formal verification of hardware and synthesis of asynchronous circuits. Dr. Yoneda is a member of IEEE, Institute of Electronics, Information, and Communication Engineers of Japan, and Information Processing Society of Japan.

Chris Myers received the B.S. degree in electrical engineering and Chinese history in 1991 from the California Institute of Technology, Pasadena, CA, and the M.S.E.E. and Ph.D. degrees from Stanford University, Stanford, CA, in 1993 and 1995, respectively. He is a Professor in the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT. Dr. Myers is the author of over 80 technical papers and the textbook Asynchronous Circuit Design. He is also a co-inventor on 4 patents. His research interests include algorithms for the analysis of real-time concurrent systems, analog error control decoders, formal verification, asynchronous circuit design, and the modeling and analysis of genetic regulatory circuits. Dr. Myers received an NSF Fellowship in 1991, an NSF CAREER award in 1996, and best paper awards at Async1999 and Async2007.

Takashi Nanya received his B.S. and M.S. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1969 and 1971, respectively, and his Ph.D. degree in electrical engineering from the Tokyo Institute of Technology, Tokyo, Japan in 1978. He was with the NEC Central Research Laboratories from 1971 to 1981, and on the faculty of Tokyo Institute of Technology from 1981 to 1996. In 1996, he joined the University of Tokyo where he is a professor. From 2001 to 2004, he served as the director of the Research Center for Advanced Science and Technology, and a councilor of the University of Tokyo. His research interests include dependable computing, computer architecture, VLSI design and asynchronous computing. He is currently serving as the Chair of IEEE-CS TC on Dependable Computing and Fault Tolerance and also the Chair of the Steering Committee of International Conference on Dependable Networks and Systems. He is a Fellow of IEEE and IEICE, and a Professor Emeritus of the Tokyo Institute of Technology, and a member of the Science Council of Japan.