Register-Bridge Architecture and its Application to Multiprocessor Systems

Takafumi Fujii Shinichi Nishizawa Kazuhito Ito

Graduate School of Science and Engineering
Saitama University
255 Shimookubo, Sakura-ku, Saitama, 338-8570, Japan
kazuhito@ees.saitama-u.ac.jp

Abstract— The interconnection delay in data transfer is becoming the dominant factor to restrain the improvement of the maximum clock frequency of LSIs. The regular distributed register (RDR) architecture is proposed where data transfer between the islands is separated from the computation and local data access, and distant data transfer is done using multiple clock cycles. In this paper a novel register-bridge (RB) architecture is proposed so that data transfer between adjacent islands is done through bridge registers in between the islands, thereby the necessary number of clock cycles for data transfer would be reduced. The experimental results show about 11 % reduction in the latency on average when example procedures are implemented on a multiprocessor system based on the RB architecture.

I. Introduction

As LSI manufacturing technology evolves, the wiring delay is relatively more increased than the logic delay and becoming the dominant factor to restrain the improvement of the maximum applicable clock frequency of LSIs [1]. To overcome the issue, the regular distributed register (RDR) architecture [1] is proposed, which divides the LSI chip area into regular sized islands and distributes data registers as well as functional units (FUs) and controllers to the islands. FUs read data from the registers which are local to the island (LREGs) and write the results also to the LREGs within a clock cycle. For the data transfer from an island to another, dedicated clock cycles are allocated. The more distant a data transfer is, the more clock cycles are allocated for the data transfer. Hence the clock period is minimized despite the relative increase of wiring delay. Many network-on-chips (NoC) have been designed and the distributed register architecture is assumed [2, 3, 4].

Since the RDR architecture constrains all the islands to be of equal size, dead space may exist in islands with a small number of FUs and registers. The huddle-based distributed register (HDR) architecture [5, 6] is proposed, where non-regular islands are allowed to improve the chip area usage by eliminating dead space. The HDR architecture also assumes that the registers are distributed to islands and only local registers can be accessed within a clock period.

The drawback of the RDR and HDR architectures is that one clock cycle is necessary even for the data transfer between adjacent islands. In [4] the routers are designed so that distant islands can be reached within a clock period, however the above mentioned drawback still exists. In this paper a novel register-bridge (RB) architecture is proposed. In the RB architecture, in addition to LREGs in each island, bridge registers (BREGs) are placed in between adjacent islands. For an island, its adjacent BREGs are placed on its boundary and thus the wiring delay for the data transfer between FUs and BREGs is not largely different from the one between FUs and LREGs. Hence the clock period would remain as small as the case of the RDR architecture. Since BREGs are accessed by their adjacent islands similarly to the LREGs, transferring a data between two adjacent islands is achieved by simply writing the data to a BREG by one island and reading it from the same BREG by the other island. No dedicated clock cycle is necessary for the data transfer between adjacent islands, and the required number of clock cycles would be reduced and faster processing might be achieved.

This paper is organized as follows. The RB architecture is proposed in section II. The application of the RB architecture to multiprocessor systems is considered in section III. The processor extension necessary to implement the RB architecture is also presented there. The experimental results are shown in section IV, and section V concludes the work.

II. Register-bridge architecture

Figure 1 shows an example of the RDR architecture consisting of 4 islands. The islands are connected with their adjacent islands through data wires. Each island has LREGs, FU(s), and a controller, e.g. a finite state machine (FSM). Data are read from one or more LREGs and used by a computation in an FU, and the computation result is stored in an LREG in the identical island. That is, data are read from LREGs, the data are transferred to an FU, the FU executes a computation using the data, and the computation result is transferred to an LREG and stored in the LREG. All these operations are performed in one clock cycle (CC).

In the RDR architecture, data is transferred between two islands as follows. One island reads a data from its local LREG and outputs the data to the other island, and the other island inputs the data and stores it into its local LREG. The transferred data travels from the LREG in the source island to the LREG in the destination island. In the RDR architecture, it is assumed that it takes one CC for the data to travel between adjacent islands. For example, suppose the island i1 executes a

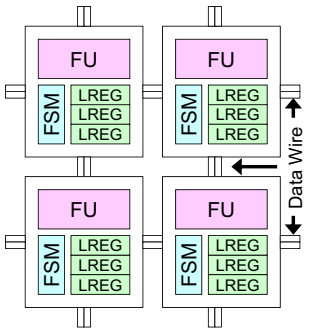


Fig. 1. Regular distributed register architecture.

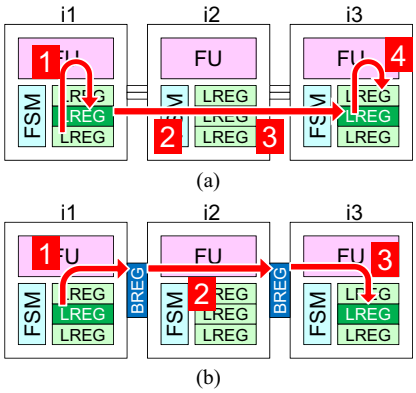


Fig. 2. Computations and data transfer. (a) In RDR architecture. (b) In RB architecture.

computation using its local data and the result is then computed in the island i3 as shown in Fig. 2(a). In the first CC (CC1), i1 executes a computation and the result is stored in an LREG in i1. The result is read from the LREG in i1 and transferred to i3 and stored in an LREG in i3 using CC2 and CC3. Finally, i3 executes a computation using the data stored in the LREG in CC4. Consequently, the operation takes 4 CCs.

Here an RB architecture is proposed to reduce CCs consumed for data transfer. As shown in Fig. 3, BREGs are placed between adjacent islands instead of data wire. The BREGs can be read or written from both their adjacent islands. For an island, its adjacent BREGs are placed on its boundaries but still inside the island. Hence the wiring distance between FUs and BREGs is not largely different from the distance between FUs and LREGs. An island reads data from its LREGs or BREGs, execute a computation in an FU, and store the result to an LREG or a BREG within a CC, while the clock period is almost identical to or slightly longer than the clock period in the case of the RDR architecture.

The same operation illustrated in Fig. 2(a) for the RDR architecture is performed as shown in Fig. 2(b) for the RB architecture. In CC1, the island i1 executes a computation and the result is stored in an BREG between i1 and i2. Then i2 reads the result from the BREG and stores it in a BREG between i2 and i3 in CC2. Finally, i3 executes a computation using the data stored in the BREG in CC3. Consequently, the operation takes only 3 CCs.

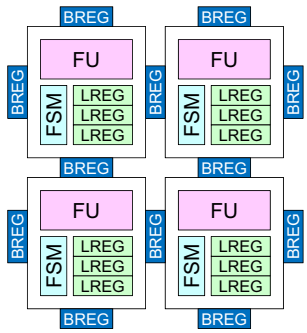


Fig. 3. Register-bridge architecture.

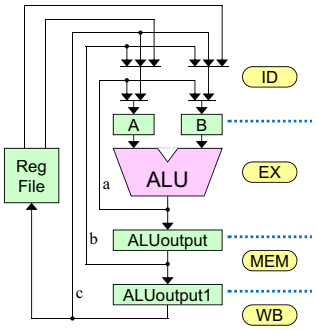


Fig. 4. The datapath of DLX processor.

III. Multiprocessor Based on RB Architecture

The application of the RB architecture for constructing multiprocessor systems is presented.

A. Base Processor

The DLX processor [7] is used as the base processor. It is briefly reviewed. The datapath of the DLX is shown in Fig. 4. The ALU is an arithmetic-logic unit and the registers A and B provide the source data to the ALU. The register ALUoutput stores the result of the ALU. There are 32 integer general purpose registers (GPR) $0 to $31 and these GPRs form a register file (RF). $0 is special and always reads 0. An instruction requires at most two source data from GPR(s) and may write the result into a destination GPR. Hence the RF is capable of reading data from two GPRs and writing data into a GPR at the same time. The original DLX has the floating point part and the integer multiplier and divider, but those are omitted in Fig. 4. The program counter, the instruction fetch unit, the instruction decoder, memory address and data registers for load and store instructions exist, but are omitted for simplicity.

The execution of instructions is pipelined in 5 stages of instruction fetch (IF), instruction decode (ID), execution (EX), memory access (MEM), and write back of the result (WB). In the IF stage, an instruction is loaded from the program memory. The ID stage decodes the instruction and necessary source data are read from the RF and stored in A and B registers. Then the ALU executes a computation and the result is stored in the register ALUoutput in the EX stage. For memory load and store instructions, the memory address is calculated using the

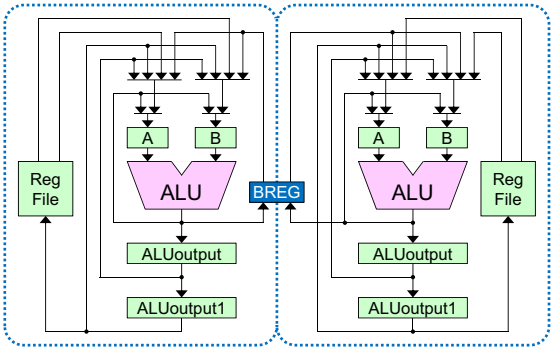


Fig. 5. Bridge-register architecture.

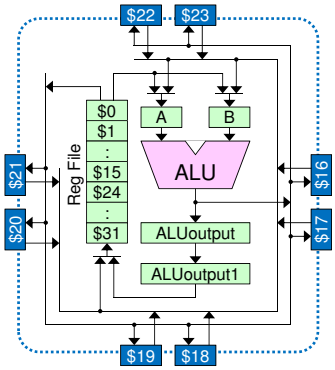


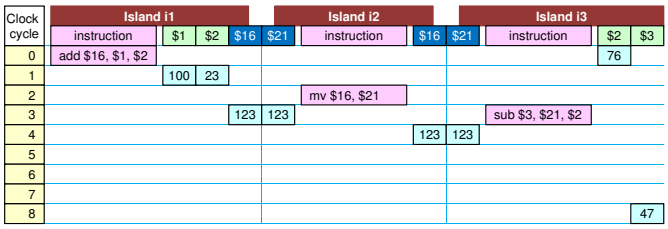
Fig. 6. The extended processor for RB architecture.

ALU in the EX stage. In the MEM stage, the data memory is accessed in the case of load and store instructions. For other instructions, the MEM stage does nothing and the content of ALUoutput is transferred to another register ALUoutput1. Finally the result of the instruction is written back to a GPR in the WB stage.

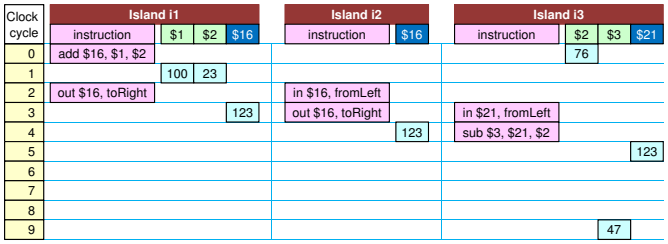
In a normal execution of an instruction, it takes 5 CCs after an instruction is initiated to store the result of the instruction in a GPR in RF. In order to resolve read after write (RAW) hazard in the pipelined execution of instructions, forwarding [7] is implemented. Since the source data of the ALU are required in the ID stage, the forwarding is performed from the EX stage of the previous instruction (marked as 'a' in Fig. 4), from the MEM stage of the 2nd previous instruction (b), and from the WB stage of the 3rd previous instruction (c).

B. Processor extension for RB architecture

Figure 5 shows an example of a processor array based on the RB architecture, where two processors are connected with each other through a BREG. The detail of the processor extended for the RB architecture is shown in Fig. 6, where the forwarding paths are omitted for readability. It is assumed that two BREGs are allocated to each of the 4 edges of a processor, thus 8 BREGs exist in total. The 8 BREGs are mapped to GPRs $16 to $23 so that the change in the instruction formats is not necessary. In other words, access to GPRs $16 to $23 is translated to the access to the corresponding BREGs. Note



(a)



(b)

Fig. 7. An example operations. (a) In the RB architecture. (b) In the RDR architecture.

that since BREGs are shared by adjacent two processors, the BREGs $16 and $17 of a processor are identical to the BREGs $21 and $20, respectively, of the processor to the right. The BREG can be written and read by either of the two processors. One processor writes a data into the BREG $16 and the processor to the right reads the data from the BREG $21, thereby a data is transferred between adjacent two processors. Simultaneous writing to an identical BREG by adjacent processors is avoided by instruction scheduling.

The BREG is designed to receive a data from the output of the ALU. Hence the result of an instruction is stored in the BREG at the end of the EX stage. While a GPR in RF requires 5 CCs to receive a new data by executing an instruction, the BREG requires only 3 CCs and it contributes to reduce the latency in data communication between processors.

For an efficient data transfer from one BREG to another or between a BREG and a GPR within a processor, the instruction format is extended to order a 'move' operation. In addition to the original opcode field, source and destination register index fields, etc., in an instruction, two more fields are added to specify the source and the destination of the move operation. Now a single instruction can execute at most two operations; one is using the ALU and the other is the move. Not executing any move is indicated by setting the destination to $0. Once an instruction is fetched, the move is performed in the ID stage.

Figure 7(a) shows an example of three processors of the RB architecture executing the similar operations as shown in Fig. 2(b). The processors (islands) i1 and i2, and i2 and i3 are respectively connected through BREGs. At CC 0, the processor i1 fetches the instruction 'add $16, $1, $2,' which adds the data in the registers $1 and $2 and the sum is stored in the register $16. The sum is computed in the EX stage of the instruction (CC 2). Since $16 is a BREG, $16 receives the sum (= 123) directly from the ALU and the sum is available in $16 at CC 3. Then the data is read by the processor i2 from the BREG $21

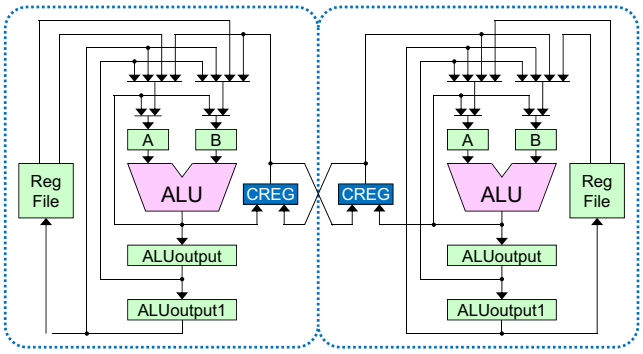


Fig. 8. Regular distributed register architecture.

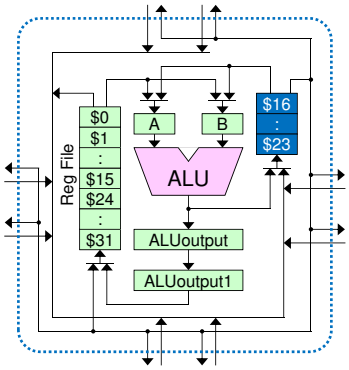


Fig. 9. The extended processor for RDR architecture.

and moved to the BREG \$16 in CC 3. To implement this move, i2 fetches the instruction 'mv \$16, \$21' at CC 2 and the data in \$21 is moved to \$16 in the ID stage at CC 3. In this case, the instruction only executes the move operation and the operation using the ALU is not executed. Finally, at CC 3, the processor i3 fetches the instruction 'sub \$3, \$21, \$2,' which subtracts \$2 from \$21 and stores the result to \$3. \$3 is a normal GPR and it receives the result in the WB stage at CC 7.

C. *Processor extension for RDR architecture*

For comparison, the processor array based on the RDR architecture is considered as shown in Fig. 8. The detail of the processor extended for the RDR architecture is shown in Fig. 9. There is one or more communication registers (CREG) in the processor. It is assumed in Fig. 9 that 8 CREGs are allocated and mapped to GPRs \$16 to \$23. In the RDR architecture, reading a data from a register in one island, transferring it to an adjacent island, and storing it in a register are performed in a single CC. Therefore, to transfer a data from one processor to another adjacent processor, the former processor stores the data in one of its CREGs, and then the data is copied to one of the CREGs in the latter processor.

Similar to the BREG in the RB architecture, the CREG receives a data from the ALU at the end of the EX stage.

To implement sending a data in a CREG to an adjacent processor and storing a data from an adjacent processor into a CREG, the instruction format is extended to add fields which

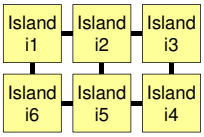


Fig. 10. A processor array.

```
 1: delta = 0;
 2: if( diff >= step ){
 3:     delta = 4;
 4:     diff -= step;
 5:     vpdiff += step;
 6: }
 7: step >>= 1;
 8: if( diff >= step ){
 9:     delta |= 2;
10:     diff -= step;
11:     vpdiff += step;
12: }
13: step >>= 1;
14: if( diff >= step ){
15:     delta |= 1;
16:     vpdiff += step;
17: }
```

Fig. 11. A part of the C source of the procedure 'coder' in ADPCM [8].

order an 'out' operation to send a data to an adjacent processor or an 'in' operation to receive a data from an adjacent processor. In addition, a 'move' operation is also supported to exchange a data between a CREG and a GPR.

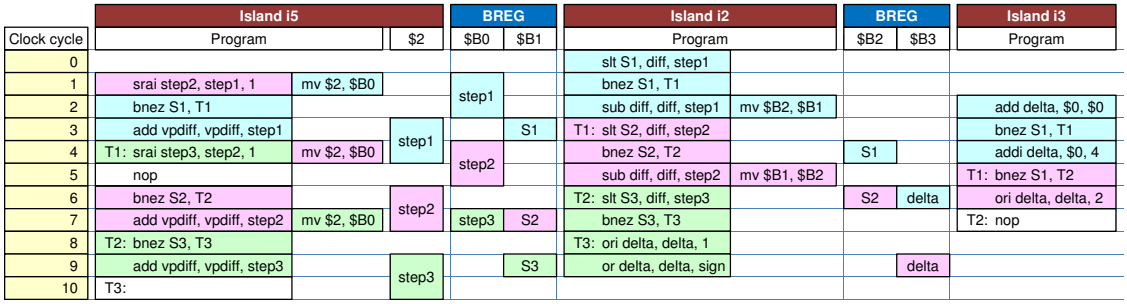
Figure 7(b) shows an example of three processors of the RDR architecture executing the similar operations as shown in Fig. 2(a). The processors (islands) i1 and i2, and i2 and i3 are respectively connected through data wire. At CC 0, the processor i1 fetches the instruction 'add \$16, \$1, \$2.' Since \$16 is a CREG, the result (the sum = 123) is available in \$16 at CC 3. The processor i1 also fetches 'out \$16, toRight' at CC 2, which sends the data in \$16 to the processor i2 at CC 3. The processor i2 fetches 'in \$16, fromLeft' at CC 2, which receives the data from i1 and stores it in the CREG \$16 in i2 at CC 3, and the data is available in \$16 at CC 4. i2 also fetches 'out \$16, toRight' at CC 3, which sends the data in \$16 to the processor i3 at CC 4. At CC 3, the processor i3 fetches 'in \$21, fromLeft,' which stores the received data in \$21 at CC 4. Finally, at CC 4, the processor i3 fetches the instruction 'sub \$3, \$21, \$2,' and the result is stored in \$3 in the WB stage at CC 8.

As can be seen from the example, the processor array of the RB architecture executes the operations one CC shorter than that of the RDR architecture.

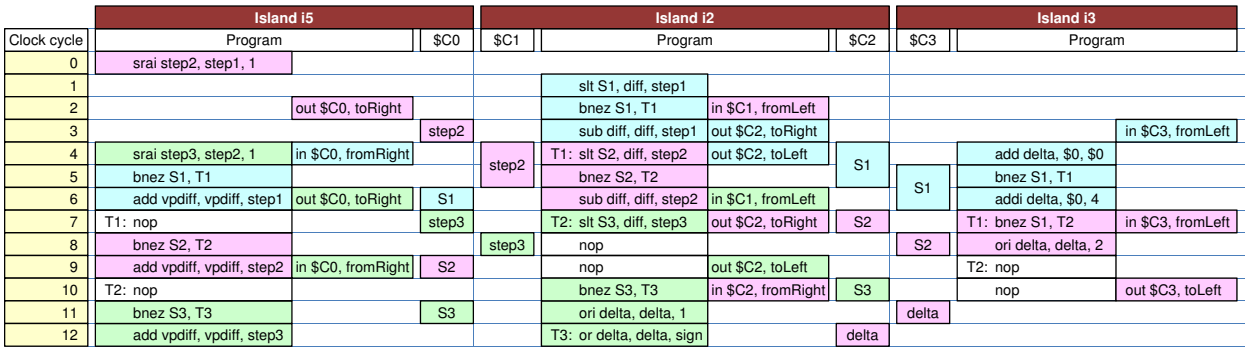
IV. Experimental Results

To show the effectiveness of the proposed RB architecture, practical procedures were mapped onto the processor array of the RB architecture. The procedures 'coder' and 'decoder' in ADPCM in MediaBench [8] were selected as examples and the processor array size was assumed to be 2×3. The processors are named 'Island i1' to 'Island i6' as shown in Fig. 10.

Figure 11 shows a part of the C source of the procedure



(a)



(b)

Fig. 12. Optimized compilation results for the C source shown in Fig. 11. (a) RB architecture. (b) RDR architecture.

TABLE I
INSTRUCTIONS OF THE PROCESSORS

instruction	operation
add *dst*, *src1*, *src2*	$dst \leftarrow src1 + src2$
addi *dst*, *src1*, *im*	$dst \leftarrow src1 + im$
bnez *src*, *label*	branch to *label* if $src \neq 0$
nop	do nothing
or *dst*, *src1*, *src2*	$dst \leftarrow src1 \mid src2$
ori *dst*, *src1*, *im*	$dst \leftarrow src1 \mid im$
slt *dst*, *src1*, *src2*	$dst \leftarrow 1$ if $src1 < src2$, 0 otherwise
srai *dst*, *src1*, *im*	$dst \leftarrow src1 >> im$
sub *dst*, *src1*, *src2*	$dst \leftarrow src1 - src2$
mv *dst*, *src*	$dst \leftarrow src$
in *dst*, *src*	receive a data from *src* and store in *dst*
out *src*, *dst*	read a data from *src* and send to *dst*

'coder.' The results of the optimized compilation, which consists of the scheduling and mapping of instructions, of the C source are shown in Fig. 12 for both RB and RDR architectures. Figure 12 only shows the islands which execute the instructions compiled from the C source shown in Fig. 11. The instructions employed in the results are described in Table I. In Fig. 12, some of the source and destination of instructions are indicated not with the register index but with the variable name in the C source and the register assignment is omitted for readability. Note that the source data of the instructions is accessed in the ID stage of the execution, which is one CC after the instruction is started.

In the case of the RB architecture (Fig. 12(a)), the variable *delta* in the C source is maintained in the processor i3, *diff* in i2, and *step* and *vpdiff* in i5. The line 1 of the C source (denoted as L1) is implemented by the add instruction in i3 at CC 2. The comparison in L2 is implemented by the slt instruction in i2 at CC 0. The slt instruction reads the value of *step*, denoted as 'step1,' from the BREG $B0 between i2 and i5 at CC 1. The bnez instruction at CC 1 skips the next sub instruction if the condition $diff \geq step$ is not true. The result of the slt instruction at CC 0, denoted as 'S1,' is stored in BREG $B1 and available at CC 3. It is referenced by the bnez instruction started at CC 2 in i5 and controls the execution of the add instruction at CC 3 compiled from L5. S1 is moved from BREG $B1 to BREG $B2 by the mv instruction at CC 2 in i2. It is important to note that the mv instruction follows the bnez instruction at CC 1. When the branch is taken in the bnez instruction, the IF and ID stages of the following instruction are always performed in the pipelined instruction execution. Therefore the move is always performed regardless of whether the branch is taken or not taken. S1 is transferred to $B2 and used by the bnez instruction started at CC 3 in i3 to control the execution of the addi instruction at CC 4 compiled from L3. While the comparison result S1 is first stored in $B1 and then moved to $B2, S2 is first stored in $B2 and then moved to $B1. This is to perform the update of *delta* compiled from L9 as early as possible.

The value of step is changed in L7 and it is implemented by the srai instruction at CC 1 in i5. The new value, denoted as 'step2,' is stored in $B0 at CC 4. The previous value 'step1' is referenced at CC 3 in i5 by the add instruction. The add instruction cannot read 'step1' from $B0 because the updated 'step2' is passed from the srai instruction by forwarding. To suppress the forwarding, the value 'step1' is moved to GPR

TABLE II
COMPARISON OF CLOCK CYCLES AND LATENCY

	clock cycles		latency [ns]	
procedure	RDR arch.	RB arch.	RDR arch.	RB arch.
coder	37	33 (−10.8 %)	250	228 (−8.97 %)
decoder	34	29 (−14.7 %)	230	200 (−12.9 %)
average		−12.7 %		−10.9 %

$2 by the mv instruction at CC 1 and the add instruction at CC 3 reads 'step1' from $2. Consequently, the RB architecture implements the procedure for 10 CCs as shown in Fig. 12(a).

In the case of the RDR architecture (Fig. 12(b)), the variable *delta* in the C source is maintained in the processor i3, *diff* and *vpdiff* in i2, and *step* in i5. The comparison in L2 is implemented by the slt instruction in i2 at CC 1. The value of *step*, denoted as 'step1,' has been transferred from i5 and exists in CREG $C1 in i2 at CC 2. The result of the slt instruction at CC 1, denoted as 'S1,' is stored in CREG $C2 in CC 4 and transferred to CREG $C3 in i3 at CC 5. Hence the bnez instruction in i3 is started at CC 3 and controls the execution of the add instruction at CC 5 compiled from L5. As shown in Fig. 12(b), 13 CCs are used to implement the procedure. Since the srai instruction in i5 at CC 0 can be started earlier, the required CCs is reduced to 12 for the RDR architecture.

Table II compares the CCs of the procedures 'coder' and 'decoder' complied for the conventional RDR and the proposed RB architectures. The RB architecture reduces the required CCs for the coder and the decoder by 10.8 % and 14.7 %, respectively.

The proposed RB architecture and the conventional RDR architecture were implemented in VHDL. The word length of the processor is 32 bits and the processor has an ALU but no integer multiplier, divider, nor floating-point units. The length of the baseline DLX instruction is 32 bits. The mv, out, and in instructions were encoded with 10 bits, hence the total length of the instruction word is 42 bits. The Harvard architecture was adopted for the memory, and the capacity of the instruction memory and the data memory were 42×1024 bits and 32×1024, respectively.

The HDL description was synthesized targeting a commercial FPGA device xc7k70tfbv676-1 (Kintex-7) from Xilinx Inc. [9]. The used FPGA design tool was Vivado 2015.2 WebPACK [9] and was run with the maximum clock period constraint of 7 ns. Table III shows the maximum applicable clock frequency (Fmax), and logic resource usage in the number of slices (SLICE), and the memory resource usage in the number of memory blocks (BRAM). 'BRAM' is a 36×1024-bit memory block embedded in the FPGA device.

The results show that Fmax of the proposed BR architecture is 2.03 % slower than the conventional RDR architecture. This is because more multiplexors are needed at the input of the BREGs to select the data from their two adjacent processors.

The total latency of an operation is evaluated by the product of the number of clock cycles and the clock period. The number of clock cycles of the RB architecture is reduced by 12.7 % on average, and the clock period is increased by 2.03 % from those of the RDR architecture. Consequently, the latency of the proposed RB architecture is reduced by 10.9 % on average compared to the conventional RDR architecture as shown in Table II at the expense of 19.2 % increase in logic usage.

TABLE III
SYNTHESIS RESULTS

architecture	Fmax [MHz]	#SLICE	#BRAM
RDR arch.	148.0	15451	22.5
RB arch.	145.0 (−2.03%)	18414 (+19.2%)	22.5

V. CONCLUSIONS

In this paper a register-bridge architecture is proposed to achieve faster processing than the conventional RDR architecture by reducing the clock cycles for data transfer between islands. The proposed RB architecture was applied to multiprocessor systems and the latency is reduced by about 11 % on average for example procedures from the RDR architecture when the designed multiprocessor systems were synthesized targeting FPGA.

The analysis of the cause of increase in logic usage, and the automation of optimizing the scheduling and mapping remain as future work.

REFERENCES

[1] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang, "Architecture and synthesis for on-chip multicycle communication," *IEEE Trans. Computer-Aided Design Integrated Cirtuit. Syst.*, vol. 23, no. 4, pp. 550–564, Apr. 2004.

[2] G. Chen, F. Li, and M. Kandemir, "Compiler-directed channel allocation for saving power in on-chip networks," in *Proc. of 33rd ACM SIGPLAN-SIGACT Symp. POPL '06*, 2006, pp. 194–205.

[3] S. Park, T. Krishna, C.-H. O. Chen, B. Daya, A. P. Chandrakasan, and L.-S. Peh, "Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45nm SOI," in *Proc. DAC '12*, 2012, pp. 398–405.

[4] T. Krishna, C.-H. O. Chen, W.-C. Kwon, and L.-S. Peh, "Smart: Single-cycle multihop traversals over a shared network on chip," *IEEE Micro*, vol. 34, no. 3, pp. 43–56, 2014.

[5] S. Abe, M. Yanagisawa, and N. Togawa, "An energy-efficient high-level synthesis algorithm for huddle-based distributed-register architectures," in *Proc. ISCAS*, 2012, pp. 576–579.

[6] H. Akasaka, S. Abe, M. Yanagisawa, and N. Togawa, "Energy-efficient high-level synthesis for HDR architectures with clock gating based on concurrency-oriented scheduling," *IPSJ Trans. System LSI Design Methodology*, vol. 6, pp. 101–111, 2013.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[8] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proc. 30th Annual ACM/IEEE Int. Symp. on Microarch.*, 1997, pp. 330–335.

[9] Xilinx Inc., http://www.xilinx.com/.