IEICE TRANS. INF. & SYST., VOL.E96–D, NO.8 AUGUST 2013

PAPER Special Section on Reconfigurable Systems

Architecture of an Asynchronous FPGA for Handshake-Component-Based Design

Yoshiya KOMATSU[†a)], ***Nonmember***, **Masanori HARIYAMA**[†], ***Member***, ***and*** **Michitaka KAMEYAMA**[†], ***Fellow***

SUMMARY This paper presents a novel architecture of an asynchronous FPGA for handshake-component-based design. The handshake-component-based design is suitable for large-scale, complex asynchronous circuit because of its understandability. This paper proposes an area-efficient architecture of an FPGA that is suitable for handshake-component-based asynchronous circuit. Moreover, the Four-Phase Dual-Rail encoding is employed to construct circuits robust to delay variation because the data paths are programmable in FPGA. The FPGA based on the proposed architecture is implemented in a 65 nm process. Its evaluation results show that the proposed FPGA can implement handshake components efficiently.

key words: *FPGA, reconfigurable LSI, self-timed circuit, asynchronous circuit*

1. Introduction

Recent technology scaling enables designs with billions of transistors. On the other hand, the increased complexity of circuits leads to two problems. The first is the cost problem. The process development cost has increased the expense of the fabrication cost of chips. Also, design cost and verification cost become serious problem. The second is performance problem. Currently, most digital circuits are synchronous circuits which operate based on clock signals. As the number of transistors integrated on a chip has increased, clock distribution network has become complex and its power consumption has become large. In addition, it becomes severe challenge to increase clock frequency because clock signal should be distributed all over a chip.

To solve the first problem, Field-programmable gate arrays (FPGAs) are widely used to implement special-purpose processors. Since users can program logic functions and interconnections of FPGAs directly, it is easy to develop special-purpose processors. In addition, FPGAs are cost-effective because they are produced in large quantities.

To solve the second problem, asynchronous circuit is attracting attention. In asynchronous circuit, data transfer is done by handshaking using a request signal and an acknowledge signal. Since no clock signal is necessary, problems caused by clock distribution network do not arise. However, the problem is that it is difficult to design asynchronous circuits.

As the design methods for asynchronous circuits, handshake-component-based design [1] was proposed. In handshake-component-based design, asynchronous circuits are designed by connecting handshake components. Since various handshake components such as for data processing and data path control are defined, it is easy to design asynchronous data path and its controller. Therefore, handshake-component-based design is suitable for applications that contain complex data processing. Besides, Balsa [2] is proposed as a design methodology that uses handshake components. Balsa is a hardware description language and it allows circuit designers not to pay attention to low-level details such as control of handshake. Moreover, there are synthesis tools that generate handshake circuits which consist of handshake components and standard cell netlists from Balsa descriptions. Using Balsa, circuit designer can easily implement complex large-scale circuits such as a DMA controller [2] and a microprocessor [3]. Thus, handshake-component-based design is suitable for complex large-scale asynchronous circuits.

To solve the cost and performance problems, some asynchronous FPGAs has been proposed [4]–[10]. Asynchronous FPGAs developed by Cornell University [4], [5], Achronix [6] and the University of Tokyo [7] employ fine-grained pipelined architecture to achieve high throughput. References [8]–[10] propose asynchronous FPGA architecture focusing on low power consumption. The asynchronous FPGA proposed in [8], [10] combine two handshake protocols to reduce energy consumption caused by data operations and transmissions. Reference [9] proposes autonomous power-gating scheme based on handshake protocol. However, conventional asynchronous FPGAs cannot implement handshake components efficiently since their architecture only support simple handshake sequence specialized for simple data processing and transferring. Therefore, it is difficult to design control-intensive application on conventional FPGAs.

In this paper, we propose an FPGA architecture that is suitable for handshake-component-based asynchronous circuit. The proposed architecture implements handshake components that are defined in Balsa efficiently. Therefore, the proposed FPGA is suitable for implementing complex applications. Small frequently-used handshake components are implemented on a Logic Block (LB), and other handshake components are implemented using more than one LB. As handshake components can be mapped directly on the proposed architecture, circuit designers can utilize existing CAD tools that generate a netlist of handshake components. Therefore, a design method for the proposed FPGA

Manuscript received November 10, 2012.
Manuscript revised March 8, 2013.

[†]The authors are with the Graduate School of Information Sciences, Tohoku University, Sendai-shi, 980–8579 Japan.
a) E-mail: ykomatsu@ecei.tohoku.ac.jp

DOI: 10.1587/transinf.E96.D.1632

is established.

2. Handshake-Component-Based Asynchronous Circuit Design

2.1 Handshake Component

In asynchronous circuit, synchronization between circuits is done by handshaking with a request signal and an acknowledge signal. Figure 1 shows a four-phase handshake sequence. First, active port sets the request wire to "1" as shown in Fig. 1 (a). Second, passive port sets the acknowledge wire to "1" as shown in Fig. 1 (b). Third, active port sets the request wire to "0" as shown in Fig. 1 (c). Finally, passive sets the acknowledge wire to "0" as shown in Fig. 1 (d) and wire values return to initial state. Data signals are sent along with request signals or acknowledge signals.

To design asynchronous circuits, various design methodologies has been proposed. Petrify [11] is an asynchronous circuit synthesis tool that uses a Signal Transition Graph (STG) [12]. STG describes transition sequences of wires. Therefore, STG is suitable to describe control circuits. However, it is difficult to design circuits which contain many wires. Another design methodology uses asynchronous circuit elements called handshake components. Asynchronous circuits are constructed by connecting handshake components. Handshake components were created for use in the synthesis of the language Tangram [1] created by Philips Research. Figure 2 shows handshake components. Handshake components constitute a handshake circuit. Each handshake component has ports and is connected to another handshake component through a channel. Communication between handshake components is done by sending request signal from the "active" port and acknowledge signal from the "passive" port. Depending on the kind of handshake components, data signals are sent along with request signals or acknowledge signals. The number of ports of a handshake component and the width of data signal can be varied. There are 46 handshake components [13] and each handshake component is used for data processing or data path control. Figure 3 shows a Sequence component. Sequence component has an *activate* port and N *activateOut* ports. Sequence component starts handshaking sequentially from *activateOut*0 to *activateOutN* − 1. Then, handshake component connected to each *activateOut* port is activated. In this manner, Sequence component controls process sequence. Figure 4 shows signal transitions of a Sequence component. Arrows denote dependencies between signal transitions. The behavior of a Sequence component which has two *activateOut* ports is described as follows:

1. *activate.req* is set to "1"
2. *activateOut*0.*req* is set to "1"
3. *activateOut*0.*ack* is set to "1"
4. *activateOut*0.*req* is set to "0"
5. *activateOut*0.*ack* is set to "0"

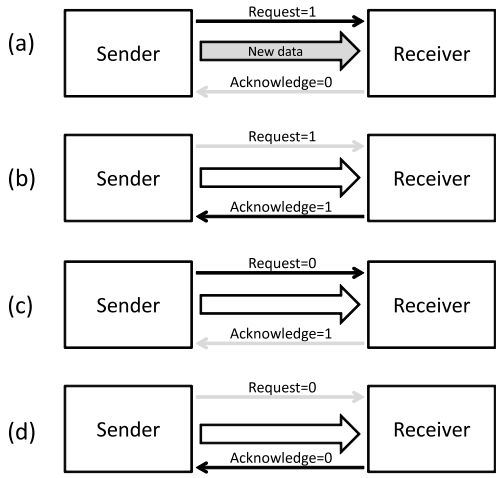


Fig. 1 A four-phase handshake sequence.

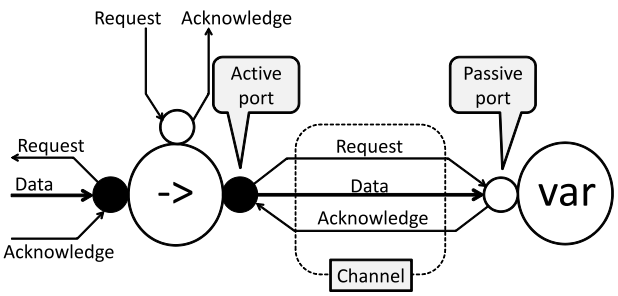


Fig. 2 Handshake components and channels.

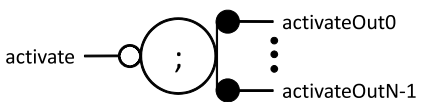


Fig. 3 Sequence component.

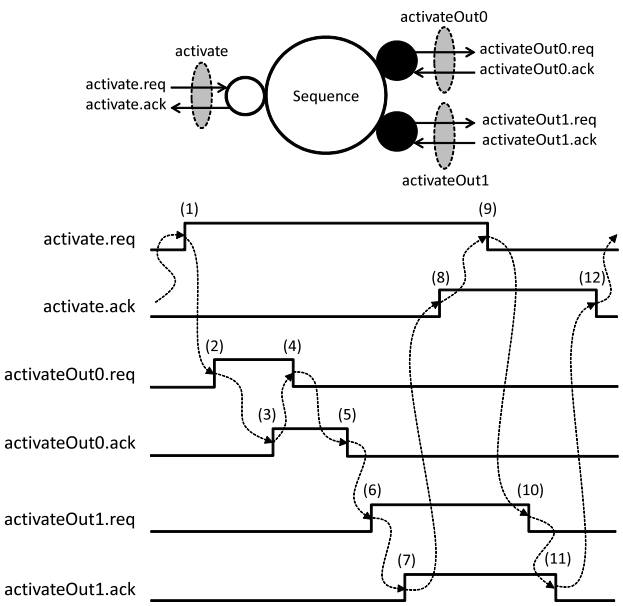


Fig. 4 Behavior of a Sequence component.

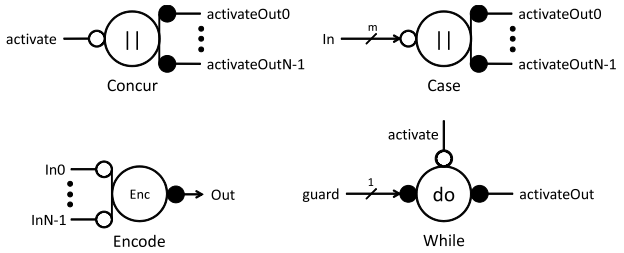


Fig. 5 Handshake components for a data path controller.

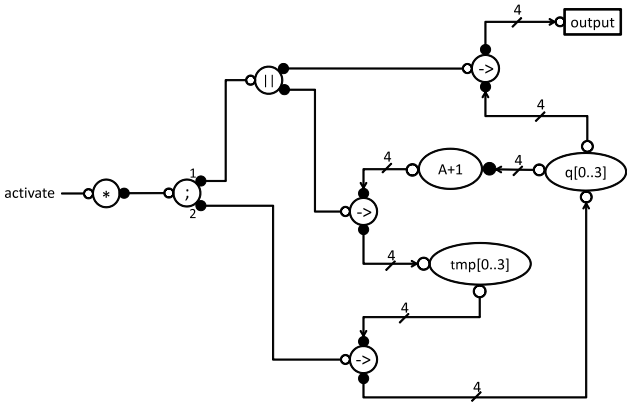


Fig. 6 A simple handshake circuit (4 bit counter).

6. *activateOut*1.*req* is set to "1"
7. *activateOut*1.*ack* is set to "1"
8. *activate.ack* is set to "1"
9. *activate.req* is set to "0"
10. *activateOut*1.*req* is set to "0"
11. *activateOut*1.*ack* is set to "0"
12. *activate.ack* is set to "0"

As seen above, handshake components execute complex handshake sequences. However, handshake circuits are easily understandable and manageable because a function of each handshake component is clear and each handshake is symbolized by a channel and ports. Asynchronous circuits with complex process control are designed using handshake components shown in Fig. 5. Figure 6 shows an example of a handshake circuit.

Also, there are tools that translate high-level circuit description into handshake circuit to synthesize asynchronous circuit. Thus, handshake-component-based design is suitable for complex and large-scale asynchronous circuits.

2.2 Implementation of Handshake Components

Circuit synthesis is done by replacing each handshake component with corresponding asynchronous circuit. Therefore, implementations in different technologies are obtained by providing circuit libraries. In asynchronous circuit, a hazard is serious problem [14]. A hazard is a unwanted glitch on a signal and it causes a malfunction. To guarantee correct operation of implemented application, asynchronous circuits that corresponds to handshake components should be hazard free.

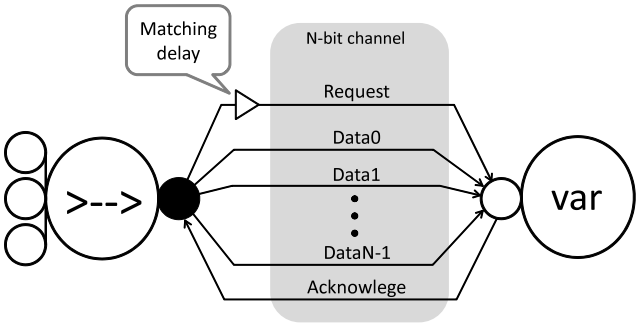


Fig. 7 A bundled-data channel.

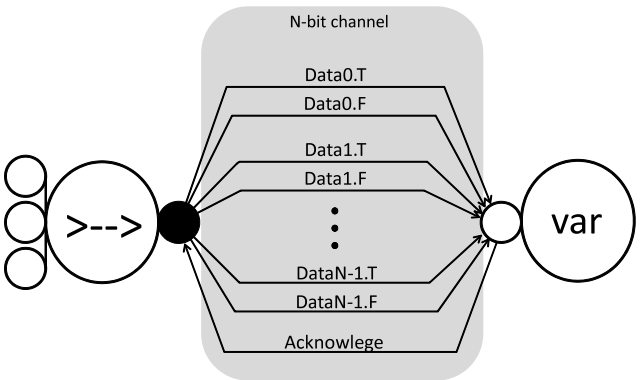


Fig. 8 A four-phase dual-rail channel.

In handshake-component-based design, implementations in different asynchronous data encodings are obtained by changing circuit libraries. Asynchronous data encoding schemes are mainly classified into

- Single-rail encoding (ex. bundled-data encoding)
- Dual-rail encoding (ex. four-phase dual-rail encoding)

Bundled-data encoding is the most common method in the single-rail encoding. Figure 7 shows a bundled-data channel. The value is encoded as in a synchronous circuit using N wires to denote an N-bit number, and control signals are encoded using dedicated wires denoted by REQ and ACK. Therefore, a channel which contains N-bit data consists of $N + 2$ wires. Bundled-data encoding requires the explicit insertion of matching delays in a control signal oriented in the same direction as data signal. This is because the control signal is never received before the bundled value is valid. For FPGAs, since the data path is programmable, complex programmable delay elements are required. As a result, bundled-data encoding is not suitable for FPGAs.

Four-phase dual-rail (FPDR) encoding is the most common method in dual-rail encodings. Figure 8 shows a FPDR channel. The FPDR encoding encodes a bit and a control signal oriented in the same direction as data signal onto two wires. Table 1 shows the code table of four-phase dual-rail encoding. The data value "0" is encoded as $(0, 1)$ and "1" is encoded as $(1, 0)$. Moreover, the spacer is encoded as $(0, 0)$. Figure 9 shows the example where data val-

Table 1 Code table of four-phase dual-rail encoding.

	Code word (T, F)
Data 0	(0,1)
Data 1	(1,0)
Spacer	(0,0)

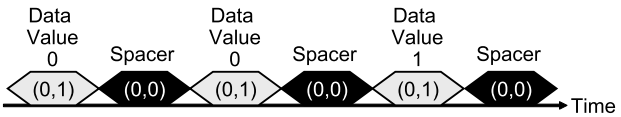


Fig. 9 Example of four-phase dual-rail encoding.

ues "0", "0" and "1" are transferred. The main feature is that the sender sends spacer after a data value. The receiver knows the arrival of a data value by detecting the change of either bit: "0" to "1". In the FPDR encoding, the value is made implicit in a control signal and no delay insertion is therefore required [14]. Hence, the FPDR encoding is robust to delay variations and the ideal one for FPGAs in which the data path is programmable. In the dual-rail encoding, to transfer an N-bit value, $2N+1$ wires are required. Therefore, the FPDR encoding is employed in the proposed architecture.

3. Architecture

3.1 Overall Architecture

Figure 10 shows the overall architecture of the proposed FPGA and Fig. 11 shows the programmable interconnection resources (Connection Blocks and Switch Blocks) around an LB. The FPGA consists of mesh-connected cells like conventional FPGAs. As shown in Fig. 10, each cell includes an LB, two Connection Blocks (CBs) and a Switch Block (SB). The upper CB connects SBs to N1, N2 and S terminals of two LBs, and the bottom CB connects SBs to E1, E2 and W terminals. The proposed architecture can implement 39 out of 46 handshake components defined in Balsa manual [13]. Handshake components that have multiple ports or wide data path can be implemented using several LBs. As mentioned in Sect. 2.2, the FPDR encoding is employed for asynchronous data encoding. Because the FPDR encoding is employed, three wires are required for a data bit. Two wires are used for a data encoded in FPDR encoding, and one wire for a request signal and an acknowledge signal. The proposed FPGA is based on Quasi-Delay-Insensitive (QDI) model which assumes that gate delays and wire delays are unknown, and signal transitions occur at the same time at all end-points in wire forks [14], [15].

As shown in Fig. 11, an SB consists of diamond switches and Req/Ack modules. Diamond switches allow a data signal on a track to connect to other tracks. Figure 12 shows the structure of the Req/Ack module. The Req/Ack module consists of switches, an OR gate and the Muller C-element [14]. It allows a control signal on a track to connect to other tracks. In addition, two control signals can be merged using a C-element or an OR gate. The LB accesses nearby communication resources through CBs, which connects input and output terminals of the LB to SBs through programmable switches.

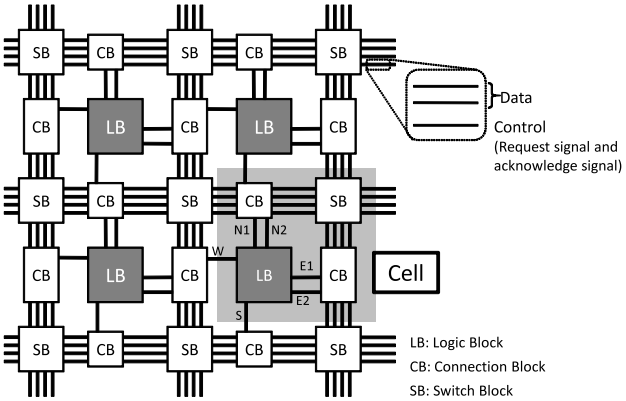


Fig. 10 Overall architecture.

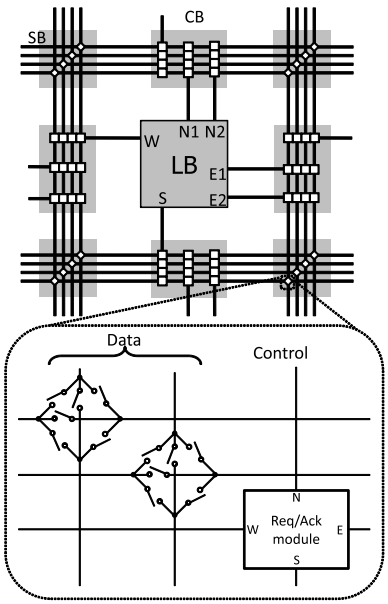


Fig. 11 Programmable interconnection resources around an LB.

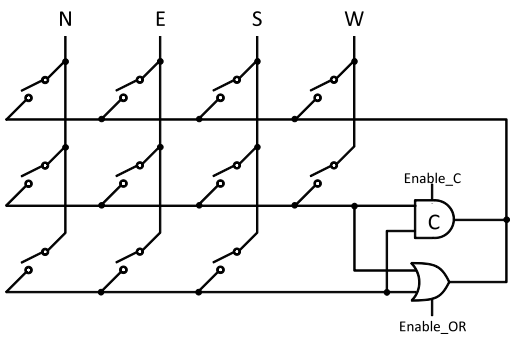


Fig. 12 Structure of an Req/Ack module.

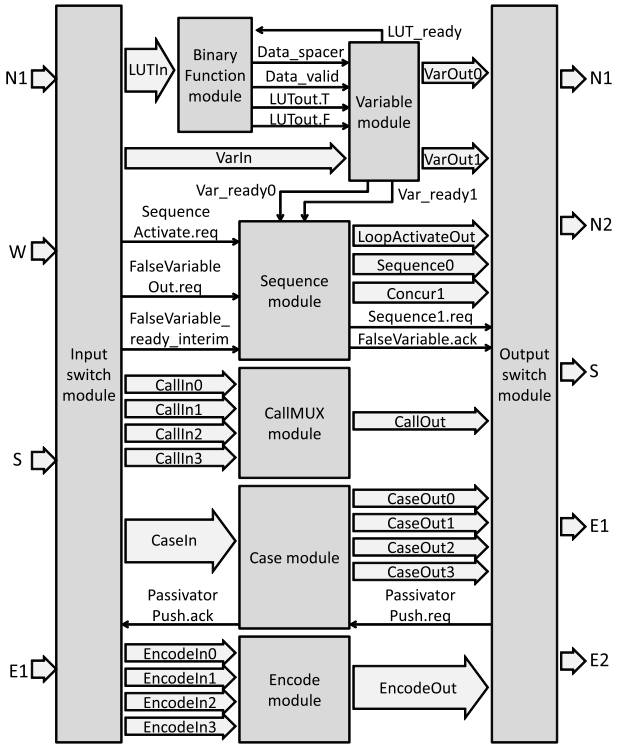


Fig. 13 Structure of an LB.

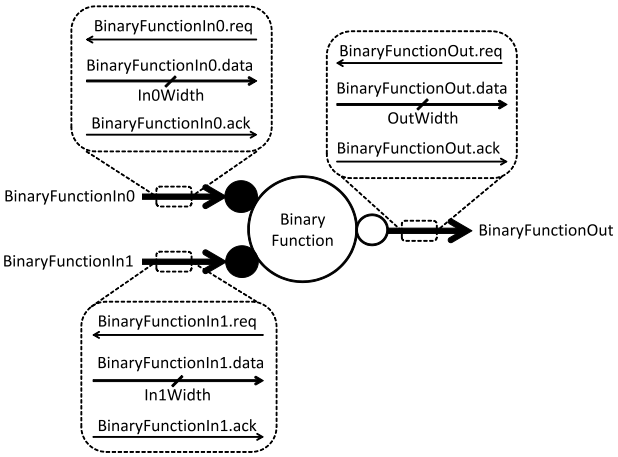


Fig. 14 BinaryFunction component.

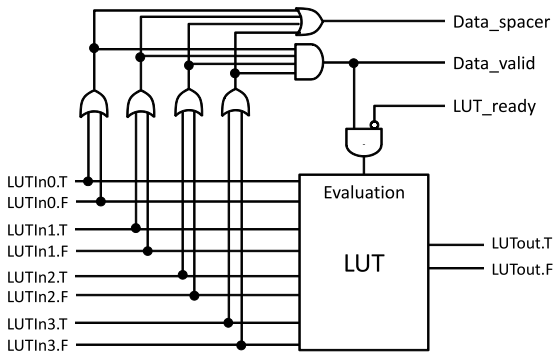


Fig. 15 Structure of a BinaryFunction module.

3.2 Logic Block Structure

3.2.1 Overall Structure of a Logic Block

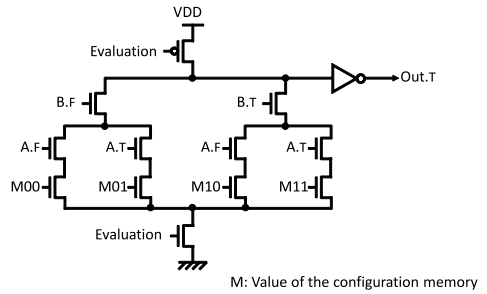
Figure 13 shows an LB of the proposed architecture. The proposed FPGA architecture can implement 39 handshake components. The LB consists of a BinaryFunction module, a Variable module, a Sequence module, a CallMUX module, a Case module, and an Encode module. An Input switch module and an Output switch module connect modules to CBs. As mentioned in previous section, circuit synthesis is done by replacing each handshake component with corresponding asynchronous circuit. Thus, asynchronous circuits can be implemented on a conventional FPGA by replacing each handshake component with a combination of LUTs. As mentioned in Sect. 2.2, asynchronous circuits that implement handshake components should be hazard free. However, it is difficult to implement hazard free asynchronous circuits using LUTs because delay time of SBs and CBs affects circuit operations. Therefore, in the proposed architecture, each LB includes dedicated circuits for implementing handshake components.

3.2.2 BinaryFunction Module Structure

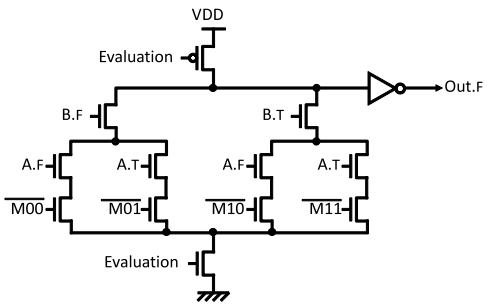
In handshake-component-based design, logical operation and arithmetic operation are denoted by BinaryFunction components as shown in Fig. 14. As mentioned in Sect. 3.1, the proposed architecture employs the FPDR encoding that encodes a bit and a control signal onto two wires. Therefore, acknowledge signals of *BinaryFunctionIn*0, *BinaryFunctionIn*1 and *BinaryFunctionOut* port are sent along with data signals. In the proposed architecture, a BinaryFunction module is used to implement a BinaryFunction component. Figure 15 shows a structure of a BinaryFunction module. the module consists of an FPDR 4-input LUT and logic gates that detect arrival of valid data and spacers. When valid signals arrive at the *LUTIn*, *Data_valid* becomes "1" and the LUT starts to operate. The result of the LUT is stored in the Variable module. Then, *LUT_ready* is set to "1" and the LUT stops its operation. Figure 16 shows the structure of the LUT. For simplicity, instead of the 4-input LUT which is used in the actual LB, a 2-input LUT is shown. The LUT is implemented based on [4] and [14]. A BinaryFunction module can implement a BinaryFunction component with two 2-bit inputs or a BinaryFunction component with a 1-bit and a 3-bit input. A complex BinaryFunction component can be implemented by combining BinaryFunction modules.

3.2.3 Variable Module Structure

Figure 17 shows a Variable component that stores data. In the proposed architecture a Variable component is imple-



(a) $Out._T$ generating circuit.



(b) $Out._F$ generating circuit.

Fig. 16 LUT structure.

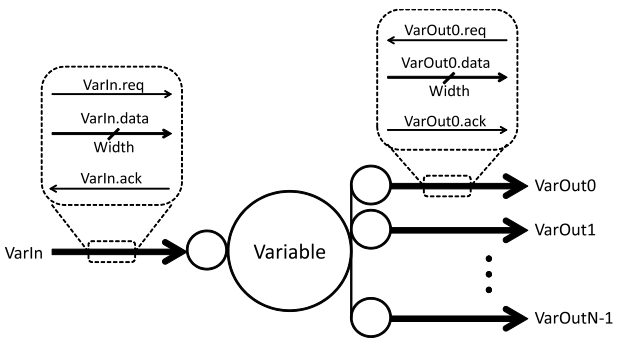


Fig. 17 Variable component.

mented using a Variable module shown in Fig. 18. The Variable module stores 2-bit data. The *VarIn* port is used to store 2-bit data and *VarOut*0 and *VarOut*1 ports are used to read 2-bit data. The Variable module mainly consists of Variable elements that store data, AND gates that generate output signals and C-elements. Figure 19 shows a structure of a Variable element. As shown in Fig. 20, Writing data is performed in the following sequence:

1. A valid data arrives at *VarIn* and the data is stored in Variable elements
2. *Var*0_*ready* and *Var*1_*ready* become "1"
3. *VarIn.ack* becomes "1"
4. A spacer arrives at *VarIn*
5. *Var*0_*ready* and *Var*1_*ready* become "0"
6. *VarIn.ack* becomes "0"

Figure 21 shows signal transitions of a Variable module in a read operation. Reading data from the *VarOut*0 port is performed in the following sequence:

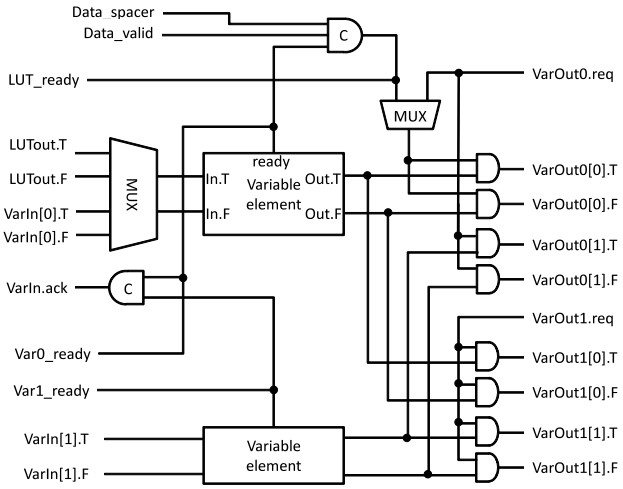


Fig. 18 Structure of a 2-bit Variable module.

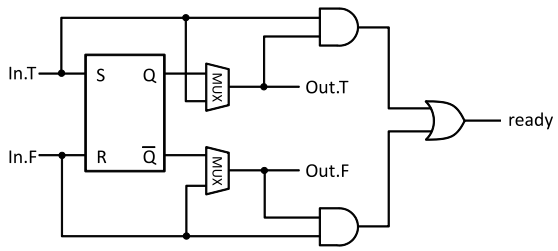


Fig. 19 Structure of a Variable element.

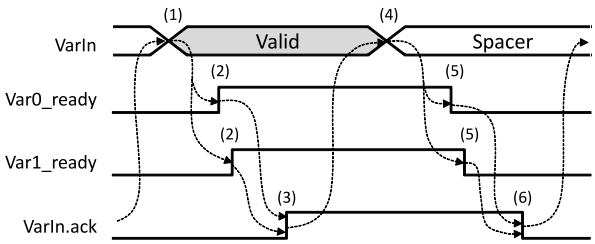


Fig. 20 Behavior of a Variable module in a write operation.

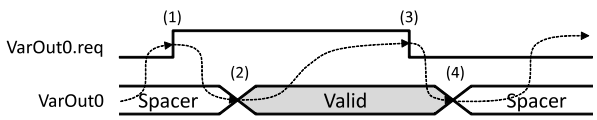


Fig. 21 Behavior of a Variable module in a read operation.

1. *VarOut*0.*req* is set to "1"
2. AND gates connected to *VarOut*0 output the data stored in Variable elements
3. *VarOut*0.*req* is set to "0"
4. AND gates output spacer

Reading data from the *VarOut*1 port is performed in a similar manner. In addition, a Variable module and a BinaryFunction module are used to implement BinaryFunction component. Figure 22 shows signal transitions of a BinaryFunction module and a Variable module. The behavior as a

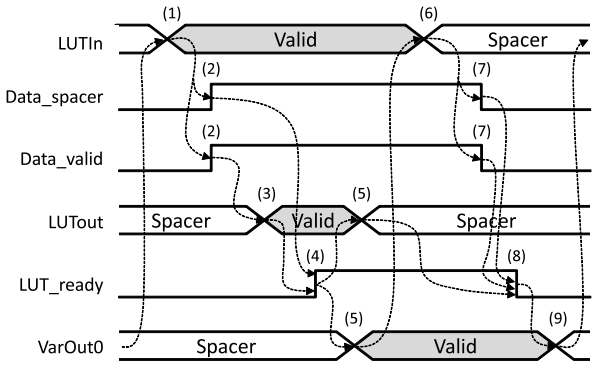


Fig. 22 Behavior of a BinaryFunction module and a Variable module as a BinaryFunction component.

BinaryFunction component is described as follows:

1. A valid data arrives at an *LUTIn* of a BinaryFunction module
2. *Data_spacer* and *Data_valid* become "1"
3. LUT starts operation following the rise of *Data_valid*
4. The Variable element connected to the LUT stores the data and then *LUT_ready* is set to "1"
5. *VarOut*0[0] outputs valid data, and *LUTout* outputs a spacer
6. A spacer arrives at the *LUTin*
7. *Data_spacer* and *Data_valid* become "0"
8. *LUT_ready* becomes "0"
9. VarOut0_0 outputs a spacer

3.2.4 Sequence Module Structure

Figure 23 shows a structure of a Sequence module. The Sequence module mainly consists of a T-element and an S/T-element. Figures 24 and 25 show detailed structure of a T-element and an S/T-element. A Sequence module can implement a Sequence component and a Concur component as shown in Figs. 26 and 27. Sequence component and Concur component are used to control a process sequence of a circuit. A Sequence module can implement two *activateOut* ports. Usually, a Sequence component is implemented by a S-element and a Concur component is implemented by two T-element and a C-element as shown in Fig. 28. In the proposed architecture, a Sequence component is implemented by S/T-element and a Concur component is implemented by T-element, S/T-element and C-element. When a Sequence component is implemented, *activate.req*, *activateOut*0.*req*, *activateOut*0.*ack* and *activateOut*1.*req* in Fig. 26 correspond to *SequenceActivate.req*, *Sequence*0.*req*, *sequence*0.*ack* and *Sequence*1.*req* in Fig. 23. Since *activate.ack* and *activateOut*1.*ack* are connected as shown in Fig. 28 (a), there is no dedicated wires in a Sequence module. Figure 29 shows signal transitions of a Sequence module as a Sequence component. The behavior as a Sequence component is described as follows:

1. *SequenceActivate.req* is set to "1"

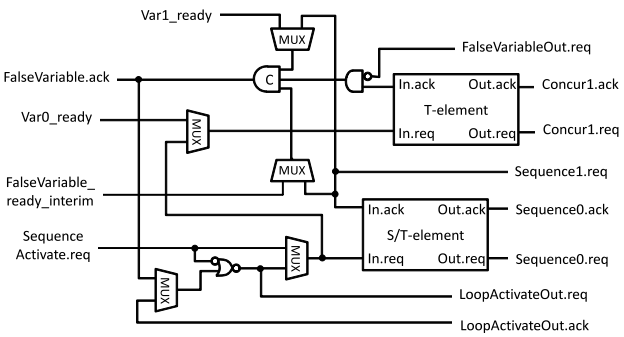


Fig. 23 Structure of a Sequence module.

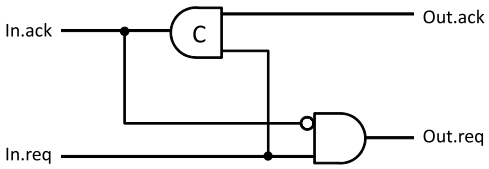


Fig. 24 Structure of a T-element.

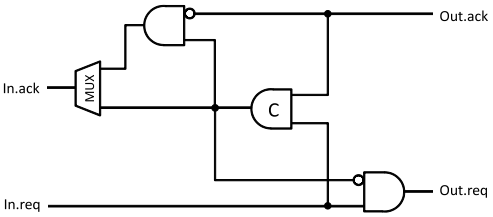


Fig. 25 Structure of a S/T-element.

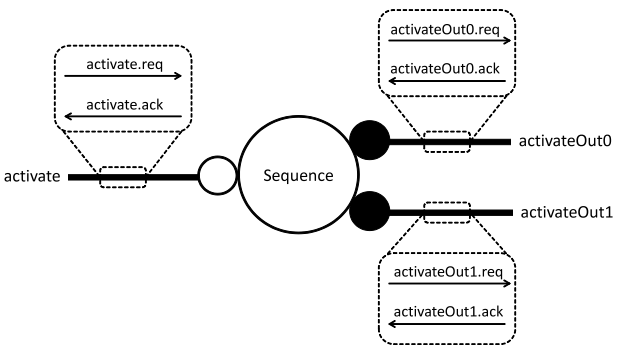


Fig. 26 Sequence component.

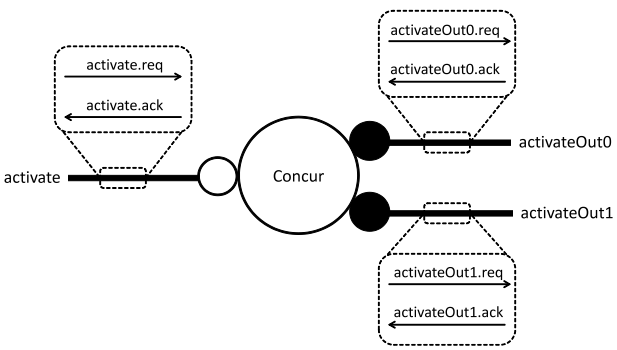
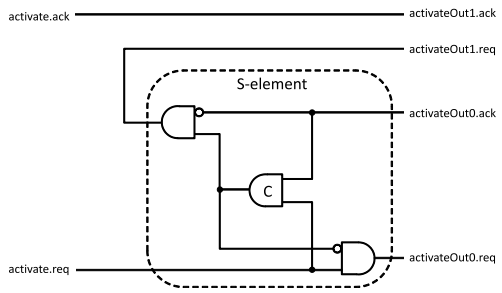
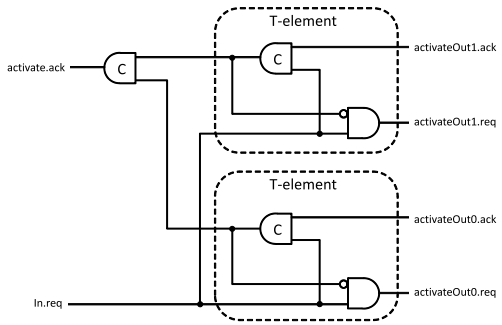


Fig. 27 Concur component.



(a) Sequence component.



(b) Concur component.

Fig. 28 Structure of Sequence component and Concur component.

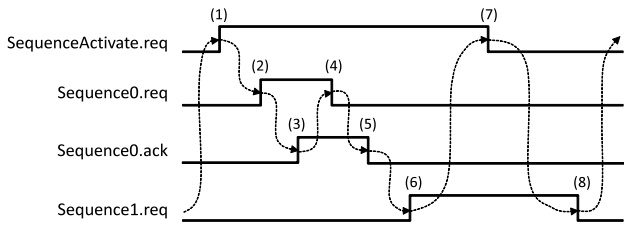


Fig. 29 Behavior of a Sequence module as a Sequence component.

2. *Sequence*0.*req* is set to "1"
3. *Sequence*0.*ack* is set to "1"
4. *Sequence*0.*req* is set to "0"
5. *Sequence*0.*ack* is set to "0"
6. *Sequence*1.*req* is set to "1"
7. *SequenceActivate.req* is set to "0"
8. *Sequence*1.*req* is set to "0"

When a Concur component is implemented, *activate.req*, *activate.ack*, *activateOut*0.*req*, *activateOut*0.*ack*, *activateOut*1.*req* and *activateOut*1.*ack* in Fig. 27 correspond to *SequenceActivate.req*, *FalseVariable.ack*, *Sequence*0.*req*, *sequence*0.*ack*, *Concur*1.*req* and *Concur*1.*ack* in Fig. 23. Figure 30 shows signal transitions of a Sequence module as a Concur component. The behavior as a Concur component is described as follows:

1. *SequenceActivate.req* is set to "1"
2. *Sequence*0.*req* and *Concur*1.*req* are set to "1"
3. - *Sequence*0.*ack* is set to "1" following the rise of *Sequence*0.*req*
   - *Concur*1.*ack* is set to "1" following the rise of *Concur*1.*req*

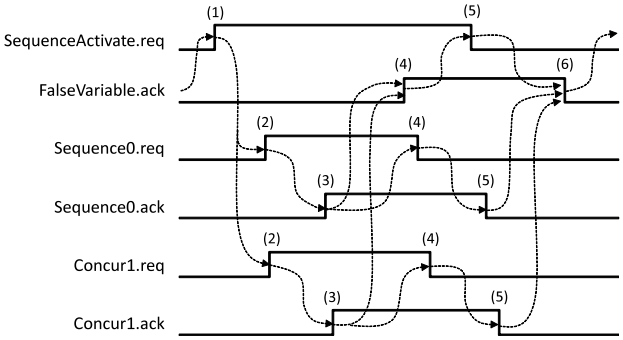


Fig. 30 Behavior of a Sequence module as a Concur component.

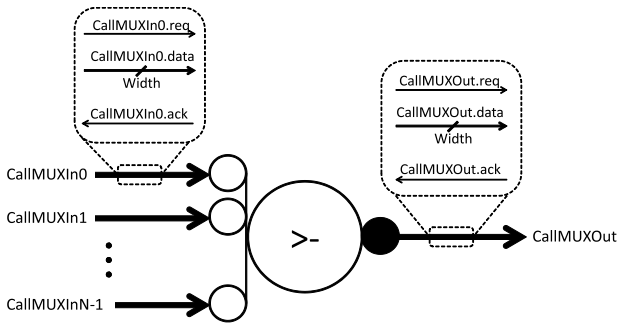


Fig. 31 CallMUX component.

4. - *FalseVariable.ack* is set to "1" following the rise of *Sequence*0.*ack* and *Concur*1.*ack*
   - *Sequence*0.*req* is set to "0" following the rise of *Sequence*0.*ack*
   - *Concur*1.*req* is set to "0" following the rise of *Concur*1.*ack*
5. - *SequenceActivate.req* is set to "0" following the rise of *FalseVariable.ack*
   - *Sequence*0.*ack* is set to "0" following the fall of *Sequence*0.*req*
   - *Concur*1.*ack* is set to "0" following the fall of *Concur*1.*req*
6. *FalseVariable.ack* is set to "0"

A Sequence module is also used to implement Loop component and While component.

3.2.5 CallMUX Module Structure

A CallMUX module implements a CallMUX component shown in Fig. 31. The CallMUX component is used to integrate input channels into a output channel. Figure 32 shows a structure of a CallMUX module. CallMUX module implements four input ports. Every input and output ports can transfer 1-bit data. Figure 33 shows signal transitions of a CallMUX module as a CallMUX component. The behavior when a data arrives at the CallMUXIn0 port is described as follows:

1. A valid data arrives at *CallMUXIn*0 port
2. *CallMUXOut* outputs the value that *CallMUXIn*0 re-

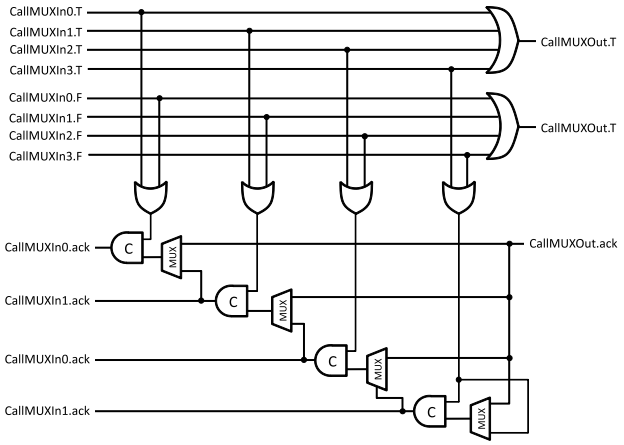


Fig. 32 Structure of a CallMUX module.

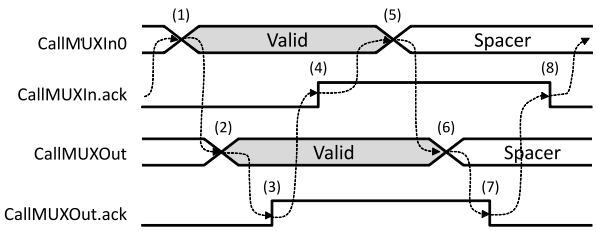


Fig. 33 Behavior of a CalMUX module as a CallMUX component.

ceived
3. *CallMUXOut.ack* is set to "1"
4. *CallMUXIn*0.*ack* is set to "1"
5. A spacer arrives at *CallMUXIn*0 port
6. *CallMUXOut* outputs a spacer
7. *CallMUXOut.ack* is set to "0"
8. *CallMUXIn*0.*ack* is set to "0"

CallMUX module can also implement Call component, Continue component and ContinuePush component.

3.2.6 Case Module Structure

A Case module implements a Case component shown in Fig. 34. Case component selects one of the *CaseOut* ports according to a value that *CaseIn* port received, and starts handshaking. Figure 35 shows a structure of a Case module. A Case module implements four *CaseOut* ports. Figure 36 shows signal transitions of a Case module as a Case component. The behavior when data "0" arrives at the *CaseIn* port is described as follows:

1. Data "0" arrives at *CaseIn* port
2. *CaseOut*0.*req* is set to "1"
3. *CaseOut.ack* is set to "1"
4. *CaseIn.ack* is set to "1"
5. A spacer arrives at *CaseIn* port
6. *CaseOut*0.*req* is set to "0"
7. *CaseOut.ack* is set to "0"
8. *CaseIn.ack* is set to "0"

Case module can also implement CaseDEMUX component,

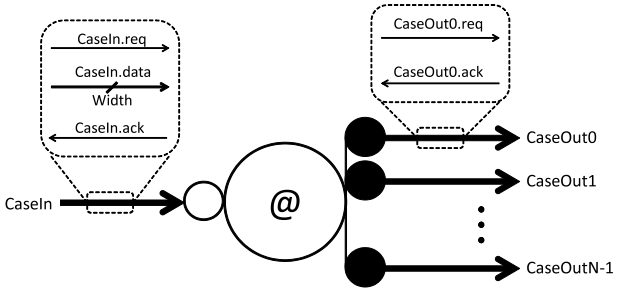


Fig. 34 Case component.

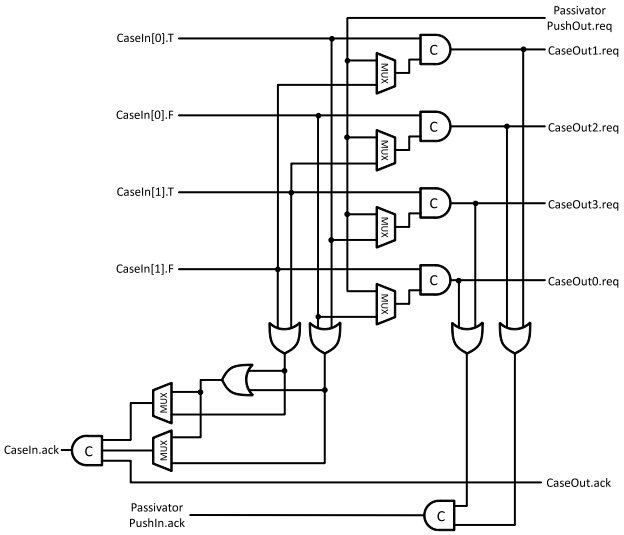


Fig. 35 Structure of a Case module.

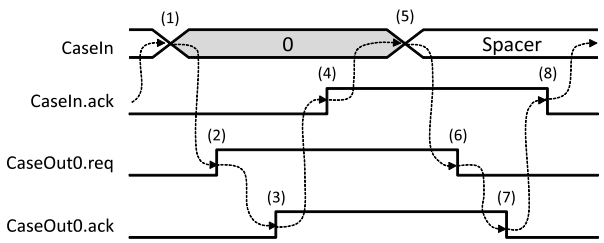


Fig. 36 Behavior of a Case module as a Case component.

CaseFetch component, DecisionWait component, PassivatorPush component and SynchPush component.

3.2.7 Encode Module Structure

An Encode module implements an Encode component shown in Fig. 37. When handshake through $EncodeIn_k$ port starts, *EncodeOut* outputs a data "*k*". Figure 38 shows a structure of an Encode module. An Encode module implements four *EncodeIn* ports. Figure 39 shows signal transitions of an Encode module. The behavior when handshake through *EncodeIn*0 ports starts is described as follows:

1. *EncodeIn*0.*req* is set to "1"
2. *EncodeOut* outputs the data "0"

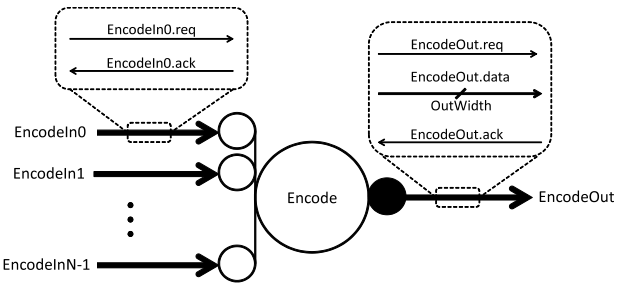


Fig. 37 Encode component.

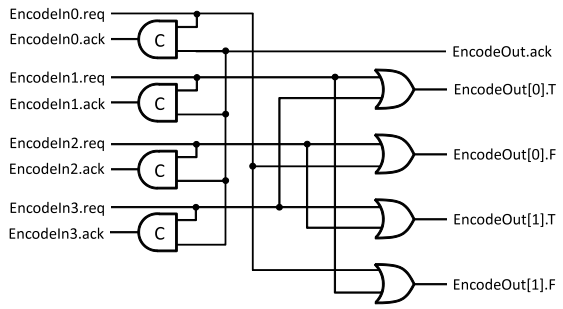


Fig. 38 Structure of an Encode module.

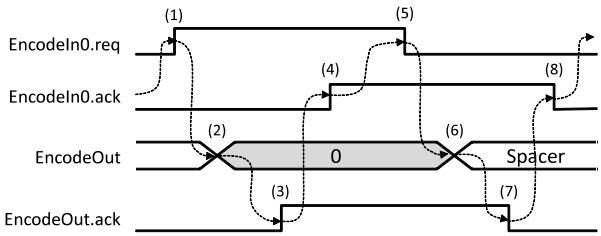


Fig. 39 Behavior of an Encode module.

Table 2 Handshake components and its corresponding resources.

Module	Handshake component
Variable	BuiltinVariable, Variable
Sequence	Concur, Loop, Sequence, While
CallMUX	Call, CallMUX, Continue, ContinuePush
Case	CallDEMUX, Case, CaseFetch, DecisionWait, PassivatorPush, SynchPush
Encode	Encode
BinaryFunction and Variable	BinaryFunc, BinaryFuncConstR, UnaryFunc
Variable and Sequence	FalseVariable, ActiveEagerFalseVariable, PassiveEagerFalseVariable
Programmable Interconnect resources	Adapt, Combine, CombineEqual, Constant, Fetch, Fork, ForkPush, Halt, HaltPush, Passivator, Slice, Split, SplitEqual, Synch, SynchPull, WireFork

3. *EncodeOut.ack* is set to "1"
4. *EncodeIn*0.*ack* is set to "1"
5. *EncodeIn*0.*req* is set to "0"
6. *EncodeOut* outputs spacers
7. *EncodeOut.ack* is set to "0"
8. *EncodeIn*0.*ack* is set to "0"

As shown in Table 2, each module implements several handshake components. Therefore, the number of the transistors of the proposed FPGA is small because of resource sharing.

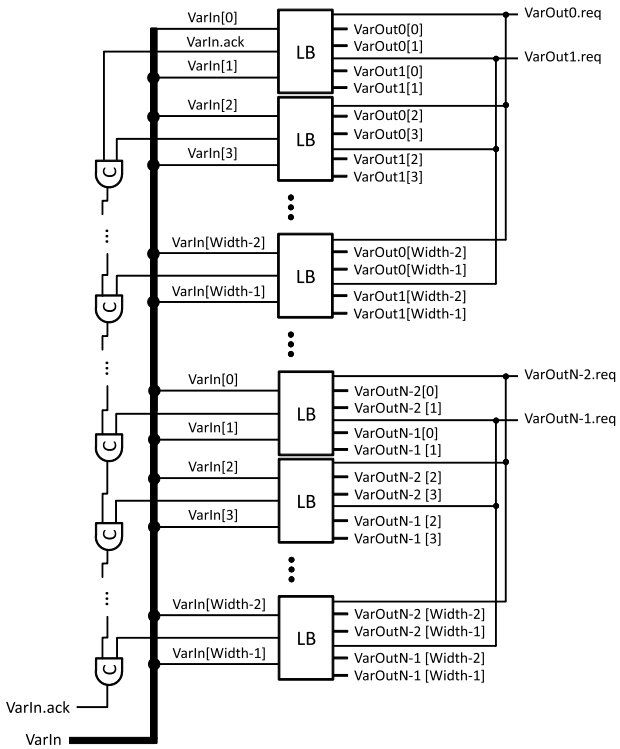


Fig. 40 An implementation of a complex Variable component.

3.3 Implementation of Complex Handshake Components

In the proposed architecture, each LB contains modules to implement handshake components. However, to keep a structure of LB simple, handshake components that can be implemented using an LB is limited. Therefore, in the proposed architecture, frequently-used simple handshake components are implemented using an LB, and rarely-used large-scale handshake components are implemented using multiple LBs and programmable interconnections. As a example of complex handshake components, an implementation of Variable component that stores *Width*-bit data is shown below. In general, Variable component has a passive port that receives a *Width*-bit data and *N* passive ports to output *Width*-bit data as shown in Fig. 17. In the proposed architecture, an LB contains a Variable module that stores 2-bit data. Also, a Variable module has a 2-bit input port and two 2-bit output ports. Therefore,

$$\left\lceil \frac{Width}{2} \right\rceil \times \left\lceil \frac{N}{2} \right\rceil \tag{1}$$

LBs are required to implement a Variable component with *N Width*-bit output ports as shown in Fig. 40.

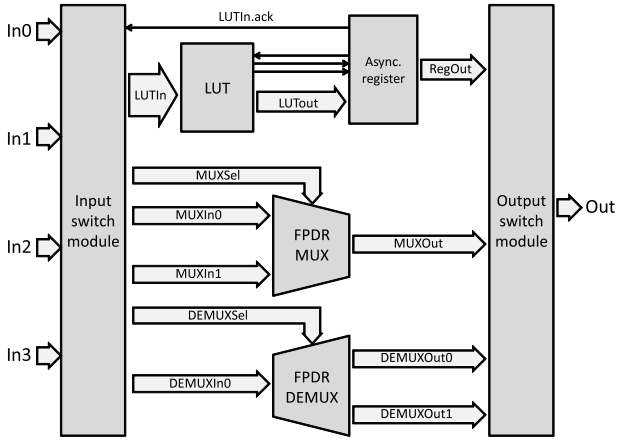


Fig. 41 LB of a conventional architecture.

Table 3 Transistor count of a cell and its breakdown.

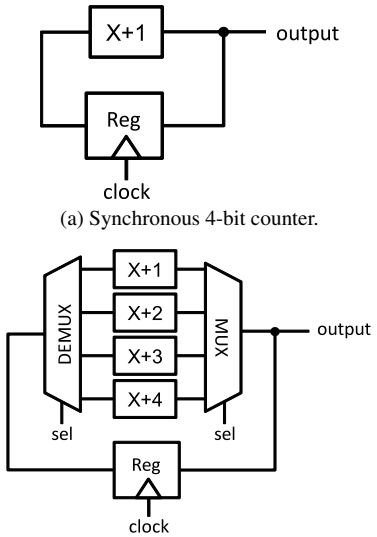
	Conventional architecture	Proposed architecture
Cell	2401	3893
LB	589	1255
SB and CBs	1812	2638

4. Evaluation

The proposed FPGA is implemented in e-Shuttle 65 nm CMOS process with 1.2 V supply. The circuits are evaluated by pre-layout simulation with HSPICE. Therefore, parasitic capacitance and resistance of programmable interconnection resources are not considered in evaluation results. For comparison, The conventional asynchronous FPGA architecture is implemented. Figure 41 shows the LB structure of the conventional FPDR FPGA architecture. The LB of the conventional FPGA mainly consists of an LUT, an asynchronous register, an FPDR multiplexer and an FPDR demultiplexer [14]. In the conventional asynchronous FPGA, applications are designed combining seven building blocks [5].

Table 3 shows the comparison result of the cells of the proposed architecture and the conventional architecture. Since the proposed architecture contains modules for handshake components, the transistor count of a cell is increased by 62%.

The next evaluation shows the implementation results of a 4-bit counter and a 4-bit counter with conditional branch. Figure 42 shows equivalent synchronous circuits of the test applications. Table 4 shows the comparison of cell counts and transistor counts. The benchmark circuits consist of cells and each cell includes an LB, an SB and two CBs as shown in Fig. 10. In the case of 4-bit counter, the number of cells is reduced by 21%. However, the transistor count is increased by 27% compared to the conventional architecture as shown in Table 4 (a). On the other hand, as shown in Fig. 4 (b) the numbers of cells and transistors are reduced by



(a) Synchronous 4-bit counter.

(b) Synchronous 4-bit counter with conditional branch.

Fig. 42 Synchronous circuits equivalent to asynchronous evaluation circuits.

Table 4 Evaluation results of transistor counts.

(a) Results of 4-bit counter.

	Conventional architecture	Proposed architecture
Number of cells	14	11
Number of transistors	33614	42823

(b) Results of 4-bit counter with conditional branch.

	Conventional architecture	Proposed architecture
Number of cells	53	29
Number of transistors	127253	112897

45% and 11% in the case of 4-bit counter with conditional branch. This is because handshake-component-based design can efficiently implement applications that include data path control such as conditional branch.

Table 5 shows the comparison of energy consumptions per operation to count up. Compared to the conventional architecture, the energy consumptions is reduced by 9% and 27% respectively. The results show that the proposed architecture is suitable for applications with complex sequence control.

Table 6 shows the comparison of throughputs. The throughput is defined by the number of operations per second. Compared to the conventional architecture, throughputs are decreased by 51% and 41% respectively. This is because handshake components execute complex handshake sequence.

Table 5 Evaluation results of energy consumptions per operation to count up.

	Conventional architecture	Proposed architecture
4-bit counter [pJ]	5.04	4.57
4-bit counter with conditional branch [pJ]	13.49	9.88

Table 6 Evaluation results of throughputs.

	Conventional architecture	Proposed architecture
4-bit counter [M operations/sec]	162.50	79.06
4-bit counter with conditional branch [M operations/sec]	85.70	50.28

5. Conclusions

This paper presented an architecture of an asynchronous FPGA for handshake-component-based design. The proposed FPGA architecture implements handshake components efficiently. Thus, the proposed architecture is suitable for the synthesis tools that generate netlists consist of handshake components, such as Balsa. In addition, the handshake-component-based design is suitable for applications that have complex data path controls. Therefore, the proposed architecture is suitable to implement complex large-scale asynchronous circuits.

As a future work, hybrid architecture of the conventional asynchronous FPGA and the proposed asynchronous FPGA can be considered. The conventional asynchronous architecture is simple and it can achieve high throughput. On the other hand, the proposed architecture is suitable for applications that have complex data path controls. Therefore, employing the conventional architecture in data path and the proposed architecture in sequence controller, low power, small area and high throughput implementation would be achieved.

Acknowledgment

This work is supported by VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with STARC, e-Shuttle, Inc., Fujitsu Ltd., Cadence Design Systems Inc. and Synopsys Inc. This work is supported by JSPS KAKENHI Grant Number 25·5513.

References

[1] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-programming language Tangram and its translation into handshake circuits," Proc. EDAC, pp.384–389, 1991.
[2] A. Bardsley, "Implementing Balsa Handshake Circuits," 2000.
[3] Q. Zhang and G. Theodoropoulos, "Modelling SAMIPS: A synthesisable asynchronous MIPS processor," Proc. 37th Annual Simulation Symposium, pp.205–212, 2004.
[4] J. Teifel and R. Manohar, "An asynchronous dataflow FPGA architecture," IEEE Trans. Comput., vol.53, no.11, pp.1376–1392, 2004.
[5] R. Manohar, "Reconfigurable asynchronous logic," Proc. IEEE Custom Integrated Circuits Conference, pp.13–20, Sept. 2006.
[6] Achronix Semiconductor Corporation, "Introduction to Achronix FPGAs," Aug. 2008.
[7] B. Devlin, M. Ikeda, and K. Asada, "A 65 nm gate-level pipelined self-synchronous FPGA for high performance and variation robust operation," IEEE J. Solid-State Circuits, vol.46, no.11, pp.2500–2513, Nov. 2011.
[8] M. Hariyama, S. Ishihara, and M. Kameyama, "Evaluation of a field-programmable VLSI based on an asynchronous bit- serial architecture," IEICE Trans. Electron, vol.E91-C, no.9, pp.1419–1426, Sept. 2008.
[9] M. Hariyama, S. Ishihara, and M. Kameyama, "A low-power field-programmable VLSI based on a fine-grained power-gating scheme," Proc. IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Knoxville (USA), pp.430–433, Aug. 2008.
[10] S. Ishihara, Y. Komatsu, M. Hariyama, and M. Kameyama, "An asynchronous field-programmable VLSI using LEDR/4-phase-dual-rail protocol converters," Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas (USA), pp.145–150, July 2009.
[11] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, Logic Synthesis for Asynchronous Controllers and Interfaces, Springer-Verlag, 2002.
[12] T.A. CHU, Synthesis of self-timed vlsi circuits from graph-theoretic specifications, PhD Thesis, MIT Laboratory for Computer Science, 1987.
[13] D. Edwards, A. Bardsley, L. Janin, L. Plana, and W. Toms, "Balsa: A tutorial guide," ftp://ftp.cs.man.ac.uk/pub/apt/balsa/3.5.1/, May 2006.
[14] J. Sparsø and S. Furber, Principles of Asynchronous Circuit Design: A Systems Perspective, Kluwer Academic Publishers, 2001.
[15] S. Hauck, "Asynchronous design methodologies: An overview," Proc. IEEE, vol.83, no.1, pp.69–93, 1995.

Yoshiya Komatsu received the B.E. degree in Information Engineering and M.S. degree in Information Sciences from Tohoku University, Sendai, Japan, in 2009 and 2011, respectively. He is currently working toward the Ph.D. degree in Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing and asynchronous architecture.

Masanori Hariyama received the B.E. degree in electronic engineering, M.S. degree in Information Sciences, and Ph.D. in Information Sciences from Tohoku University, Sendai, Japan, in 1992, 1994, and 1997, respectively. He is currently an associate professor in Graduate School of Information Sciences, Tohoku University. His research interests include VLSI computing for real-world application such as robots, high-level design methodology for VLSIs and reconfigurable computing.

Michitaka Kameyama received the B.E., M.E. and D.E. degrees in Electronic Engineering from Tohoku University, Sendai, Japan, in 1973, 1975, and 1978, respectively. He is currently Dean and a Professor in the Graduate School of Information Sciences, Tohoku University. His general research interests are intelligent integrated systems for real-world applications and robotics, advanced VLSI architecture, and new-concept VLSI including multiple-valued VLSI computing. Dr.Kameyama received the Outstanding Paper Awards at the 1984, 1985, 1987 and 1989 IEEE International Symposiums on Multiple-Valued Logic, the Technically Excellent Award from the Society of Instrument and Control Engineers of Japan in 1986, the Outstanding Transactions Paper Award from the IEICE in 1989, the Technically Excellent Award from the Robotics Society of Japan in 1990, and the Special Award at the 9th LSI Design of the Year in 2002. Dr. Kameyama is an IEEE Fellow.