2006-1095: DATAFLOW SCHEDULING AND EXPLORING DIGITAL SYSTEM DESIGN ALTERNATIVES

Chia-Jeng Tseng, Bucknell University

Dataflow Scheduling and Exploring Digital System Design Alternatives

Abstract

Dataflow scheduling is a powerful technique for exploring design alternatives at the system level. Efficient scheduling is, however, a complicated task. Software tools are often used in high-level synthesis to schedule a design specification. Since high-level synthesis is not yet widely accepted as a method of design entry, most students do not appreciate the significance of scheduling to the tradeoffs of system-level digital design. In this paper, we use a sorting algorithm to investigate the role of scheduling to the design of sorting networks. In class, we begin with a serial specification. Then, the as-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) scheduling algorithms are applied to the original description. Students are also encouraged to define their own schedules and compare them with the serial, ASAP, and ALAP schedules. The impact of various schedules on the number of sorting elements, registers, multiplexers, and control steps are analyzed. After students have derived much insight of the problem, several scheduling algorithms available in the literature such as the force-directed scheduling are studied. To investigate the impact of dataflow scheduling on hardware implementation, the data paths and controllers of two scheduled dataflow specifications are presented. The tradeoffs between hardware cost and system performance is analyzed. The methodology was taught in an "Advanced Digital Design" course as a design space exploration skill. Students' feedback indicated that the method was very systematic and robust, and constituted a powerful digital design technique. Given the instruction set specification of a computer, the technique is also applicable to explore the design space of a central-processing-unit (CPU). In addition, the materials give students a clear demonstration for the structures of a CPU, including the separation of data paths and controller as well as the impact of multiple functional units. With these considerations in mind, the module was also offered in the course of "High-Performance Computer Architectures" for students to understand the fundamentals of CPU design.

1. Introduction

As the complexity of digital design continues to increase, system level design is becoming the focus of digital design activities. These days digital design often begins with an algorithmic specification. The algorithmic description is then scheduled [8, 9]. The structure of a design is generated based on the scheduled data and control flow specification.

Given a scheduled dataflow specification, a clique-partitioning procedure can be applied to the synthesis of data paths in a digital system [9]. Slicing techniques can be used to produce a controller for the data paths [10]. In other words, hardware resources requirement is determined by the scheduled data flow. Indeed, dataflow scheduling has become an important technique for exploring design alternatives. In this paper, we describe how we used the design of sorting networks to teach students the new paradigm of dataflow scheduling for exploring digital system design alternatives.

Sorting problems have been extensively studied; numerous software algorithms including bubble sort, quicksort, etc., are available in the literature [5, 7]. Through the study of the traditional software algorithms, students learn the fundamental implication, limitation, and flexibility of von Neumann architecture.

We also address the impact of algorithm organization on dataflow scheduling, a territory often overlooked by traditional scheduling methods. One aspect is related to how data dependency is described and the other is about specifying a function as control or data flow. A sorting algorithm can be described as a combination of data flow and control flow. A list of statements containing a single entry and a single exit point, respectively, is defined as a basic block [1]. An algorithm generally consists of a number of basic blocks linked together by control flow constructs such as the *switch* instruction in the C language [6] or a concurrent fork statement [3, 10]. Unless a special technique is applied to schedule the hardware description, dataflow analysis is confined to the context of a basic block. The amount of parallelism is therefore limited by the instructions embedded in each basic block. A sorting algorithm, on the other hand, may also be specified as simple dataflow. In this case, a sorting element which consists of a comparator and a multiplexer is used as the basic operator for sorting. This type of specification may provide rich opportunity for dataflow optimization. In other words, a design specification of sorting function may offer limited parallelism due to the way that data dependency and control flow is specified. A different description for the same objective may enable dataflow scheduling to identify maximum parallelism through data dependency analysis. In this paper, different styles of hardware descriptions will be provided to address the impact of design specification to dataflow scheduling.

The aforementioned methodology was presented to the senior and first year graduate students taking "Advanced Digital Design" and "Computer Architectures" courses. The laboratory was assigned to students taking "Advanced Digital Design" as one of their term projects. The "Advanced Digital Design" course covers VHDL [2], digital system design using micro-architectural modeling techniques, and advanced topics in logic optimization. The main objective of the course was to train students the capability of performing system-level design. The knowledge of scheduling was introduced for the purpose of exploring system-level design tradeoffs. Based on the feedback given by students, the module was very instrumental to the study of digital system design and computer architectures.

The paper is organized as follows. Section 2 introduces two software sorting algorithms: bubble sort and quicksort. Section 3 presents three dataflow specifications for sorting. Section 4 discusses several scheduling methods. Section 5 investigates the hardware implementation of two scheduled dataflow descriptions. Section 6 describes the impact of separating data and control flow specification. Finally, Section 7 contains concluding remarks of the paper and an assessment for the effectiveness of the course module.

2. Software Sorting Algorithms

In order to identify the order of data, sorting involves comparisons. The simplest algorithm for sorting a given number of integers is probably the bubble sort. Let N integers be given. The following C-like routine describes its details. Essentially, the routine contains two loops. The outer loop begins with the first datum, compares it with all the data following it, one by one, until the last one, to extract the smallest data in the list. In the first iteration, the smallest data is extracted. In the second iteration, the second smallest number is isolated. This process repeats itself until the list of data is completely sorted. To achieve the goal, during an iteration of the outer loop, the inner loop walks through all the data following the reference datum in the outer loop one by one to identify the smallest datum in the remaining list. When both loops are completed, the data array contains a list of sorted data.

Bubble sort:

```
for (i = 0, i < N-1; i++) {
    for (j = i+1, j < N; j++) {
            if (A(i) > A(j)) {
                    temp = A(i);
                    A(i) = A(j);
                    A(j) = temp;
              }
        }
  }
```

In the algorithm, it takes $(N-1)*(N-2)/2$ steps of sorting operations to complete a sorting task. A sorting operation contains a comparison and three steps of data swapping. Of course, this is a natural outcome of applying conventional software development concept to a von Neumann machine which assumes that a single arithmetic or logic operation is performed in each step.

An efficient algorithm called quicksort applies a divide-and-conquer strategy to recursively partition a list of data into two lists and a pivotal element. Each time a list is partitioned into two sub-lists on the two sides of the pivotal element. If the pivotal element is located at half-way of the list, the process results in a time complexity of $N * \log N$. The average number of steps required by quicksort is $N * \log N$; however, the worst-case complexity is still of N^2.

So far we have investigated the issues of software algorithms for sorting. A software algorithm, which is often run on a von Neumann machine, assumes that a single instruction is executed in each step. In software implementation, the tradeoffs between time and memory space are often explored. Suitable data structures may be required for an implementation. In the next section, we will examine some dataflow specifications for sorting.

3. Dataflow Descriptions for Sorting

As indicated in Section 2, an efficient sorting algorithm such as the quicksort can have the average performance of $N * \log_2 N$. Given N numbers, effectively $\left\lfloor \frac{N}{2} \right\rfloor$ comparisons can be performed simultaneously. As a result, it is possible to complete the sorting task in N steps.

For clarity and presentation convenience, instead of using N input data, we will assume that five random fixed-width, e.g., 16-bit, integers are given as the input data to illustrate dataflow specification of sorting. Let the five integers be represented by IA, IB, IC, ID, and IE.

As described in Section 2, depending on the way an algorithm is designed, the average number of steps required for a software implementation can be in the order of $N \log_2 N$ to N^2 for a given N numbers. Similarly, depending on which pairs of data are compared and the order of comparing these pairs of data, there are numerous ways of specifying the dataflow of sorting five numbers. It requires creative thoughts to define an efficient description. In this section, three different descriptions are presented to demonstrate the varieties of dataflow specification. In the future, writing efficient dataflow specification may become the focus of creating a good digital design.

In the following descriptions, the symbols IA, IB, IC, ID, and IE represent five sets of input ports. Each set of input ports, e.g., IA, is assumed to contain the same number of bits; for example, 16 bits. The symbols OA, OB, OC, OD, and OE stand for five sets of output ports. Each of the other variables is represented by the alphabet A, B, C, D, or E, followed by an integer. The integer is used to identify the level in the data flow specification. The first set of statements store the data available at the input ports to A0, B0, C0, D0, and E0. The last set of statements, which are optional, show the association of the output ports OA, OB, OC, OD, OE with their respective sources. Each set of the output ports actually captures its source data in real time. We will use the statements between the first line and the last line to describe the issues of dataflow scheduling. All of the statements listed in the same line, separated by semicolons, are assumed to proceed concurrently in the same cycle. Given two adjacent lines of statements, those statements placed in the second line are assumed to be executed in the next cycle following the completion of the statements in the first line.

One way of describing the dataflow of sorting five numbers is to mimic the software algorithm for bubble sort.

Routine-1: A serial bubble-sort dataflow description of five integers

```
A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;  E0 = ID;
(A1, B1) = sort(A0, B0);
(A2, C2) = sort(A1, C0);
(A3, D3) = sort(A2, D0);
(A4, E4) = sort(A3, E0);
(B5, C5) = sort(B1, C2);
```

```
(B6, D6) = sort(B5, D3);
(B7, E7) = sort(B6, E4);
(C8, D8) = sort(C5, D6);
(C9, E9) = sort(C8, E7);
(D10, E10) = sort(D8, E9);
OA = A4;  OB = B7;  OC = C9;  OD = D10;  OE = E10;
```

A second method of specifying sorting algorithm is a modified bubble-sort description presented in Routine-2. This algorithm requires the same number of steps as Routine-1. However, the data dependency specified in this algorithm offers opportunities for generating shorter schedules.

Routine-2: An alternative serial bubble-sort dataflow description of five integers

```
A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;  E0 = ID;
(A1, B1) = sort(A0, B0);
(B2, C2) = sort(B1, C0);
(A3, B3) = sort(A1, B2);
(C3, D3) = sort(C2, D0);
(B4, C4) = sort(B3, C3);
(D4, E4) = sort(D3, E0);
(A5, B5) = sort(A3, B4);
(C5, D5) = sort(C4, D4);
(B6, C6) = sort(B5, C5);
(A7, B7) = sort(A5, B6);
OA = A7;  OB = B7;  OC = C6;  OD = D5;  OE = E4;
```

A third method of specifying sorting algorithm is a "balanced" description. Data dependency specified in this dataflow evenly spreads across all the statements. As a result, the task of sorting five numbers can be completed in five steps.

Routine-3: A balanced sorting dataflow description of five integers

```
A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;  E0 = ID;
(A1, B1) = sort(A0, B0);  (C1, D1) = sort(C0, D0);
(B2, C2) = sort(B1, C1);  (D2, E2) = sort(D1, E0);
(A3, B3) = sort(A1, B2);  (C3, D3) = sort(C2, D2);
(B4, C4) = sort(B3, C3);  (D4, E4) = sort(D3, E2);
(A5, B5) = sort(A3, B4);  (C5, D5) = sort(C4, D4);
OA = A5;  OB = B5;  OC = C5;  OD = D5;  OE = E4;
```

The first dataflow description does not offer any possible parallelism. A more relaxed data dependency is embedded in the second specification. As a result, shorter schedules can be generated from it. The data dependency in the third description is pretty much evenly spread over all the variables; maximum parallelism is available. In the following sections, we will use this description to illustrate the issues of dataflow scheduling.

4. Dataflow Scheduling

Given a dataflow specification, scheduling assigns all statements to a number of steps based on the data dependency defined by the description. The as-soon-as-possible schedule assumes that each statement is exercised at its earliest possible time. The as-late-as-possible schedule, on the other hand, places each statement at the latest possible cycle. Optimal scheduling is a complicated task. Heuristics are normally applied to identify a good schedule for an efficient hardware implementation. For example, a method called force-directed scheduling, which is based on the criterion of load balancing, is presented in [8].

To generate the ASAP schedule, the process begins with placing the first statement in the first cycle. If the second statement uses the output generated by the first statement, it is placed in the following cycle of the first statement, i.e., in the second cycle. Otherwise, it is placed in the same cycle as the first statement. This process continues for the third, fourth, etc., statements until all the statements are scheduled.

To generate the ALAP schedule, the process begins with placing the last statement in the last cycle. If the output generated by the second last statement is used by the last statement, it is placed in the previous cycle of the last statement, i.e., in the second to the last cycle. Otherwise, it is placed in the same cycle as the last statement. This process continues for the third to the last, fourth to the last, etc., statements until all the statements are scheduled.

4.1 Case Study #1

In Routine-1, starting from the second statement, each statement uses at least one result generated by the previous statement. There is no room for schedule adjustment.

4.2 Case Study #2

The ASAP/ALAP schedules for Routine-2 turned out to be identical; they are given in Routine-4. Given the ASAP and ALAP schedules, the same schedule is derived from the force-directed scheduling method.

Routine-4: The ASAP/ALAP schedule of Routine-2

```
A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;  E0 = ID;
(A1, B1) = sort(A0, B0);
(B2, C2) = sort(B1, C0);
(A3, B3) = sort(A1, B2);  (C3, D3) = sort(C2, D0);
(B4, C4) = sort(B3, C3);  (D4, E4) = sort(D3, E0);
(A5, B5) = sort(A3, B4);  (C5, D5) = sort(C4, D4);
(B6, C6) = sort(B5, C5);
(A7, B7) = sort(A5, B6);
```

```
OA = A7;  OB = B7;  OC = C6;  OD = D5;  OE = E4;
```

4.3 Case Study #3

Based on the order of statements given in the schedule shown in Routine-3, a serial schedule is given in Routine-5.

Routine-5: A serial dataflow description of Routine-3

```
A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;  E0 = ID;
(A1, B1) = sort(A0, B0);
(C1, D1) = sort(C0, D0);
(B2, C2) = sort(B1, C1);
(D2, E2) = sort(D1, E0);
(A3, B3) = sort(A1, B2);
(C3, D3) = sort(C2, D2);
(B4, C4) = sort(B3, C3);
(D4, E4) = sort(D3, E2);
(A5, B5) = sort(A3, B4);
(C5, D5) = sort(C4, D4);
OA = A5;  OB = B5;  OC = C5;  OD = D5;  OE = E4;
```

The ASAP and ALAP schedules of Routine-3 are identical; they are represented in Routine-6. Since the two schedules are the same, there is no room for further optimization. As a result, the force-directed scheduling method based on the ASAP/AEAP schedule also produces the same schedule.

Routine 6: The ASAP and ALAP dataflow description of Routine-3

```
A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;  E0 = ID;
(A1, B1) = sort(A0, B0);  (C1, D1) = sort(C0, D0);
(B2, C2) = sort(B1, C1);  (D2, E2) = sort(D1, E0);
(A3, B3) = sort(A1, B2);  (C3, D3) = sort(C2, D2);
(B4, C4) = sort(B3, C3);  (D4, E4) = sort(D3, E2);
(A5, B5) = sort(A3, B4);  (C5, D5) = sort(C4, D4);
OA = A5;  OB = B5;  OC = C5;  OD = D5;  OE = E4;
```

5. Hardware Implementation for Scheduled Dataflow Descriptions

There are three types of generic resources in the data paths of a digital design, including registers, operators, and interconnections. Resource allocation of a given dataflow description is determined by the constraints embedded in the description.

Lifetime analysis can be used to identify sharing constraints for registers. The same register can be assigned to the input operand and the output variable of the same statement if flip-flops are used for the variables. In digital design, this refers to the read/modify/write scheme. Usage analysis in each cycle can be used to determine the constraints for sharing an arithmetic and logic unit or an interconnection unit. We use the clique-partitioning procedure described in [9] to generate the hardware structure of a given schedule.

The three types of data-path components of a design can be allocated separately. An integrated approach can also be applied. No matter what method is applied, a general observation is that it will only produce an optimal solution for a small set of descriptions. In other words, no methods are universally effective. Indeed, the problem of digital system design is very complicated. An efficient method often uses a simple strategy to generate high-quality designs.

If data operators are the focus, a serial schedule would generate a single-operator structure, which is equivalent to the von Neumann architecture of early computers.

As depicted in Figure 1, a sorting element consists of a comparator and a multiplexer. For clarity, we will use the schematic shown in Figure 2 to represent a sorting element. The comparator in Figure 1 is replaced by two small circles in Figure 2. Generally speaking, a sorting network consists of registers, sorting elements, multiplexers, and a controller. The total cost of a design is comprised of the chip area occupied by these circuit components. The cycle period is determined by the longest delay from a primary input or the output of a flip-flop to a primary output or the input to a flip-flop. The overall performance is characterized by the product of the cycle period and the number of cycles. In practice, the implementation cost is technology dependent. In this study, we use FPGA as the target technology [11, 12].

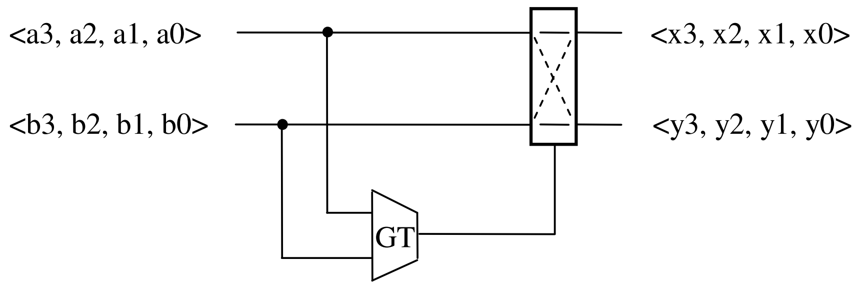


Figure 1: A Sorting Element Comprising of A Comparator and A Switch Element

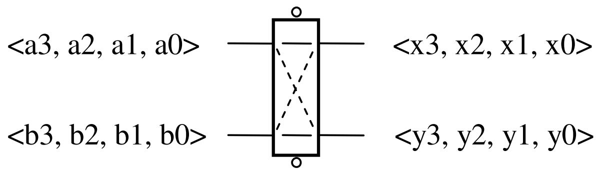


Figure 2: A Sorting Element

We will use the balanced dataflow specification presented in Routine-3 to explore the design alternatives of sorting networks. Three designs for two schedules, including a serial schedule, and the ASAP/ALAP schedule which turns out to be the same as the original balanced description, are presented in this section to demonstrate the impact of dataflow scheduling on hardware implementation. To simplify the presentation, instead of five, four input variables are used in the dataflow descriptions. Also, two input signals, called "*strobe*" and "*reset*," and one output signal, named "*done*," are included in the descriptions for handshaking purpose. The "*strobe*" signal initiates a sorting request and the "*done*" signal indicates the completion of a sorting task. The "*reset*" signal is used as an external reset for the "*done*" signal. The four variables A0, B0, C0, and D0 included in the first line of each schedule are for latching the input values. For clarity, they are not considered for further optimization while register allocation is performed.

5.1 Single sorting element implementation

Routine-7: A serial sorting dataflow description of four input variables

```
if (strobe = '1') then
  begin
    A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;      done = '0';          -- I
    (A1, B1) = sort(A0, B0);                                          -- 1
    (C1, D1) = sort(C0, D0);                                          -- 2
    (B2, C2) = sort(B1, C1);                                          -- 3
    (A3, B3) = sort(A1, B2);                                          -- 4
    (C3, D3) = sort(C2, D1);                                          -- 5
    (B4, C4) = sort(B3, C3);                                          -- 6
    OA = A3;  OB = B4;  OC = C4;  OD = D3;  done = '1';              -- X
  end
else
  begin
    if (reset = '1') then
      begin
        done <= '0';
      end;
  end;
```

In order to perform register allocation, lifetime analysis is done for the twelve variables. The result is presented in Table 1, where D and L stand for the status of a variable being dead or live, respectively, in a time slot.

Table 1: Lifetime analysis result for Routine-7

	A1	B1	C1	D1	B2	C2	A3	B3	C3	D3	B4	C4
Index	1	2	3	4	5	6	7	8	9	10	11	12
Step1	L	L	D	D	D	D	D	D	D	D	D	D
Step2	L	L	L	L	D	D	D	D	D	D	D	D
Step3	L	L	L	L	L	L	D	D	D	D	D	D
Step4	L	D	D	L	L	L	L	L	D	D	D	D
Step5	D	D	D	L	D	L	L	L	L	L	D	D
Step6	D	D	D	D	D	D	L	L	L	L	L	L
StepX	D	D	D	D	D	D	L	D	D	L	L	L

Based on the dataflow in Routine-7 and the lifetime analysis result in Table 1, an undirected graph identifying the compatible variable-pairs for sharing is given below. Applying the clique-partitioning procedure described in [9], the twelve variables can be assigned to four registers.

```
                    (1, 7)  (1, 8)  (1, 9)  (1,10)  (1,11)  (1,12)
(2, 5)  (2, 6)  (2, 7)  (2, 8)  (2, 9)  (2,10)  (2,11)  (2,12)
(3, 5)  (3, 6)  (3, 7)  (3, 8)  (3, 9)  (3,10)  (3,11)  (3,12)
                                (4, 9)  (4,10)  (4,11)  (4,12)
                (5, 7)  (5, 8)  (5, 9)  (5,10)  (5,11)  (5,12)
                                (6, 9)  (6,10)  (6,11)  (6,12)
                                                (8,11)  (8,12)
                                                (9,11)  (9,12)
```

The result of register allocation is given below and the new dataflow description is presented in Routine-8.

{ 2, 5, 8, 11 } or equivalently, { B1, B2, B3, B4 }
{ 3, 6, 9, 12 } or equivalently, { C1, C2, C3, C4 }
{ 1, 7 } or equivalently, {A1, A3 }
{ 4, 10 } or equivalently, { D1, D3 }

Routine-8: Resultant dataflow description of Routine-7 from register allocation

```
if (strobe = ‘1’) then
  begin
    A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;      done = ‘0’;       -- I
    (A1, B1) = sort(A0, B0);                                      -- 1    S0
    (C1, D1) = sort(C0, D0);                                      -- 2    S1
    (B1, C1) = sort(B1, C1);                                      -- 3    S2
    (A1, B1) = sort(A1, B1);                                      -- 4    S3
    (C1, D1) = sort(C1, D1);                                      -- 5    S4
```

```
        (B1, C1) = sort(B1, C1);                                        -- 6      S5
        OA = A1;  OB = B1;  OC = C1;  OD = D1;   done = ‘1’;            -- X
      end
    else
      begin
        if (reset = ‘1’) then
          begin
            done <= ‘0’;
          end;
      end;
```

Since only one sorting operation is performed in every cycle, all the sorting operators, represented by S0, S1, S2, S3, S4, and S5 in Routine-8, can be assigned to the same sorting element.

{ S0, S1, S2, S3, S4, S5 }

Two five-to-one and two two-to-one multiplexers are needed. The five-to-one multiplexers are for routing { A0, C0, A1, B1, C1 } and { B0, D0, B1, C1, D1 } to the first and the second inputs of the sorting element, respectively. The two-to-one multiplexers are for routing the two outputs of the sorting element to the two registers B1 and C1, respectively. Below is the schematic of the resultant data paths. The controller of the data paths is specified by the state transition diagram depicted in Figure 4.

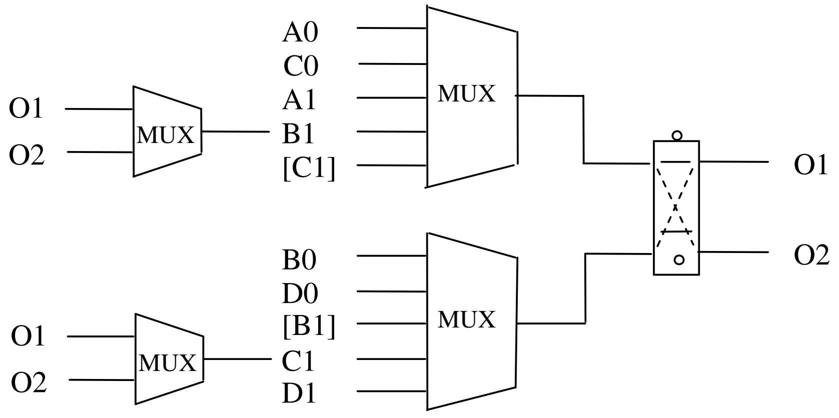


Figure 3: Data paths of Routine-8

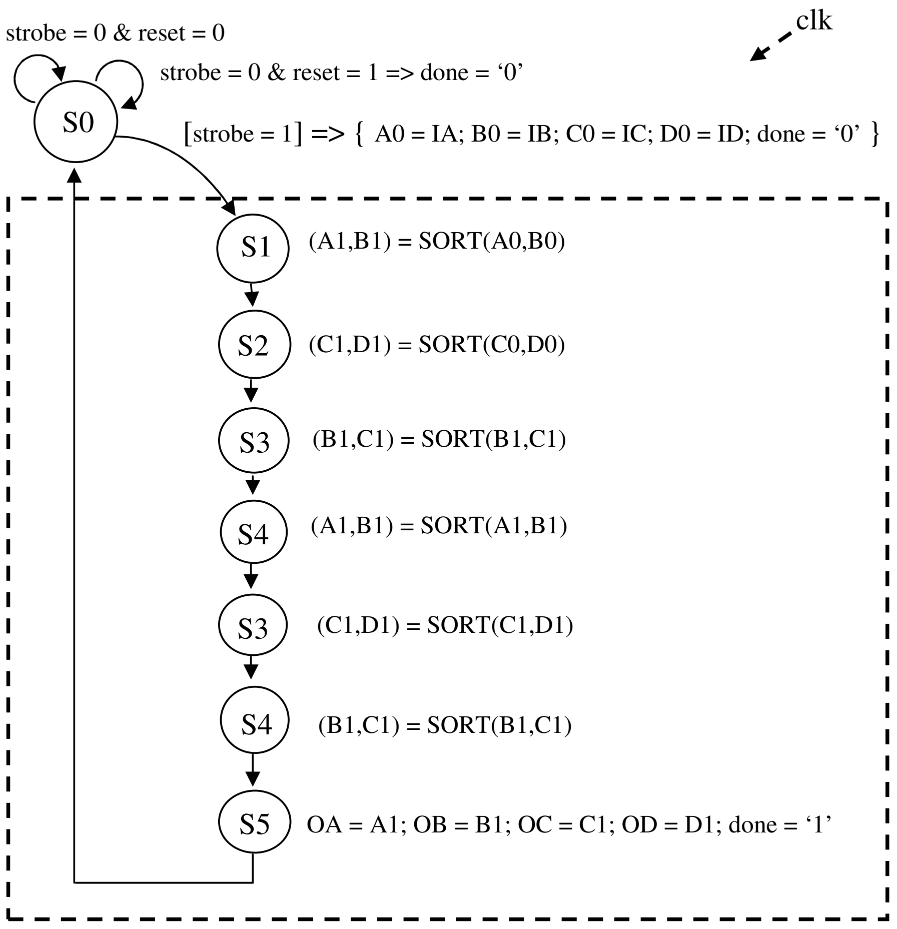


Figure 4: A finite-state-machine control specification for Routine-8

5.2 ASAP/ALAP Implementation.

Routine-9 contains the ASAP/ALAP schedule of the four-input sorting function defined in Routine-7.

Routine-9: The ASAP/ALAP dataflow description of four variables

```
if (strobe = '1') then
  begin
    A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;        done = '0';              -- I
    (A1, B1) = sort(A0, B0);        (C1, D1) = sort(C0, D0);               -- 1
    (B2, C2) = sort(B1, C1);                                               -- 2
    (A3, B3) = sort(A1, B2);        (C3, D3) = sort(C2, D1);               -- 3
```

```
      (B4, C4) = sort(B3, C3);                                     -- 4
      OA = A3;  OB = B4;  OC = C4;  OD = D3;   done = '1';         -- X
    end
  else
    begin
      if (reset = '1') then
        begin
          done <= '0';
        end;
    end;
```

To perform register allocation, the result of lifetime analysis for Routine-9 is given in Table 2.

Table 2: Lifetime analysis result for Routine-9

	A1	B1	C1	D1	B2	C2	A3	B3	C3	D3	B4	C4
Index	1	2	3	4	5	6	7	8	9	10	11	12
Step1	L	L	L	L	D	D	D	D	D	D	D	D
Step2	L	L	L	L	L	L	D	D	D	D	D	D
Step3	L	D	D	L	L	L	L	L	L	L	D	D
Step4	D	D	D	D	D	D	L	L	L	L	L	L
StepX	D	D	D	D	D	D	L	D	D	L	L	L

Based on the dataflow in Routine-9 and the lifetime analysis result in Table 2, an undirected graph identifying the compatible variable-pairs for sharing is given below.

```
                  (1, 7) (1, 8)                  (1,11) (1,12)
(2, 5)   (2, 6)   (2, 7) (2, 8) (2, 9)  (2,10)  (2,11) (2,12)
(3, 5)   (3, 6)   (3, 7) (3, 8) (3, 9)  (3,10)  (3,11) (3,12)
                                 (4, 9)  (4,10)  (4,11) (4,12)
                  (5, 7) (5, 8)                  (5,11) (5,12)
                                 (6, 9)  (6,10)  (6,11) (6,12)
                                                 (8,11) (8,12)
                                                 (9,11) (9,12)
```

Register allocation concludes that the twelve variables can be assigned to four registers. The assignments are listed below.

{ 2, 5, 8, 11 } or equivalently, { B1, B2, B3, B4 }
{ 3, 6, 9, 12 } or equivalently, { C1, C2, C3, C4 }
{ 1, 7 } or equivalently, {A1, A3 }
{ 4, 10 } or equivalently, { D1, D3 }

The new dataflow description is presented in Routine-10.

Routine-10: Resultant dataflow description of Routine-9 from register allocation

```
if (strobe = ‘1’) then
  begin
    A0 = IA;  B0 = IB;  C0 = IC;  D0 = ID;        done = ‘0’;      -- I
    (A1, B1) = sort(A0, B0);       (C1, D1) = sort(C0, D0);        -- 1     S0, S1
    (B1, C1) = sort(B1, C1);                                        -- 2     S2
    (A1, B1) = sort(A1, B1);       (C1, D1) = sort(C1, D1);        -- 3     S3, S4
    (B1, C1) = sort(B1, C1);                                        -- 4     S5
    OA = A1;  OB = B1;  OC = C1;  OD = D1;   done = ‘1’;      -- X
  end
else
  begin
    if (reset = ‘1’) then
      begin
        done <= ‘0’;
      end;
  end;
```

The sorting operators, represented by S0, S1, S2, S3, S4, and S5 in Routine-10, can be assigned to two sorting elements.

```
{ S0, S2, S3, S5 }      =>      sortX
{ S1, S4 }              =>      sortY
```

Below is a segment of the dataflow description showing which sorting element is used in each statement.

```
(A1, B1) = sortX(A0, B0);     (C1, D1) = sortY(C0, D0);     -- 1     S0, S1
(B1, C1) = sortX(B1, C1);                                   -- 2     S2
(A1, B1) = sortX(A1, B1);     (C1, D1) = sortY(C1, D1);     -- 3     S3, S4
(B1, C1) = sortX(B1, C1);                                   -- 4     S5
```

Two three-to-one and four two-to-one multiplexers are needed. The three-to-one multiplexers are for routing { A0, A1, B1 } and { B0, B1, C1 } to the first and the second inputs of the sorting element denoted by sortX, respectively. Two of the two-to-one multiplexers are for routing { C0, C1 } and { D0, D1 } to the two inputs of the sorting element denoted by sortY, respectively. The other pair of two-to-one multiplexers is for routing the outputs of the sorting elements to the two registers B1 and C1, respectively. Figure 5 depicts the schematic of the resultant data paths. The controller of the data paths is specified by the state transition diagram depicted in Figure 6.

A sorting element is a commutative operator; its input operands can be swapped without affecting the output. If the two input operands B1 and C1 in Step 2 and 4 of Routine-10 are swapped, the number of inputs to one of the three-input multiplexer can be reduced to two.

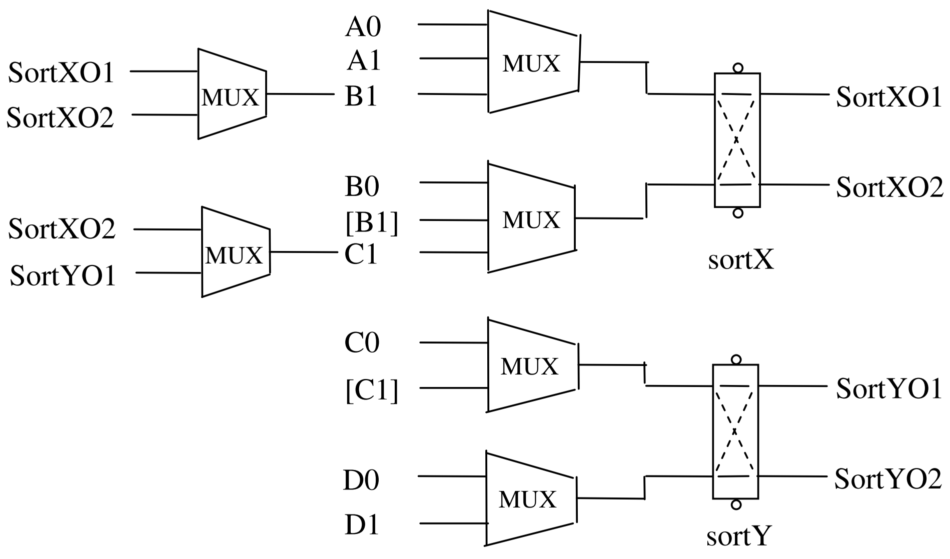
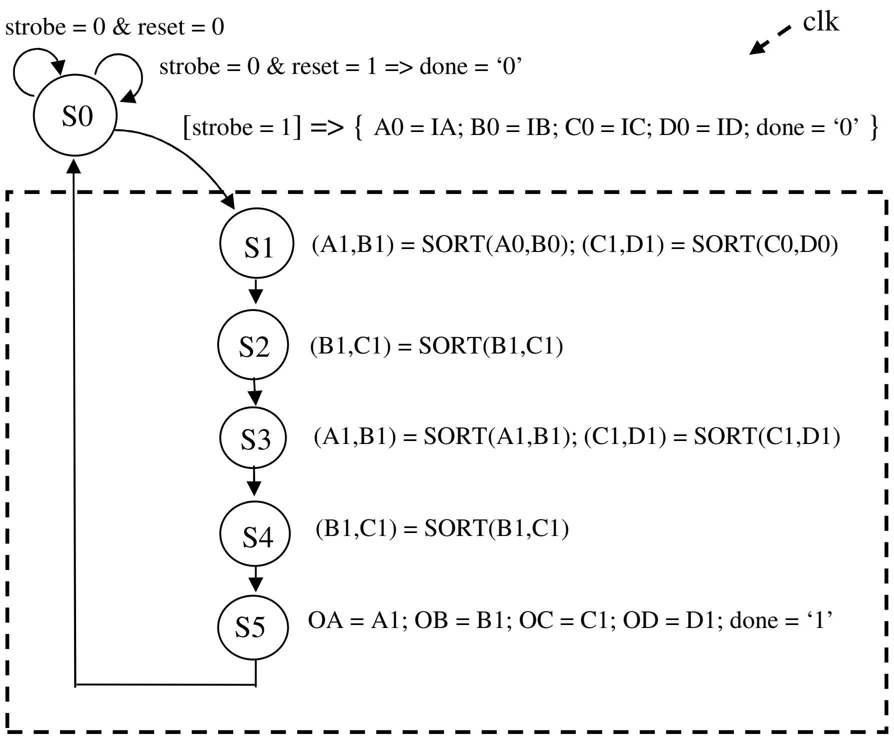


Figure 5: Data paths of Routine-10



strobe = 0 & reset = 0
strobe = 0 & reset = 1 => done = ‘0’
clk
S0
[strobe = 1] => { A0 = IA; B0 = IB; C0 = IC; D0 = ID; done = ‘0’ }
S1
(A1,B1) = SORT(A0,B0); (C1,D1) = SORT(C0,D0)
S2
(B1,C1) = SORT(B1,C1)
S3
(A1,B1) = SORT(A1,B1); (C1,D1) = SORT(C1,D1)
S4
(B1,C1) = SORT(B1,C1)
S5
OA = A1; OB = B1; OC = C1; OD = D1; done = ‘1’

Figure 6: A finite-state-machine control specification for Routine-10

5.3 Direct implementation of the ASAP/ALAP schedule

Instead of applying the clique-partitioning procedure to allocate registers, sorting elements, and multiplexers, as depicted in Figure 5 and Figure 6, the dataflow description in Routine-9 can be directly implemented. Figure 7 depicts the data-path schematic. The finite state transition diagram of the controller is depicted in Figure 8.

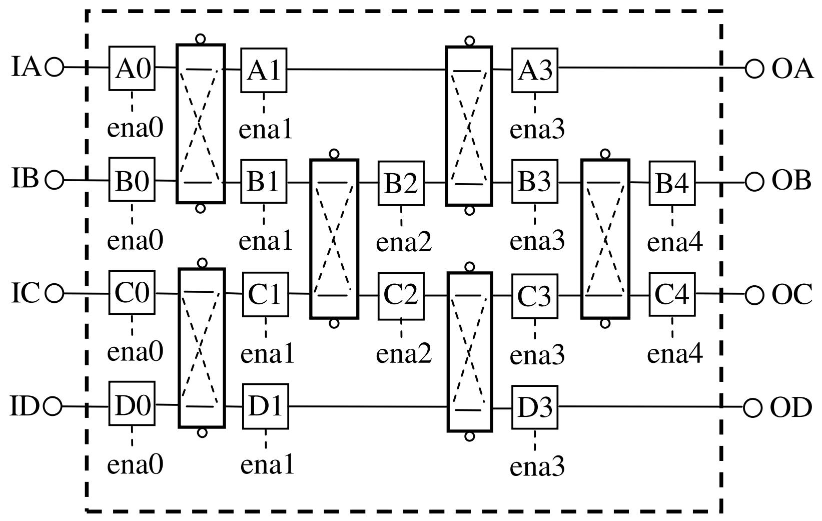


Figure 7: Direct Implementation Data paths of Routine-9

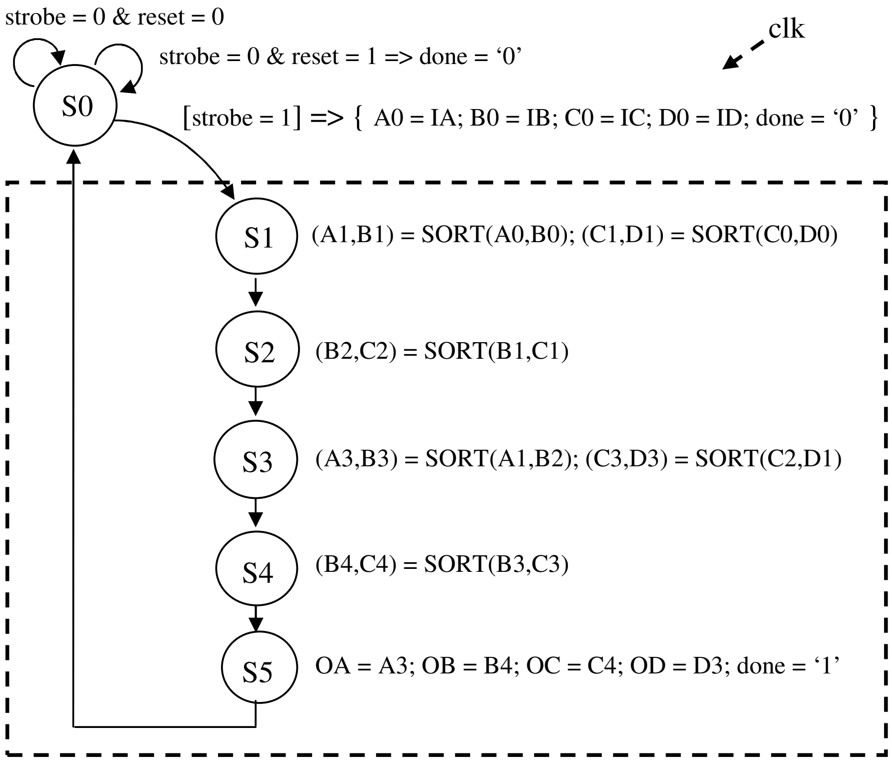

strobe = 0 & reset = 0
strobe = 0 & reset = 1 => done = ‘0’
clk
S0
[strobe = 1] => { A0 = IA; B0 = IB; C0 = IC; D0 = ID; done = ‘0’ }
S1
(A1,B1) = SORT(A0,B0); (C1,D1) = SORT(C0,D0)
S2
(B2,C2) = SORT(B1,C1)
S3
(A3,B3) = SORT(A1,B2); (C3,D3) = SORT(C2,D1)
S4
(B4,C4) = SORT(B3,C3)
S5
OA = A3; OB = B4; OC = C4; OD = D3; done = ‘1’

Figure 8: A control specification for the direct implementation of Routine-9

5.4 Implementation Data Summary

Sorting networks are data-path intensive designs; the circuit cost strongly depends on the width of data paths. If the input variables consists of a large number of bits, the cost of the finite state machine controller is insignificant and stays fairly constant. Table 3 summarizes the implementation data, where *N* is the number of bits contained in the input variables. Since we assume that there are only four input variables, the AEAP/ALAP schedule requires less area than the serial implementation. If the number of input data is large, then the serial implementation would be the most cost-effective one. The direct implementation requires a profound number of registers and sorting elements. Its implementation cost is generally the highest one.

Table 3: Hardware implementation data

Design ID	Serial	ASAP/ALAP	Direct
Number of states in controller	8	4	4
Number of flip-flops in data paths	8**N*	8**N*	16**N*
Equivalent number of 2-input multiplexers in data paths	10**N*	8**N*	0
Number of sorting elements	1	2	6

6. Sorting Specification with Separate Comparator and Data Switch

The sorting statement of (A1, B1) = sort(A0, B0) can be expanded as the following program segment.

```
if (A0 > B0) then
        begin
                A1 = B0;        B1 = A0;
        end
else
        begin
                A1 = A0;        B1 = B0;
        end;
```

The presence of the relational expression (A0 > B0) would prevent normal dataflow scheduling from obtaining a description with maximum parallelism. One method of addressing this issue is to identifying the output variable of each comparison as a net variable, instead of a register variable [10]. Generally speaking, whether to implement an operation in the data paths or controller is also an important consideration in digital system design.

7. Conclusion and Assessment

In this paper, we address the issues of exploring the design space of digital systems through dataflow scheduling. Several sorting dataflow specifications are presented. Given a dataflow specification, we discuss the issues in optimizing the number of registers, operators, and interconnections. The factors which affect the cost and performance of the resultant system are discussed. Also, we briefly examine the implication of separating and integrating dataflow and control flow on design space exploration.

Students analyzed and compared software algorithms and hardware dataflow descriptions. They also investigated the designs generated from various dataflow specifications. From this exercise, they learned the importance of writing efficient algorithms for generating high-quality digital designs and how to explore the system-level design tradeoffs.

Many students indicated that "Advanced Digital Design" was one of the most useful courses that they had taken during their college years. Some of them applied the skills to their senior design projects; they used the techniques to produce compact and working circuits such as a speaker identification system using an FPGA technology [11, 12]. Several of them had also applied the methodologies to the design of new mathematical transforms in digital signal processing for their Master theses and conference papers [4].

Acknowledgement

The author is very grateful to the anonymous reviewers for their valuable suggestions to improve the presentation of the paper.

References

1. Aho, V. and J. D. Ullman, "Principles of Compiler Design," Addison-Wesley, Reading, MA, 1977.

2. Peter J. Ashenden, The Designer's Guide to VHDL, 2002, Morgan Kaufmann Publishers, San Francisco, California 94104.

3. Bhasker, J., "A System C Primer," Star Galaxy Publishing, 2004.

4. Goodman, T. J. and Aburdene, M. F., "A Hardware Implementation of the Discrete Pascal Transform for Image Processing," IS&T/SPIE 18th Annual Symposium on Electronic Imaging, San Jose, California, January 2006.

5. Hoare, C.A.R., "Partition: Algorithm 63:," "Quicksort: Algorithm 64," and "Find: Algorithm 65." Communications of the ACM, 4 pp. 321-322, 1961.

6. Kerninghan, B. W. and Ritchie, D. M., The C Programming Language, 1978, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

7. Knuth, D., "The Art of Computer Programming," Vol. 3: Sorting and Searching," Addison-Wesley, 1997, Section 5.2.2: Sorting by Exchanging.

8. Paulin, P. G. and Knight, J. P., "Force Directed Scheduling for The Behavioral Synthesis of ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, pp. 661-679, Miami Beach, FL, July 1987.

9. Tseng, C. J. and Siewiorek, D. P."Automated Synthesis of Data Paths in Digital Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-5, No. 3, pp. 379-395,June 1989.

10. Tseng, C. J., Wei,R. S., Rothweiler, S. G., Tong, M. M. and Bose, A. K., "Bridge: A Versatile Behavioral Synthesis System," Proceedings of the 25th Design Automation Conference, Anaheim, California, June 1988.

11. Xilinx, “Xilinx Synthesis Technology (XST) User Guide,” California, 2002.

12. Xilinx, “Spartan-3^{TM} Development Board,” California, 2004.