High-level Synthesis Method Using Semi-programmable Hardware for C Program with Memory Access

Akira Yamawaki and Masahiko Iwane

Abstract—**The SPHW (semi-programmable hardware: IWANE Architecture) is a design-level hardware architecture residing on the path of which the C program with memory access is converted to hardware. By using the SPHW, the memory access controller and buffer can be implemented by writing the software program and parameters respectively in contrast to the conventional hardware design method. Compared with the cases that use only HDL (Hardware Description Language), the SPHW which can design the efficient memory controllers at C-level abstraction reduces the development time significantly. In addition, the SPHW shows the comparable performance compared with the HDL hardware including the custom memory access controller even if it is written at high-abstraction in the SPHW. In general, the HLS (high-level synthesis) tool converting the C program to the hardware is often used to reduce the burden designing the data processing hardware. However, the SPHW has not been introduced into any HLS technology yet. This paper develops the true C level-design environment including the SPHW as the data processing hardware on a real commercial HLS tool, Handel-C. By using the SPHW providing the register-based data interface to the data processing hardware, we demonstrate that the HLS tool can easily write the hardware accessing to the memory in C language. This is because the interface provided by the SPHW to the data processing hardware hides the detail of the memory devices and the memory access patterns by the simple stream data. Since most HLS tools assume the data interface as the stream data, the SPHW can be easily combined with the Handel-C. In order to extract the performance of the hardware maximally, hiding memory access latency is very important. On the SPHW, the simple software-pipelining can be applyed to the memory access program and the parameters of the buffer to hide the memory access latency by overlapping the data processing with the memory access. Consequently, the designer can realize the data processing hardware with an efficient data-prefetching mechanism at the complete C-level design entry.**

Index Terms—**high-level synthesis, C program, hardware design, memory latency hiding, system-on-chip, hardware architecture**

I. Introduction

FOR the design of the system-on-chips, the high-level synthesis (HLS) technologies generating the hardware from C program have been researched and developed [1]–[7]. The HLS tool can reduce the design burden significantly due to the high design abstraction. Generally, the HLS technologies are good at generating a data processing hardware assuming simple and typical data access patterns like the stream data. This is because an algorithmic way to automatically convert the C program to the hardware can handle only trivial and statically expectable patterns. For example, some compilers of the HLS support only the dedicated memory access pattern [4], [6], [7]. However, across the users, the application programs and the buffering methods, the memory access patterns are commonly different. Thus, the memory accesses are hard to be treated systematically by an algorithmic way.

In order to extract the performance of the hardware maximally, hiding memory access latency is very important. However, the conventional HLS tools [1]–[7] cannot implicitly hide memory access latency by data prefetching [8]. To hide memory latency, the hardware has to be written skillfully in the C description with the deep knowledge of the used HLS tool and target device. As a result, the HLS tool might generate the hardware including an efficient memory access controller. Even if the HLS tool is used, such burden may be comparable to designing a custom memory access controller from scratch in a hardware description language (HDL).

To tackle the problems mentioned above, we have proposed a design-level hardware architecture, SPHW (semi-programmable hardware: IWANE Architecture) which is inserted onto the path converting the C program with memory access to the hardware [9]. The SPHW implements the memory access controller and the buffer by writing the software program and parameters respectively in contrast to the conventional hardware design method. Compared with the design cases that use the HDL (Hardware Description Language), the SPHW which can design the efficient memory access controllers at C-level abstraction reduces the development time significantly [9]. In addition, the SPHW shows the comparable performance compared with the HDL hardware including the custom memory access controller even if it is written at high-level abstraction [9].

However, the SPHW has not been introduced into any HLS technology yet. This paper attempts to realize the complete C level-design environment including the SPHW on a real commercial HLS tool, Handel-C [10]. By using the SPHW providing the register-based data accessing interface, we demonstrate that the HLS tool can easily write the hardware accessing to the memory in C. This is because this interface hides the detail of the memory devices and the memory access patterns, by providing the data processing hardware with the simple stream data. By introducing the SPHW into the HLS tool, the simple software-pipelining with double buffering to hide memory access latency is easily

Manuscript received January 26, 2011. This research was partially supported by the Kayamori Foundation of Informational Science Advancement and the Grant-in-Aid for Young Scientists (B) (22700055).

A. Yamawaki is with Faculty of Engineering, Kyushu Institute of Technology, 1-1, Sensui, Tobata, Kitakyushu, Fukuoka, 804-8550, Japan, e-mail: yama@ecs.kyutech.ac.jp.

M. Iwane is with Yuundo Co.,Ltd., 5, Higasibori, Shikama, Himeji, 672-8054, Japan.

(Advance online publication: 10 February 2011)

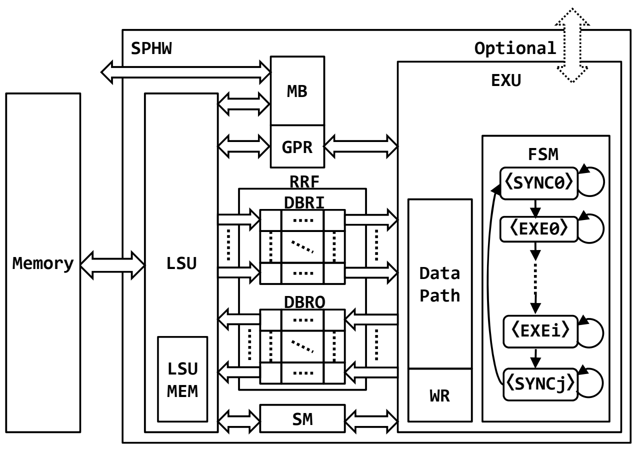


Fig. 1. Block Diagram of SPHW

```
Load line
 LLS(B/Be/BE/E/Eb/EB, mem addr, str width, num of trans, sync)
Load word
 LWS(B/BE/E/EB,       mem addr, str width, num of trans, sync)
Store line
 SLS(B/Be/BE/E/Eb/EB, mem addr, str width, num of trans, sync)
Store word
 SWS(B/BE/E/EB,       mem addr, str width, num of trans, sync)

  B : Increment the bank  pointer per each word transfer.
  Be: Increment the bank  pointer per each word transfer.
      Increment the entry pointer per each line transfer.
  BE: Increment the bank  pointer per each word transfer.
      Increment the entry pointer when all of transfers finish.
  E : Increment the entry pointer per each word transfer.
  Eb: Increment the entry pointer per each word transfer.
      Increment the entry pointer per each line transfer.
  EB: Increment the bank  pointer per each word transfer.
      Increment the entry pointer when all of transfers finish.
```

Fig. 2. Load/store Instructions of LSU

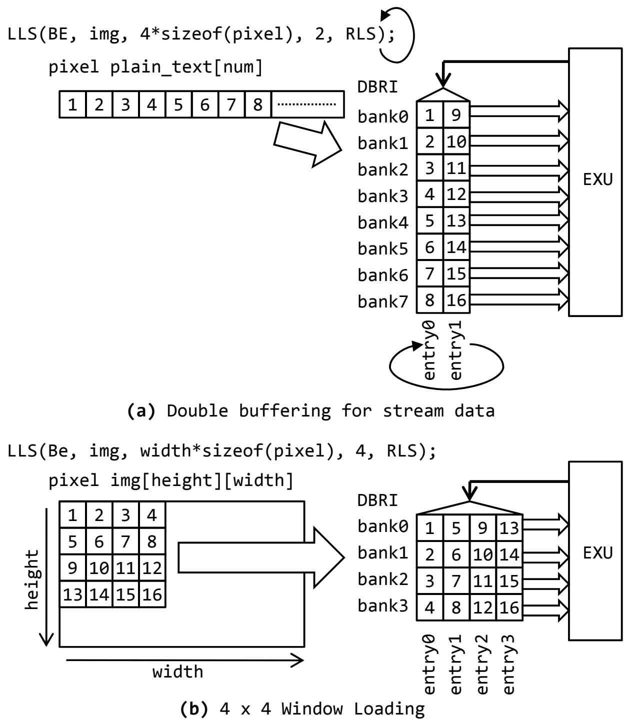

```
LLS(BE, img, 4*sizeof(pixel), 2, RLS);
pixel plain_text[num]
```
```
LLS(Be, img, width*sizeof(pixel), 4, RLS);
pixel img[height][width]
```

Fig. 3. Example of Load Instruction

implemented for the data processing hardware at high-level abstraction.

The rest of the paper is organized as follows. Section 2 describes the overview of the SPHW. Section 3 shows the design flow using the SPHW. Section 4 demonstrates the SPHW mapping by using the examples of the color conversion from RGB to YCrCb. Section 5 performs some preliminary experiments and shows the experimental results. Finally, Section 6 concludes the paper.

II. SPHW Architecture

A. Organization

Fig. 1 shows the organization of the SPHW. The load/store unit (LSU) transfers the data between memory and reconfigurable register file (RRF). The programs to be executed by the LSU are stored in the LSU memory (LSUMEM). The execution unit (EXU) is data processing hardware. The synchronization mechanism (SM) performs the producer–consumer synchronization between the LSU and the EXU. The producer performs the release synchronization to invoke the consumer waiting for the data on the RRF by the wait synchronization.

The reconfigurable register file (RRF) consists of the input/output data buffer registers (DBRI and DBRO). They have one or more banks which contain one or more entries. The number of the banks and entries are configurable by the parameters. Thus, the suitable buffer for the data processing hardware on the EXU can be implemented by parameters. The mailbox (MB) is control/status registers for the SPHW. The external modules can check the statuses of the SPHW via mailboxes. The parameters required for the SPHW execution can be set via the mailboxes. The general purpose register (GPR) is used by the LSU and the EXU.

The EXU has the finite state machine (FSM), the working registers (WR) and the data path. The FSM has the states ($\langle \mathrm{EXE_i} \rangle$) to control the data path. In addition, the states ($\langle \mathrm{SYNC_i} \rangle$) to synchronize the LSU are inserted.

B. Memory Access

The LSU has the load/store instructions per the word and the line containing continuous words as shown in Fig. 2. Each of instructions can specify the number of transfers and the stride width per each word/line transfer. That is, the LSU can perform the gather/scatter operations by one instruction. Since the load/store instructions have the synchronization field, the synchronization can be also performed simultaneously with the memory access. The pointers to the bank and entry can be incremented automatically according to the notation in the instruction as shown in Fig. 2. The LSU converts the distributed data in the memory to the streamed data in the RRF, executing such sophisticated load/store instructions.

Fig. 3 (a) shows the examples implementing a double buffer for the streaming data. We assume that the line contains 4 words, the number of banks of the DBRI is 8 and each bank contains 2 entries. Fig. 3 (b) shows an example loading the 4 $\times$ 4 window. In this example, the DBRI has 4 banks containing 4 entries. As shown in these examples, the

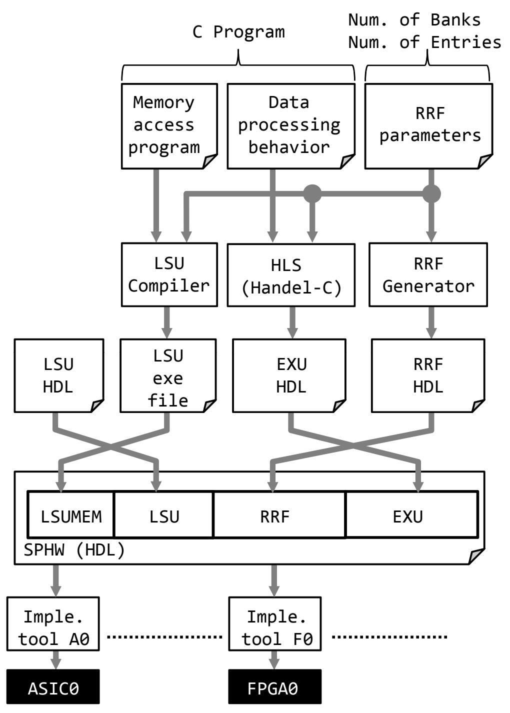


Fig. 4. SPHW Design Flow When Using Handel-C

LSU can easily realize the sophisticated memory accesses.

III. SPHW Design Flow

Fig. 4 shows the framework of the design flow which employs the SPHW. In this paper, we employ the Handel-C [1] as the HLS tool for the EXU.

For the SPHW, the memory accesses are implemented by the software programming to the load/store unit (LSU). The reconfigurable register file (RRF) is configured by the parameters to implement the optimum buffer. The data processing unit (EXU) streamly processes the sequential data on the RRF. The memory data which shows the sophisticated access patterns are put into the RRF as the stream data by the LSU.

The Handel-C is based on the concept of the communicating sequential processes (CSP) model [11] whose input/output are the streaming interface. Thus, the Handel-C hardware is a good candidate as the EXU to be connected to the RRF

Since the LSU and the EXU are executing individually across the RRF, the memory access by the LSU can overlap onto the data process by the EXU. By using the SPHW, the designer can design the hardware with the data prefetching mechanism [8] easily in the high-level description using the program and parameters.

IV. Mapping Example

A. Interface Description of EXU

Fig. 5 shows the interface description of the EXU in Handel-C. All modules share the clock signal (clk_i) and the reset signal (rst_i).

As for the DBRI and the DBRO, the number of banks (BN) and the bank width (BW) are configured by the same parameters as the RRF. The width of the entry pointer (EPW) is calculated with the number of the entries.

When the EXU is the producer to the LSU, it asserts the rls_o to execute the release synchronization. When the EXU is the consumer to the LSU, it asserts the wait_o to execute the wait synchronization and waits while the stall_i is asserted.

After the LSU loads the data needed into the DBRI, the released EXU reads the entry of the DBRI to which the nD points. These entry data in all banks can be read simultaneously via the D.D_i[BW×BN-1:0].

The EXU writes the processed data into the entries in the DBRO to which the nQ points. These entries can be written simultaneously across all banks via the q[BW×BN-1:0].

B. Color Conversion from RGB to YCrCb

Fig. 6 and Fig. 7 show an example of mapping the color conversion from RGB to YCrCb into the SPHW. The former indicates the hardware behavior in Handel-C. The latter shows the LSU program in C-like language. We have developed the tool converting the LSU program to the machine code by perl.

Now, we assume that the pixel of the image data is 32bit containing each of 8bit-R, G and B data. In this version, the SPHW supports the following features.

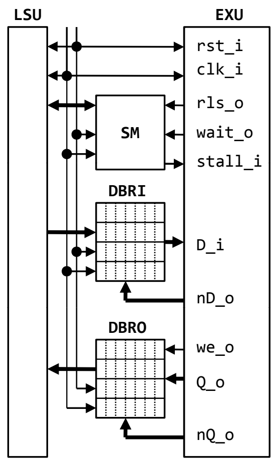
(1) The number of words in the line is 4. The word width is 32bit.
(2) The width of each bank of the DBRI/O is 32bit.
(3) The LSU supports the burst transfer containing 4 words.
(4) The LSU is the pipelined scalar processor with 3 stages.

Fig. 6 shows an overview of mapping to the SPHW. The RGB data in the memory is loaded by the LSU into the DBRI. The EXU waits until the RGB data needed is stored into the DBRI. When the LSU loads the RGB data and releases the EXU, the EXU starts to process the RGB data in the DBRI and stores YCrCb data into the DBRO. Then the EXU releases the LSU waiting the YCrCb data. After this, the LSU stores the YCrCb data into the memory.

Fig. 7 shows the LSU programming. In this case, the mailbox #0 (MB0) is used as the start flag invoking the SPHW. The read address of the RGB data and write address of the YCrCb data are set to the MB1 and MB2. The stride width per line transfer that is 16byte is set to the MB3. The number of the transferred lines to fill all banks is set to MB4, which divides the number of banks by 4. The number of total lines over the image data is set to the MB5, which divides the number of total pixels by the number of banks. The MB6 is the flag indicating that the SPHW finishes.

Fig. 7 (a) is the straight-forward programming. The LSU waits by spin-lock on the MB0 until it is set to 1. Then, the LSU resets the MB0 as the start flag and resets the MB1 as the end flag. The LSU loads the lines into all banks and performs the release synchronization (RLS). Then, the LSU performs the wait synchronization (WAIT) and stores the processed lines in the DBRO into the memory. This program is very simple and intuitive but suffers from the memory access latency.

Fig. 7 (b) is the LSU program of which the software pipelining [8] is applied to hide the memory access latency. In the software pipelining, the load instruction (LLS) and the store instruction (SLS) in the main loop are copied to the front of the main loop and the back of it respectively. In

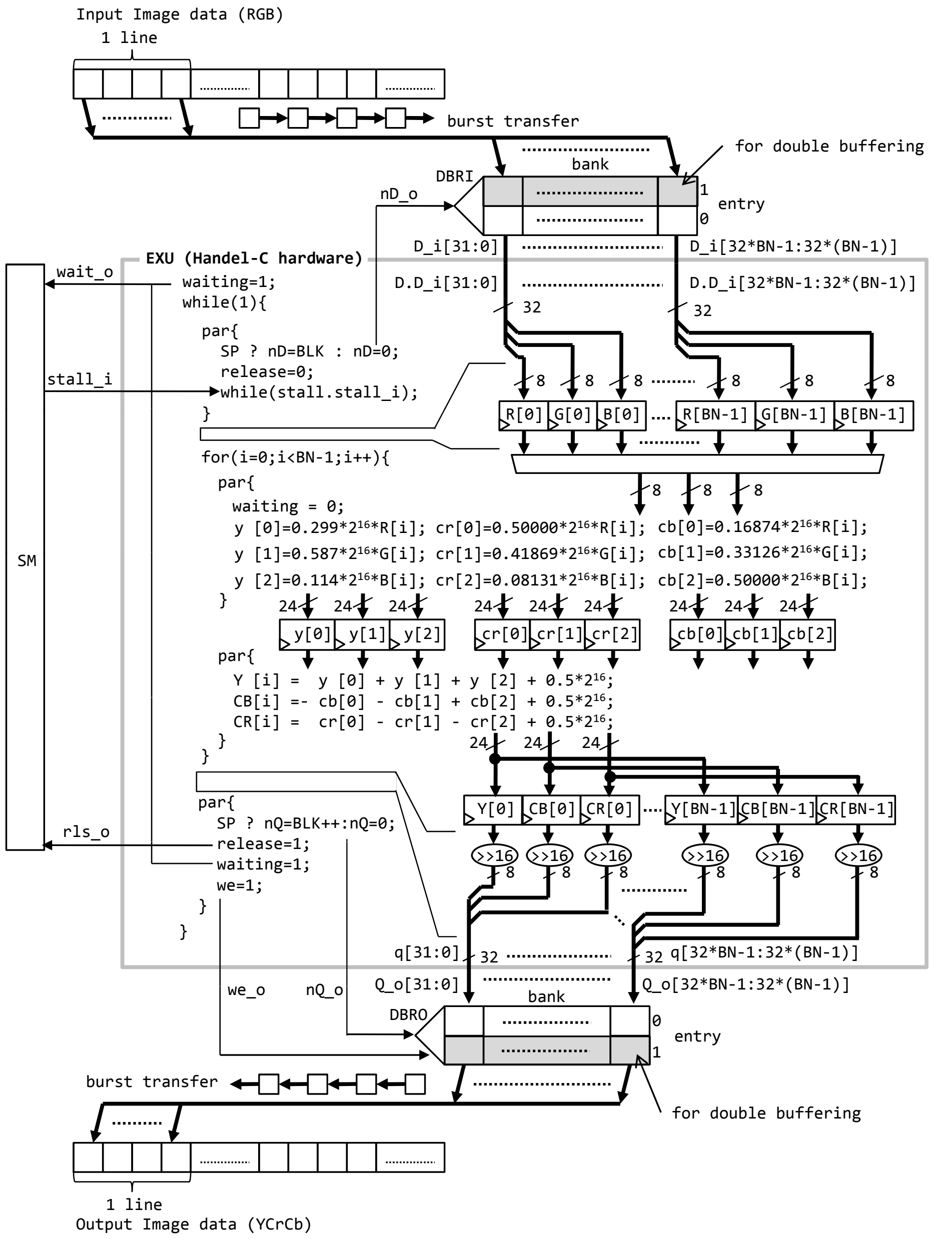


(a) Image of EXU Interface

```
// clock and reset ===================================================================
interface port_in( uint 1 clk_i ) clk( );
interface port_in( uint 1 rst_i ) rst( );
set clock = internal clk.clk_i;
set reset = internal rst.rst_i;
// SM Interface ======================================================================
static uint 1 waiting;
static uint 1 release;
interface port_in  (uint 1 stall_i )STALL( ) with {vhdl_type = "std_logic"} ;
interface port_out ( )WAIT(signal<uint 1>wait_o=waiting) with {vhdl_type = "std_logic"};
interface port_out ( )RLS (signal<uint 1>rls_o =release) with {vhdl_type = "std_logic"};
// RRF Interface =====================================================================
static signal <uint (BN * 32)> q;
static signal <uint 1> we;
static uint EPW nD;
static uint EPW nQ;
interface port_in  (uint (BN * BW) D_i)D( ) with {vhdl_type = "std_logic_vector"};
interface port_out ( )ND(signal<uint EPW    >nD_o=nD) with {vhdl_type="std_logic_vector"};
interface port_out ( )Q (signal<uint (BN*BW)>Q_o =q ) with {vhdl_type="std_logic_vector"};
interface port_out ( )NQ(signal<uint EPW    >nQ_o=nQ) with {vhdl_type="std_logic_vector"};
interface port_out ( )WE(signal<uint 1      >we_o=we) with {vhdl_type="std_logic"};
//===================================================================================
```

(b) Description of EXU Interface in Handel-C

Fig. 5. Interface Description of EXU in Handel-C.



```
waiting=1;
while(1){
  par{
    SP ? nD=BLK : nD=0;
    release=0;
    while(stall.stall_i);
  }
  for(i=0;i<BN-1;i++){
    par{
     waiting = 0;
     y [0]=0.299*2^16*R[i]; cr[0]=0.50000*2^16*R[i]; cb[0]=0.16874*2^16*R[i];
     y [1]=0.587*2^16*G[i]; cr[1]=0.41869*2^16*G[i]; cb[1]=0.33126*2^16*G[i];
     y [2]=0.114*2^16*B[i]; cr[2]=0.08131*2^16*B[i]; cb[2]=0.50000*2^16*B[i];
    }
    par{
     Y [i] =  y [0] + y [1] + y [2] + 0.5*2^16;
     CB[i] =- cb[0] - cb[1] + cb[2] + 0.5*2^16;
     CR[i] =  cr[0] - cr[1] - cr[2] + 0.5*2^16;
    }
  }
  par{
    SP ? nQ=BLK++:nQ=0;
    release=1;
    waiting=1;
    we=1;
  }
}
```

Fig. 6. Mapping Image from RGB2YCrCb to EXU.

the main loop, the data used at the next iteration is loaded at the current iteration. Thus, the memory accesses of the

```
-- MB0 : Start flag
-- MB1 : Read  address
-- MB2 : Write address
-- MB3 : Stride width (16)
-- MB4 : Number of burst trans.
--       = (num. of banks) / 4
-- MB5 : Number of total trans.
--       = (num. of pixels) / banks
-- MB6 : End flag
do{
  while( MB0 == R0 );
  MB0=0; MB6=0;
  do{
    LLS(B, MB1, MB3, MB4, RLS );
    SLS(B, MB2, MB3, MB4, WAIT);
    R2 = R2 + R1;
  }while( MB5 > R2 )
  MB6 = 1;
}while(1);
```

(a) Straight-forward LSU program

```
do{
  while(MB0 == R0);
  MB0=0; MB6=0;
  LLS(BE, MB1, MB3, MB4,RLS);
  R2++;                  copy
  do{
    LLS(BE, MB1, MB3, MB4, RLS );
    SLS(BE, MB2, MB3, MB4, WAIT);
    R2++;
  }while( MB5 > R2 );    copy
  SLS(BE, MB2, MB3, MB4, WAIT);
  MB6 = 1;
}while(1);
```

(b) Software-pipelined LSU program

Fig. 7. LSU Programming of RGB2YCrCb.

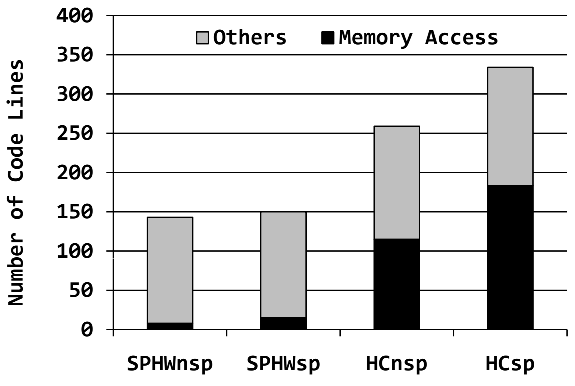


Fig. 8. Number of Code Linse.

LSU are overlapped with the data processing of the EXU. For the EXU, as shown in Fig. 7 (a), the number of entries of the DBRI/O becomes two by denoting the SP as 1. In each iteration in the main loop, the entry pointer (BLK) is toggled. As mentioned above, the double buffering can be implemented easily.

V. Experiment and Discussion

A. Design Burden

To evaluate a burden of hardware design for the color conversion mentioned above, we compare the SPHW version with the cases of which the data processing hardware and memory access controller are described in Handel-C (HC). In this evaluation, we measured the number of code lines [1].

[1]As for the comparison with the cases of which the whole hardware is described in HDL, we have already shown the validity of the SPHW in [9].

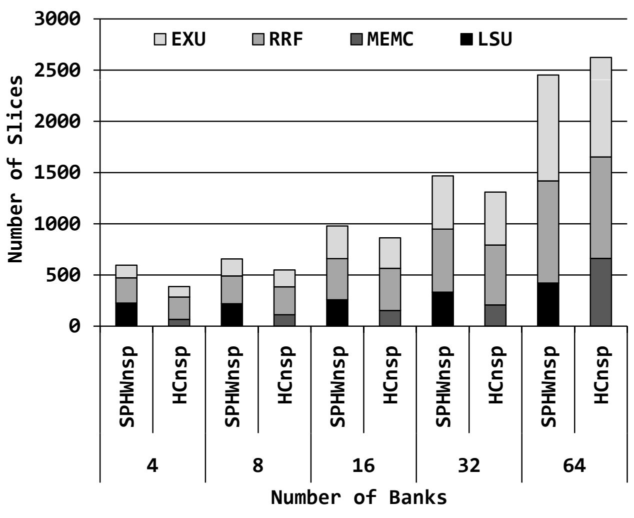


Fig. 9. Number of Slices for SPHWnsp and HCnsp.

Fig. 8 shows the result of this comparison. The vertical axis means the number of code lines. The horizontal axis indicates the cases of hardware design. In the horizontal axis, the *nsp means the case not employing the software-pipelining, while the *sp means the case employing it. All cases used the same data processing hardware (EXU).

The result shows that the SPHW can significantly reduce the number of code lines compared with the full Handel-C description. The code lines of 45% are reduced in the total number of the straight-forward cases (SPHWnsp vs. HCnsp), while those of 55% are reduced in the software-pipelined cases (SPHWsp vs. HCsp).

As for the memory access controller, the SPHW reduces the code lines of about 93% compared with the full Handel-C cases. This is because the SPHW can easily describe the memory access controller and the data-prefetching by the LSU programming as shown in Fig. 7.

This fact indicates that by introducing the SPHW to the HLS tool the data processing hardware containing the sophisticated memory access controller can be implemented with lower burden than the conventional C-level hardware design.

B. Implementation Result (SPHW vs. HC)

Varying the number of banks, we implemented the hardware mentioned above into the Virtex5 FPGA of which the speed grade is 10. We used the ISE12.2 in implementation. Fig. 9 shows the result of the straight-forward hardware while Fig. 10 indicates the result of the software-pipelined hardware. The vertical axis means the number of slices consumed by the EXU, the LSU, the RRF, and the memory access controller described in Handel-C (MEMC). The horizontal axis indicates the same meaning as Fig. 8.

Both cases show the same tendency. Where the number of banks is from 4 to 32, the SPHW shows larger amount of hardware than that of the full Handel-C hardware. This is because the SPHW has the LSU which is an embedded processor performing the memory access. This overhead makes SPHW larger than the full Handel-C hardware. However, when the number of banks is 64, the size of the SPHW is lower than the full Handel-C hardware. This is because the

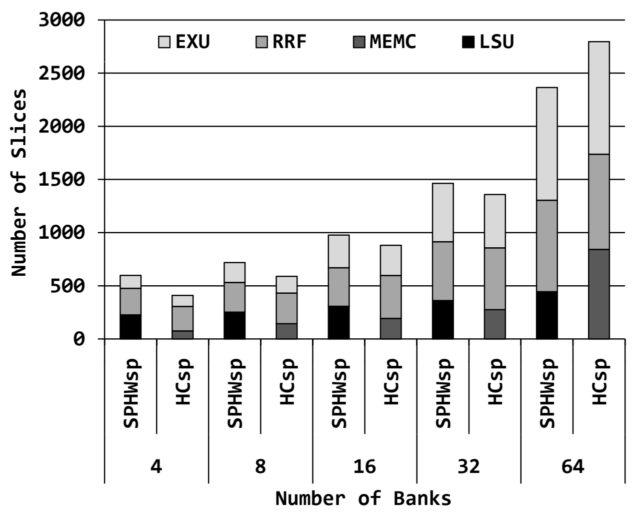


Fig. 10. Number of Slices for SPHWsp and HCsp.

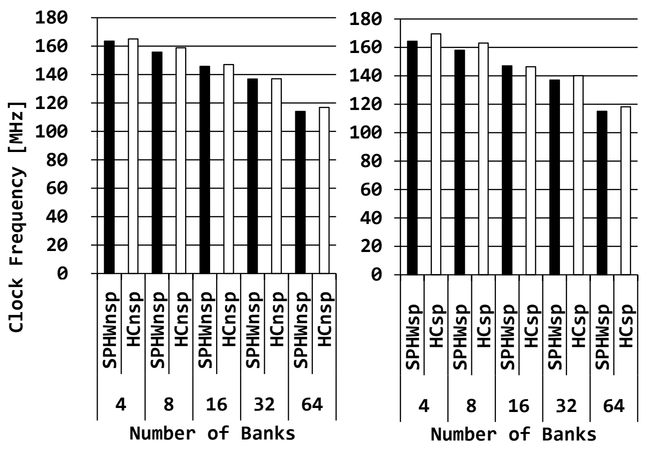


Fig. 11. Clock Frequency of SPHW and HC.

Handel-C hardware becomes more complex and larger in order to handle the larger number of banks by the large one-hot decoders and the multiplexers generated by the Handel-C compiler. In contrast, the SPHW can handle a lot of banks efficiently by the sophisticated load/store instructions of the LSU.

Fig. 11 shows the clock frequency reported by the ISE12.2. According to the report generated by the ISE12.2, the critical path resides on the data processing hardware (EXU). The result shows that the SPHW can be combined with the data processing hardware generated by the HLS tool, without the bad influence to the hardware speed.

C. Implementation Result (SPHWnsp vs. SPHWsp)

In order to evaluate an influence to the amount of the hardware due to introducing the mechanism hiding memory access latency by software-pipelining, we compare in detail the logic resources consumed by the SPHWsp and the SPHWnsp. Fig. 12 shows the number of look-up tables (LUTs), the number of flip-flops (FFs) and the number of slices. In the FPGA used, the slice contains 4 FFs and 4 LUTs.

In Fig. 12 (a), the reconfigurable register file (RRF) significantly increases the number of LUTs as the number of banks becomes larger in the software-pipelined SPHW (SPHWsp). In contrast, the RRF of the straight-forward SPHW (SPHWnsp) significantly increase the number of FFs as shown in Fig. 12 (b). This is because the used logic synthesis tool (XST) assigns the banks of RRF with one entry in SPHWnsp to the registers, while it assigns the banks of RRF with two or more entries in the SPHWsp to the LUTs. Finally, the SPHWnsp and SPHWsp show the comparable number of slices as shown in Fig .12. This fact shows that the software pipelining does not significantly affect the circuit size despite fact the number of the bank entries is doubled compared to the straight-forward hardware.

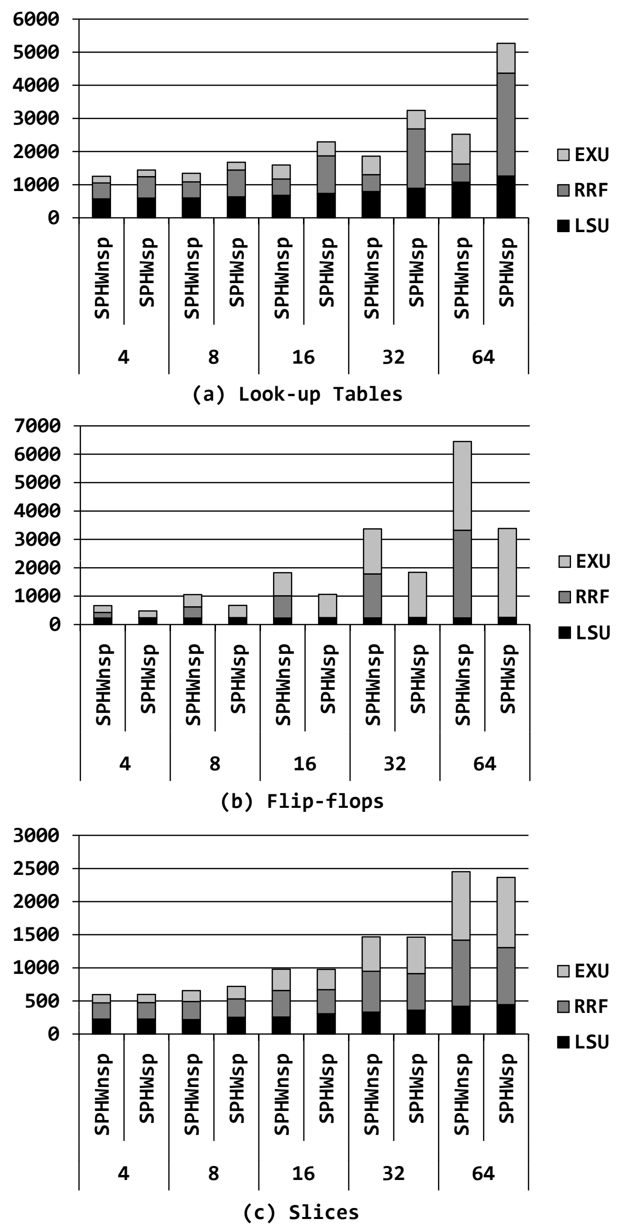


Fig. 12. Implementation Result (SPHWnsp vs. SPHWsp).

Each LUT of the FPGA used to the bank of the DBRI and DBRO can contain 64 entries. Thus, until the number of entries exceeds 64, the circuit size of the SPHWsp does not expand extremely.

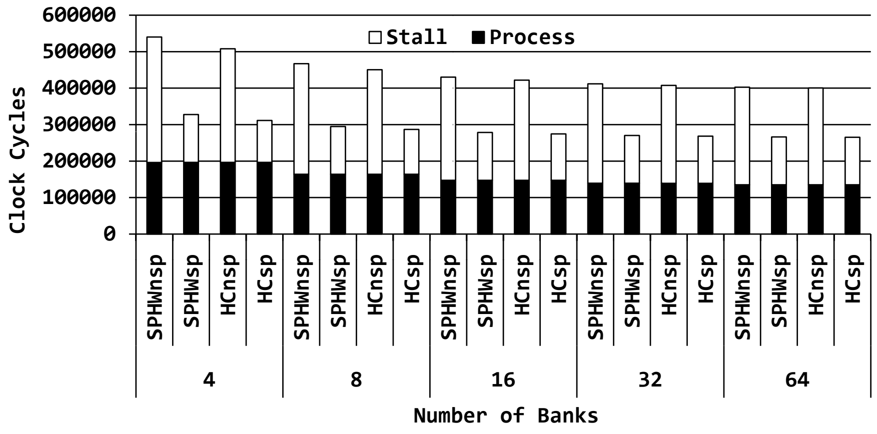


Fig. 13. Performance Result.

D. Performance Result

Fig. 13 shows the performance result when the line transfer from the memory to the LSU consumes 6 clocks. The image size is 256 × 256. The breakdowns that the EXU shows are also included in this figure. The stall means the clock cycles consumed by the EXU waiting for the memory access. The process indicates the clock cycles of which the EXU consumes to process the data.

Compared with the full Handel-C hardware (HC*), the SPHW shows slight performance degradation ranging from 0.4% to 6.4%. This is because the LSU consumes 3 clocks to fill the pipeline in branching since it is a pipeline processor with 3 stages. In full Handel-C hardware, the memory access behavior is well tuned so as to take only 1 clock for branching. This difference appeared as the performance difference. However, we think such little performance differences can be compensated enough by the SPHW lowering design burden.

By hiding memory access latency, the SPHWsp can improve the performance of 1.51 to 1.65 times compared with the SPHWnsp. We were able to perform such tradeoff among the number of banks, the circuit size, the clock frequency, and the performance easily and quickly by only changing parameters and the LSU program.

VI. Conclusion

The semi-programmable hardware is a design-level hardware architecture residing on the pass of which C program with memory accesses is converted to hardware. The SPHW realizes the memory access controller and the buffer by writing the software program and parameters respectively.

In this paper, we have introduced the SPHW as the data processing hardware into a real commercial HLS tool, Handel-C. By using the SPHW providing the register-based data access interface, we have demonstrated that the HLS tool can easily write the hardware accessing to the memory in C. This is because this interface hides the detail of the memory devices and the memory access patterns, by providing the data processing hardware with the simple stream data. For hiding memory access latency, the simple software-pipelining is able to be applyed to the memory access program and the parameters of the buffer. Consequently, we can realize the data processing hardware with data-prefetching mechanism at the complete C-level design entry, with lower burden.

As future work, we will introduce the SPHW into more HLS tools and evaluate using more application programs.

References

[1] Mentor Graphics, “Handel-C Synthesis Methodology,” http://www.mentor.com/products/fpga/handel-c/, 2010.

[2] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, “Using global code motions to improve the quality of results for high-level synthesis,” *IEEE Trans.on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 302–312, 2004.

[3] D. Lau, O. Pritchard, and P. Molson, “Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions,” in *IEEE Symp. on Field-Programmable Custom Computing Machines*, 2006, pp. 45–56.

[4] X. Liang and J. Jean, “Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems,” *The Journal of Supercomputing*, vol. 1, no. 19, pp. 77–91, May 2001.

[5] Mitrionics, *Mitrion Users'Guide 1.5.0-001*. Mitrionics, 2008.

[6] J. Park and P. C. Diniz, “Synthesis of Pipelined Memory Access Controllers for Streamed Data Applications on FPGA-based Computing Engines,” in *Proc. of intl. symp. on Systems synthesis*, Oct. 2001, pp. 221–226.

[7] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall, 2005.

[8] S. P. Vanderwiel, “Data Prefetch Mechanisms,” *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[9] A. Yamawaki, S. Serikawa, and M. Iwane, “An Efficient Hardware Architecture from C Program with Memory Access to Hardware,” in *Proc. of the 2010 International Conference on Computational Science and Its Applications, Part II*, 2010, pp. 488–502.

[10] A. Yamawaki, S. Serikawa, and M. Iwane, “Introducing Semi-programmable Hardware to a Real High-Level Synthesis Tool,” *Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2010,* WCECS 2010, 20-22 October, 2010, San Francisco, USA, pp. 175–180.

[11] C. A. R. Hoare, “Communicating Sequential Processes,” *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.