Dynamic Pipeline Design of an Adaptive Binary Arithmetic Coder

Shiann Rong Kuang, Jer Min Jou, Ren Der Chen, and Yeu Horng Shiau

***Abstract*—Arithmetic coding is an attractive technique for lossless data compression but it tends to be slow. In this paper, a dynamic pipelined very large scale integration architecture with high performance for on-line adaptive binary arithmetic coding is presented. To obtain a high throughput pipelined architecture, we first analyze the computation flow of the coding algorithm and modify the operations whose data and/or control dependencies cause the difficulties in pipelining. Then, a novel technique called dynamic pipelining is developed to pipeline the coding process with variant (or run-time determined) pipeline latencies (or data initialization intervals) efficiently. As for data path design, a systematic design methodology of high level synthesis and a less-area but faster fixed-width multiplier are applied, which make the architecture with a little additional hardware. The dynamic pipelined architecture has been designed and simulated in Verilog HDL, and its layout has also been implemented with the 0.8-μm SPDM CMOS process and the ITRI-CCL cell library. Its simulated compression speeds under working frequencies of 25 and 50 MHz are about 6 and 12.5 Mb/s, respectively. About two times the speedup with 30% hardware overhead relative to the original sequential one is achieved.**

***Index Terms*—Arithmetic coding, data compression, dynamic pipeline.**

I. Introduction

LOSSLESS DATA compression, which can recover compressed data without any distortion, is a useful technique for many applications; one is for compressing source files in which any loss of information is not allowed, such as text files, executable files, and medical images. The other is for lossy image compression, in which lossless coding is a part of the whole coding algorithm, such as those algorithms specified by JPEG and MPEG. Arithmetic coding [1], [2] is an attractive technique for lossless data compression that permits eliminating redundancies in data sequences with a better efficiency than Huffman coding [8]. It permits coding a symbol with a noninteger number of bits, so that it reflects the amount of information provided by each symbol with better fidelity. It also presents the advantage that there is a clear separation between the encoding and the probability model. However, arithmetic coding tends to be slow, because in the simplest form it requires at least one multiplication per input symbol. Moreover, if an adaptive coding scheme is applied, an extra division may be needed at every coding cycle. Therefore, efficient hardware design for it to promote compression speed is essential for time-critical applications.

A CMOS realization of an arithmetic encoder/decoder for binary images has already been reported [3]. An updated version of the system, known as the Q-coder, appeared in [4] and [23]. Some architectural advances in the high speed very large scale integration (VLSI) implementation of arithmetic coders were accomplished by using loop unrolling and speculative execution [5]. Moreover, Fu and Parhi [6] proposed an algorithm that uses redundant arithmetic to obtain further speedup in the VLSI implementation of the QM-coder. However, all the Q-coder based arithmetic coding hardware described above are designed to compress mainly bilevel image data and may be poor for other types of data. It would be advantageous to have a compression chip universal enough to quickly compress any type of data (i.e., text, binary, and image files) that could still achieve a good compression ratio. In [7], we have proposed an adaptive division-free arithmetic coding algorithm (ADBAC) and an application specified integrated circuit (ASIC) chip for efficient lossless compression of universal data. It codes binary symbols "1" or "0" in any type file iteratively by three sequential phases: probability estimation, arithmetic operation, and the normalization loop. The probability estimation phase uses a new modeler that can efficiently estimate the conditional probability of the next symbol so that the algorithm can obtain high compression ratios for data of any type. A table-look-up method has been designed and applied to promote the chip's coding throughput. However, its compression speed is still slow and needs to be enhanced.

To design high speed and high throughput circuits, pipelining is one of the most efficient and economic techniques. For functional pipelining, consecutive iterations of a hardware loop are initiated at a time interval called latency, that is fixed [10]–[15] or has some fixed values [16]. However, in the main ADBAC processing loop, variant execution length of each iteration and time-relative data dependencies between them cause its pipeline latencies unfixed and hard to pipeline. In this paper, we present a new and efficient dynamic pipelined architecture with run-time determined latencies for ADBAC. We first identify and modify some complicated data and/or control dependencies in ADABC so that pipelining is easy to carry out, and then separately pipeline two parts: the normalization loop and the main loop. In the main loop portion, the inner normalization loop phase is viewed as an unbound delay operation whose delay is data- and iteration-dependent. The

Manuscript received November 3, 1999; revised August 20, 2001. This work was supported in part by the National Science Council, R.O.C., under Grant NSC-86-2221-E-006-022. This paper was recommended by Associate Editor R. Geiger.

S. R. Kuang is with the Department of Electronic Engineering, Southern Taiwan University of Technology, Tainan 70101, Taiwan, R.O.C.

J. M. Jou, R. D. Chen, and Y. H. Shiau are with the Department of Electrical Engineering, National Cheng Kung University, Tainan 70101, Taiwan, R.O.C. (e-mail: jou@j92a21.ee.ncku.edu.tw).

Publisher Item Identifier S 1057-7130(01)10418-0.

```
Encoding()
{
        C=0x00;    A=0xff;    R=0x0000;    AS=0000000000;
        for (each input binary symbol) {
                phase1: Generate P('0'|AS) by Eq.(1);
                phase2: AP=A* P('0'|AS);
                        if (input symbol=='0')   A=AP;
                        else {
                                A=A-AP;    C=C+AP;
                                if (carry occurs)    R++;
                        }
                        Update the adaptive modeler by Eq. (2);
                        Shift the input symbol into AS;
                phase3: while (MSB of A==0)   normalization_of_encoding();
        }
}

normalization_of_encoding()
{
        Shift MSB of R as output, shift MSB of C into R, and shift left A and C one bit;
        if (Output=='1') {
                if (C1num==15) {   Output two consecutive '0''s;    C1num++;   }
        }
        else   C1num=0;
        if (R==0xffff) {   Output two consecutive '1''s;   R=0xfffc;   }
}
```

(a)

```
Decoding()
{
        C and R is the encoded input;    A=0xff;    AS=0000000000;
        while (true) {
                phase1:  Generate P('0'|AS) by Eq. (1);
                phase2: AP=A* P('0'|AS);
                        if (C<AP) { A=AP;    '0' is decoded; }
                        else { A=A-AP;    C=C-AP;      '1' is decoded; }
                        Update the adaptive modeler by Eq. (2);
                        Shift the output symbol into AS;
                phase3: while (MSB of A==0)   normalization_of_decoding();
        }
}

normalization_of_decoding()
{
      Read input symbol;
      Shift left A and C one bit, shift MSB of R into C, and shift the input symbol into R;
      if (Intput=='1') {
              if (C1num==15) {
                      Read two stuffing bits "$b_1b_2$";
                      if ("$b_1b_2$"=="01") { R=0;   C++; }
                      if ("$b_1b_2$"=="1x")    End decoding;
              }
              C1num++;
      }
      else   C1num=0;
}
```

(b)

Fig. 1. (a) Adaptive binary arithmetic encoding algorithm. (b) Adaptive binary arithmetic decoding algorithm.

normalization and main loop portions are then serially processed by using ASAP-pipeline-scheduling [17], [18], and/or speculation techniques [19], [20], and finally are integrated into a dynamic pipeline. After dynamic pipelining, the pipeline latencies of ADBAC are dependent on the execution length of the normalization phase, and are naturally unfixed, and vary between 4–12. The average latency value is about 4 (obtained by hardware simulation). Thus, the compression speed of ADBAC can be significantly improved and is about two times speedup to its sequential version while still maintaining all the original functionality. While designing the pipelined ADBAC datapath, we applied the techniques of high level synthesis [17], [18] and a less area but faster fixed-width multiplier [21] to save hardware units, therefore, according to our experiment the dynamic pipelined architecture only increases about 30% area overhead with respect to the sequential architecture. The layout of the architecture has been implemented with Cadence tools [22] based on the 0.8-μm SPDM CMOS process and the ITRI-CCL cell library [9]. The simulated compression speed of it under working frequencies of 25 and 50 MHz are about 6 and 12.5 Mb/s, respectively.

The rest of this paper is organized as follows. Section II gives background information on the ADBAC algorithm and its sequential architecture. In Section III, we explain the de-

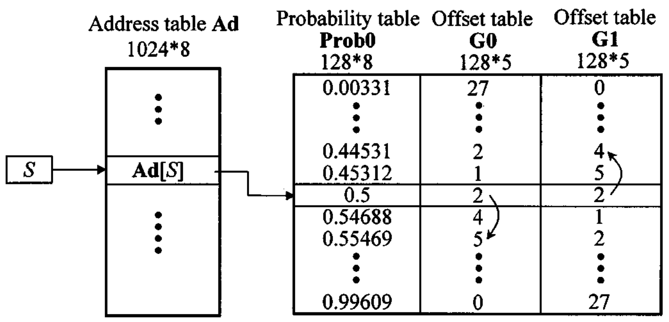


Fig. 2. Relation between tables $\mathbf{Ad}$, $\mathbf{Prob0}$, $\mathbf{G0}$, and $\mathbf{G1}$.

sign process of dynamic pipelining and propose the dynamic pipelined architecture of the algorithm. Section IV shows the simulation and the comparative results of the sequential and pipelined architecture. Finally, concluding remarks are made in Section V.

II. Overview of the ADBAC Algorithm and Its Sequential Architecture [7]

Arithmetic coding is one of the best statistical coding techniques. The encoding process of arithmetic coding begins with the open interval [0, 1] and subdivides it into subintervals. Each subinterval represents a unique source symbol, and the size of the interval is proportional to that symbol's probability of occurrence. For a given source symbol, the encoder locates the corresponding subinterval, and then divides this interval into subintervals whose ratios are the same as the original cumulative probabilities. The decoding process of arithmetic coding recovers the source symbols from the received interval using a procedure similar to that of the encoding process. Binary arithmetic coding deals with only two input symbols: "1" and "0." Therefore, the coding process will be simplified correspondingly and be easier to implement. The operations of the ADBAC algorithm proposed in [7] is presented in Fig. 1, which consists of three main phases: probability estimation (phase 1), arithmetic operation (phase 2), and the normalization loop (phase 3).

In Fig. 1, the probability estimation phase estimates the probabilities of binary symbols using the table-look-up approach. There are four tables used in the probability estimation phase: $\mathbf{Prob0}$, $\mathbf{G0}$, $\mathbf{G1}$, and $\mathbf{Ad}$. The probability table $\mathbf{Prob0}$ saves the probability values of occurring frequency for symbol "0" in the sources, which is used to approximate the conditional probability of symbol "0." The offset tables $\mathbf{G0}$ and $\mathbf{G1}$ store the distances between new probability and original probability in the probability table $\mathbf{Prob0}$ for the current input "0" and "1," respectively. Also, there are 2^{10} pointers in the address table $\mathbf{Ad}$ that point to the entries of $\mathbf{Prob0}$. A 10-b register named as S is used to record the condition of the previous 10 input symbols, called conditional states or contexts. The value of S is used as an index for the $\mathbf{Ad}$ table. The relation among tables $\mathbf{Ad}$, $\mathbf{Prob0}$, $\mathbf{G0}$, and $\mathbf{G1}$, as well as S, is shown in Fig. 2. Based on the above mechanism, the formulate to generate condition probability, $\mathrm{P}(\text{“0”}|S)$, of input symbol "0" given the previous input conditions recorded in the S register is

$$\mathrm{P}(\text{“0”}|S) = \mathbf{Prob0}[\mathbf{Ad}[S]]. \tag{1}$$

Once $\mathrm{P}(\text{“0”}|S)$ generated by the probability estimation phase is sent to the arithmetic operation phase, the $\mathbf{Ad}$ table must be updated by the following equation:

$$\mathbf{Ad}[S] = \begin{cases} \mathbf{Ad}[S] + \mathbf{G0}[\mathbf{Ad}[S]] & \text{if symbol “0” is coded} \\ \mathbf{Ad}[S] - \mathbf{G1}[\mathbf{Ad}[S]] & \text{if symbol “1” is coded.} \end{cases} \tag{2}$$

The arithmetic operation phase in Fig. 1 calculates the new code values. In it, 8-b registers A and C are used to keep track of the width of the subinterval and the left point of the subinterval, respectively. In the arithmetic operation phase of encoding for each new input symbol, the $AP = A * \mathrm{P}(\text{“0”}|S)$ first is calculated and then A and C are updated as follows:

$$\text{If symbol “0” is encoded: } A = AP \tag{3}$$

$$\text{If symbol “1” is encoded: } A = A - AP \quad C = C + AP. \tag{4}$$

Decoding is achieved by interpreting C as magnitude, by performing searching with magnitude comparison, and by taking the inverse recursion for C.

At each arithmetic operation phase, A holds the information capacity and C holds the code point. Because fixed precision registers are used, the multiplication results have to be maintained to a fixed number of bits by renormalizing A and C. The normalization phase in Fig. 1 normalizes A and C by shifting left, whenever the value of A is less than half of the initial coding range. Moreover, an additional 16-b register called R is required as the output buffer in the encoding process to solve the source-termination and carry-over problems simultaneously. Note that the number of iteration times of the normalization phase (loop) is register A's value dependent and then run time determined, which makes the execution length of ADBAC variant each iteration. Conventionally, we all assume that each iteration execution time of an ASIC loop is a constant and, therefore, pipeline latency is also a constant or has some fixed cyclic values. When we design the fixed latency pipeline for the ADBAC loop, then the largest latency value will be applied, and we shall obtain a slowest pipelined circuit or even a nonpipelined (sequential) circuit. How do we pipeline the circuit with variant-iteration-execution-time? If we can do that, then a higher performance dynamic pipelined circuit will be obtained; otherwise, only a slow or sequential circuit can be used. A simple concept example is given in Fig. 3. From it, we can find that dynamic pipeline has higher performance than conventional fixed latencies pipeline and thus the sequential (or nonpipelined) one.

Fig. 4 shows the functional block diagram of the sequential arithmetic coding architecture [7]. The architecture, which contains an adaptive probability estimation modeler, an arithmetic operation unit, and an execution-time-unfixed normalization unit, performs the operations of encoding and decoding sequentially. The 10-b shift register S records the 10 previous input symbols. The adaptive probability estimation modeler

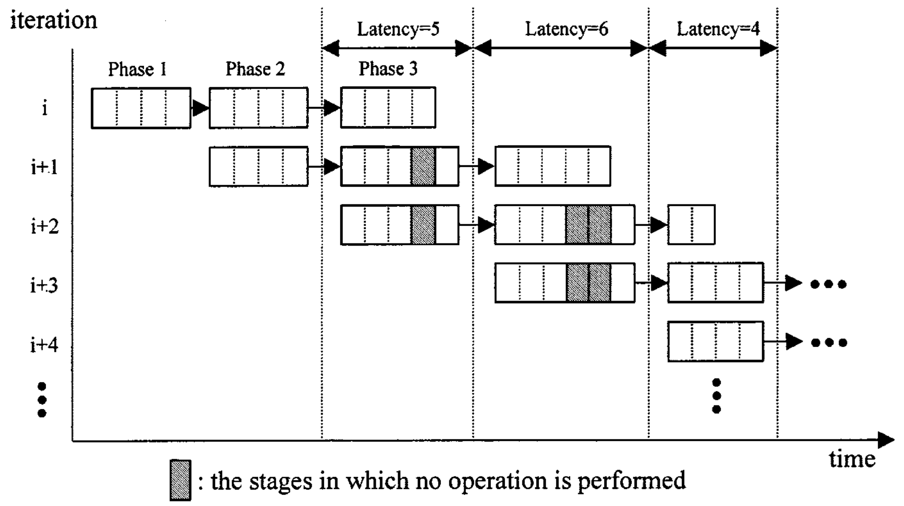


Fig. 3. An example of dynamic pipeline schedule.

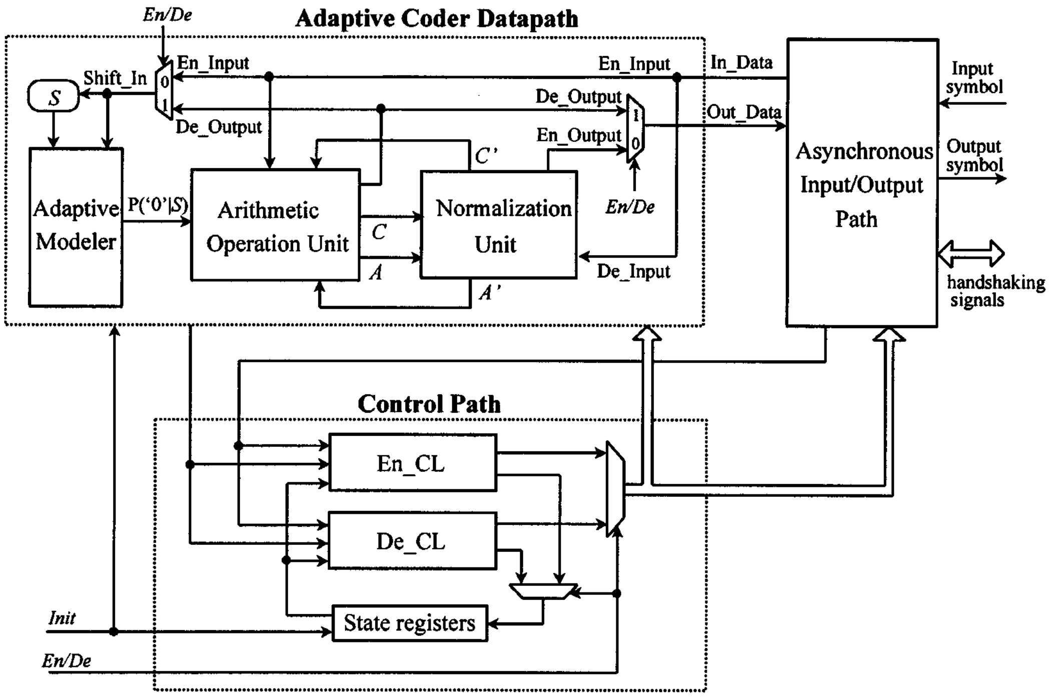


Fig. 4. Functional block diagram of the sequential arithmetic coding architecture.

generates the condition probability P(“0”$|S$) of symbol “0” given the previous input condition recorded in register S. Once the probability P(“0”$|S$) is obtained from the adaptive modeler, the arithmetic operation unit uses it to calculate AP as well as the new values of A and C, which are successively sent to the normalization unit to be normalized if necessary. Moreover, the normalization unit will send the encoding result to I/O Path or receive the input symbols to be decoded from I/O Path. For more details, please refer to [7].

III. Dynamic Pipeline Design of the ADBAC Algorithm

The architecture presented in Fig. 4 performs the ADBAC algorithm sequentially, and the compression speed is limited by the recursive dependence of the three phases and is slow. Increasing its speed becomes a pressing problem. Traditionally, the sequential architecture can have its performance efficiently improved by using the technique of pipelining with a (or some) fixed latency (latencies). However, since its normalization phase has data dependent number of iterations and the restrictions of data and/or control dependencies between different iterations of the loop, the ADBAC algorithm is impossible to pipeline conventionally with some fixed latencies. In addition, ADBAC also contains data dependent branches, which cannot be sped up very much without using the concepts similar to ASAP-pipelining scheduling [17], [18] and speculative computation [19], [20]. The dynamic pipelined architecture using the variant pipeline latencies scheme and the technique similar to ASAP-speculative scheduling must be developed to perform ADBAC efficiently.

```
Encoding()
{
    C=0x00;   A=0xff;   R=0x0000;   AS=0000000000;                ....... 1
    for (input binary symbol ≠ NULL) {                            ....... 2
        phase1:  radd=Ad[AS];                                     ....... 3
                 P('0'|AS)=Prob[radd];                            ....... 4
                 PR0=G0[radd];   PR1=G1[radd];                    ....... 5
        phase2:  AP=A* P('0'|AS);                                 ....... 6
                 if (input symbol=='0') {                         ....... 7
                     temp= radd+PR0;                              ....... 8
                     A=AP;                                        ....... 9
                 }
                 else {
                     temp= radd-PR1;                              ....... 10
                     A=A-AP;                                      ....... 11
                     C=C+AP;                                      ....... 12
                     if (carry occurs)   R++;                     ....... 13
                 }
                 Ad[AS]=temp;                                     ....... 14
                 Shift the input symbol into AS;                  ....... 15
        phase3:  while (MSB of A==0) {                            ....... 16
                     Shift MSB of R as output;                    ....... 17
                     Shift MSB of C into R;                       ....... 18
                     Shift left A and C one bit;                  ....... 19
                     if (output=='1') {                           ....... 20
                         if (C1num==15) {                         ....... 21
                             Output two consecutive '0"s;         ....... 22
                             C1num++;                             ....... 23
                         }
                     }
                     else   C1num=0;                              ....... 24
                     if (R==0xffff) {                             ....... 25
                         Output two consecutive '1"s;             ....... 26
                         R=0xfffc;                                ....... 27
                     }
                 }
    }
}
```

Fig. 5. Hardware behavioral description of the encoding algorithm.

A. Dynamic Pipelining ADBAC

To efficiently design the dynamic pipelined encoding architecture, ADBAC is first modeled as a hardwared behavioral description. Next, we analyze the restrictions and modify corresponding operations in the description that prevent the pipelining execution of ADBAC, and then use the approach similar to ASAP-speculative scheduling to speed up the execution. Finally, a dynamic pipelined architecture (including a datapath and a special controller) of coding with variant latencies is designed. The following subsections explain these processes.

To begin designing the dynamic pipelined architecture for ADBAC, the algorithm in Fig. 1(a) is first transformed into a hardware behavioral description as shown in Fig. 5. In it, the operations of addition, subtraction, multiplication, comparison, up-count, shift-left and assignment are denoted as $+$, $-$, $*$, $==$, $++$, $\ll$, and $\leftarrow$, respectively. Moreover, the operations in line i have added to them a postfix i. For example, the addition operation in line 8 of Fig. 5 is denoted as $+8$. The operations of multiplication and reading data from SRAM need two clock cycles, respectively, and the other operations like addition, subtraction, or comparison need one clock cycle, respectively, by using the 0.8-μm SPDM CMOS cell library [9] and one clock cycle is 15 ns in length. The control condition produced by the comparison operation in line i is denoted as ci. If condition $c2$ ($c16$) is true, the ***for (while)*** loop in Fig. 5 will continue to execute; otherwise, the loop will terminate its execution.

1) Operation Modifying: Pipelining Fig. 5 directly is difficult since 1) the execution length of the normalization (while) loop is unknown (run-time determined) and unpredictable, and, 2) there are many complicated data and/or control dependencies in it. We solve the less difficult 2) first, and then propose the dynamic pipeline approach to the dominant and harder 1) in the next subsection. We first incrementally unwind the main loop of Fig. 5 (see Fig. 6). In the unwound pipelining, the original location of phase 3 (the normalization phase) in it is regarded as an unbound delay operation whose delay is dependent on the result generated by comparison operation $==16$. Fig. 6 shows the pipelining pattern by unwinding the main loop of Fig. 5 three times and overlapping them, the smallest latency of the pipeline of it is equal to 4 since the SRAM that stores the content of table $\mathbf{Ad}$ cannot be read and written simultaneously. However, the pipeline in Fig. 6 cannot be correctly executed due to some restrictions that are caused by the complicated data and/or control dependencies among three phases. In the following, we discuss these restrictions that cause incorrect pipelining, and then present the solutions for them. To distinguish the operations and data values in different iterations of Fig. 5, the operations and data values in iteration j is attached a subscript j.

The first restriction is the data dependencies caused by A and C between the unbound delay operation of iteration *i* (i.e., the operation $\ll 19_i$ in phase 3 of Fig. 5) and operations $*6_{i+1}$ and $+12_{i+1}$ which restrict their concurrent executions. This restriction can be loosened by detecting in advance the number of bits,

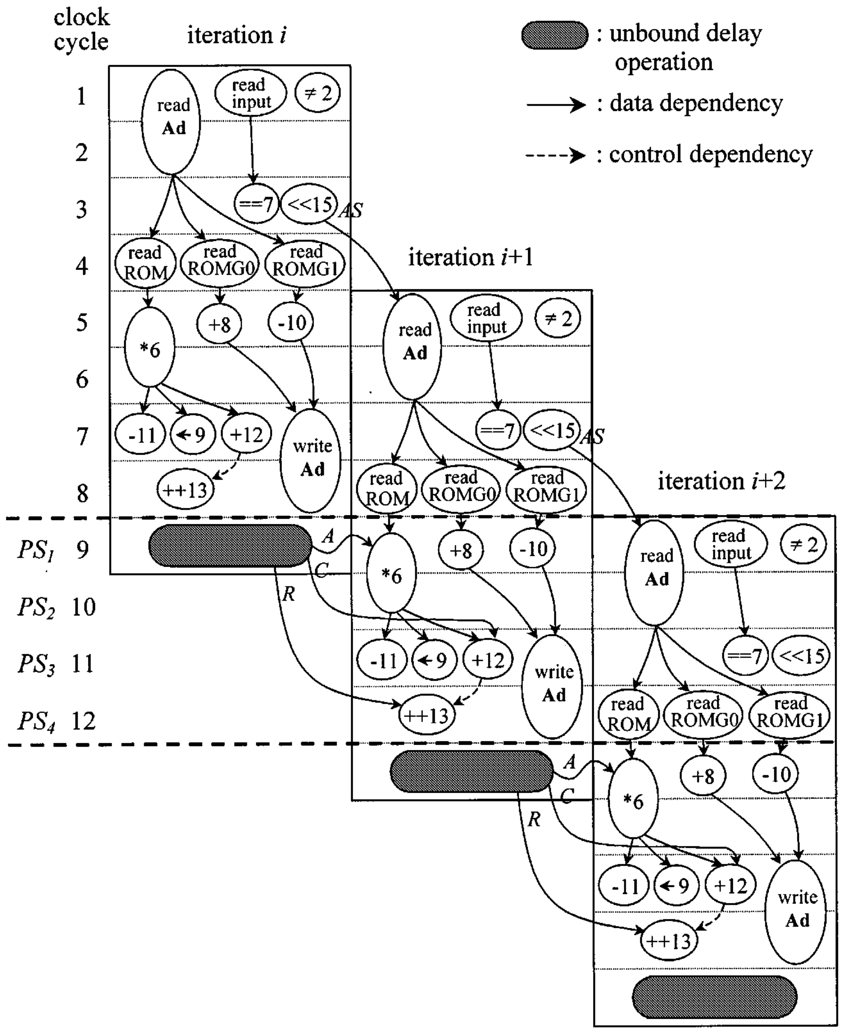


Fig. 6. Three times unwinding of the main loop of Fig. 5.

denoted as *numb*, needed to be normalized (shifted left) for both A_i and C_i from the result of A_i produced by operation ←9 or −11 at clock cycle 7, and then uses barrel shifters to accomplish the normalization task at clock cycle 8. A new operation Fir1(A) which produces *numb* is added at cycle 8, and two barrel shifting operations $\lll$33 and $\lll$34 used to shift A and C left are also added. The operation $numb > 0$, denoted as $>$ 35, now replaces the operation ==16 in the tester of the ***while*** loop to judge whether or not the loop terminates its execution. Since the old value of C before normalizing will be used in lines 18 and 19 of Fig. 5, it must be copied to another register, $C2$, in advance by the operation denoted as ←32. Then, register C in line 18 and 19 of Fig. 5 must be replaced by register $C2$ to avoid data errors and these modified operations are denoted as $\ll$36 and $\ll$37, respectively.

Second, the address value AS_{i+1}, which is produced by $\lll 15_i$ and is used as the reading address of table $\mathbf{Ad}$ in iteration $i+1$, must be produced before the operation of reading table $\mathbf{Ad}$ in iteration $i+1$. Therefore, the operation $\lll 15_i$ must be scheduled before clock cycle 5. However, the operation of writing table $\mathbf{Ad}$ in iteration i uses the old AS, AS_i, as its address. Thus, the old AS_i value must be copied to another register $AS2$ (before new AS_{i+1} is produced) to provide the address for the operation of writing table $\mathbf{Ad}$ at clock cycle 7 of iteration i, and this copy operation is denoted as ←30.

Third, the dependency of the data in table $\mathbf{Ad}$ between the operation of writing $\mathbf{Ad}$ at clock cycle 7 of iteration i, and the operation of reading $\mathbf{Ad}$ at clock cycle 5 of iteration $i+1$ must be resolved. Since the reading $\mathbf{Ad}$ operation of iteration $i+1$ is scheduled before the writing $\mathbf{Ad}$ operation of iteration i, if their addresses are the same, the read $\mathbf{Ad}$ operation will get an error data. To avoid this error, we add a checking operation denoted as ==31 which checks the reading address, AS_{i+1}, and the writing address, AS_i, and schedule it at clock cycle 5. Then, a simple circuit to accompany the checking operation is designed; it will replace the reading result_{i+1} with the written result_i directly, if the checked addresses are the same.

After all modifications described above are made, the possibility of pipelining ADBAC increases much more, however, it still cannot be pipelined without resolving the dominant and harder (1), which will be overcome by newly developed dynamic pipelining described in the next subsection.

2) Dynamic Pipelining of ADBAC: Pipelining a circuit with some fixed latencies is easy [10], [11], but ADBAC cannot be pipelined with this type of design due to the normalization phase with a data dependent number of iterations. We analyze the situ-

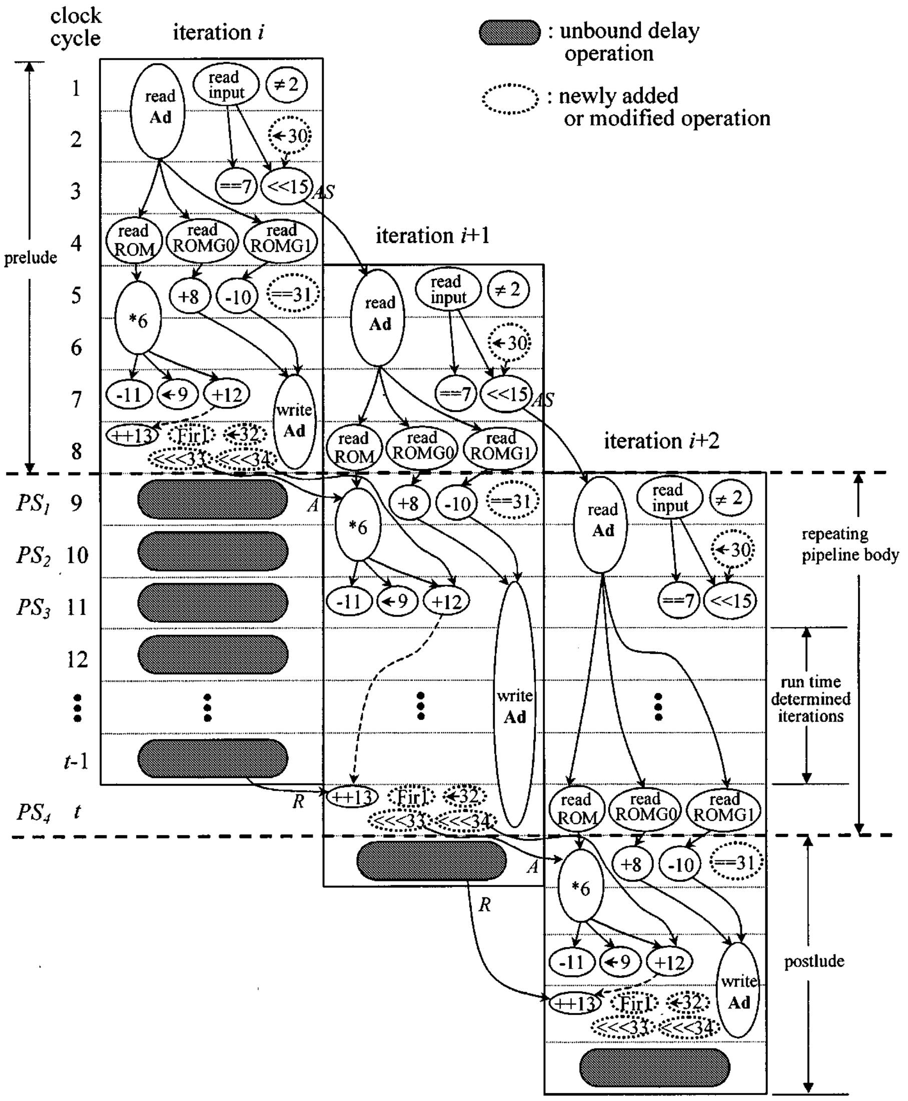


Fig. 7. Modified pipeline scheduling of ADBAC.

ations, and propose the novel dynamic pipeline scheme to overcome it as follows. Observing Figs. 5 and 6, value R_i produced by the unbound delay operation at clock cycle 9 of iteration i is consumed by operation $++13_{i+1}$ at clock cycle 12. This means that the unbound delay operation of iteration i must be completed within 3 clock cycles to satisfy the data dependency of R. Assuming that all of the unbound delay operation in each iteration can complete their operations within 3 clock cycles, an ADBAC pipeline with a latency equal to 4 can be obtained. As mentioned before, however, the execution length of the unbound delay operation varies. Once the execution length w of the unbound delay operation of iteration i is larger than 3, the operations at and after clock cycle 12, in all iterations following iteration i, must be delayed $w-3$ clock cycles. In such a pipelining execution, the latency is variant and determined at run time by the execution length w of the unbound delay operation. If $w \leq 3$, the latency is 4. Otherwise, the latency is $w+1$. We call the pipeline with variant latencies a dynamic pipeline. The implementing of this kind of pipeline requires a special control path, which consists of 2 interacting finite state machines (FSMs). Its design and description can be found in Section III-A-6.

After the design and modifications described above are finished, the modified pipeline scheduling of ADBAC is generated and shown in Fig. 7. The dynamic pipeline pattern in Fig. 7 is now correct, the repeating pipeline body in it is formed and found; all operations of different iterations which are executed at the same time are formed a state such as $PS_1 \sim PS_4$ as shown in Fig. 7. In the repeating pipeline body, because some states with only unbound delay operations may exist, *that makes the pipeline's latencies unfixed and run-time determined.* Based on Figs. 7 and 5, the initial state transition graph STG_m will be derived. Since the execution length of the unbound delay operation is unknown, a null state $Noop_n$, which represents the execution of the normalization phase and will be extended into another STG_n described in the next section, is added into STG_m between the state PS_3 and its child state PS_4 (see Fig. 8). Two control signals *down* and $start$ are also added for interacting communications between STG_m and STG_n, which will be described in Section III-A-4. Integration of the two STGs by the signals to design a dynamic pipeline controller will also be described. Now, the initial STG_m has been derived and shown in Fig. 8. Note that the operations at states $PS_1 \sim PS_3$ of STG_m

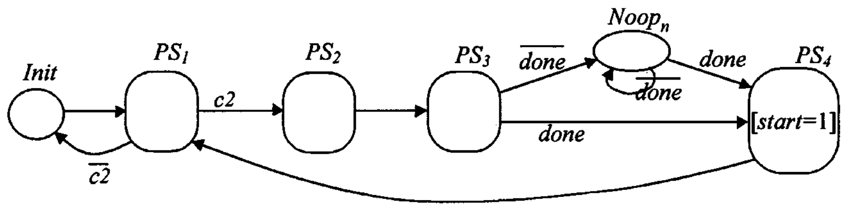


Fig. 8. Initial STG_m.

may be executed concurrently with the operations in STG_n, and thus, they are called *concurrent states*.

3) Speed Optimization of the Normalization Phase: Section III-A-2 has explained that the execution length w of the normalization phase will influence the pipeline latency as well as the performance of the dynamic pipeline design, consequently, how to reduce the execution length w is a very important task. However, the normalization phase cannot be pipelined, because it can only outputs at most one datum at each clock cycle, and pipelining it will output more than one datum per cycle. Therefore, we employ the concepts similar to ASAP scheduling [17], [18] and speculative computation [19], [20] to optimize the speed of the normalization phase and to reduce w.

In order to obtain the ASAP-speculative like scheduling of the normalization phase, each comparison operation in the ***if*** instructions of it may be modified with another comparison operation which will use the most early but correct data in its computation to accelerate the execution speed. The first modification is that the comparison operation ==20 (i.e., $output ==$ "1") is replaced by the new operation: the most significant bit (MSB) of $R == 1$ also denoted as ==20. Additionally, the comparison operation in line 25 of Fig. 5 (i.e., $R == 0xffff$) is modified into the new operation: $R == 0x7fff$ *and MSB of* $C2$ ==1 also denoted as ==25. After the modifications, the data dependency of value R (output) between operation $\ll$36 ($\ll$17) and old operation ==25 (==20) is removed, that makes the execution more rapid, and the new operation of $R == 0x7fff$ *and MSB of* $C2 == 1$ (also denoted as == 25) is scheduled at state S_1 (see Fig. 9). Applying the concept similar to speculative computation, which schedules the condition operations with their dependent comparison operations (such as operation $\ll$17 and operation >35) at the same state to promote the execution speed, the comparison operations >35, ==21, and the new operations ==20 are also scheduled at state S_1.

Moreover, with the same concept we can schedule the original and modified conditional operations in line 17, line 18 (which has been modified and is denoted as $\ll$36), and line 19 (denoted as $\ll$37) of Fig. 5, as well as the operation: $numb$-- (denoted as --38) derived in Section III-A-1 at state S_1, and determine whether or not these conditional results are stored into registers at the ending of state S_1 according to control condition $c35$ produced by operation >35. Now, initial STG_n has been derived and shown in Fig. 9, which will be used to construct the final STG_n and to design one of two interacting controllers for dynamic pipeline control. However, how to prevent these conditional operations at state S_1, whose executions depend on the result of logical operation >35, overrunning (when its executing condition is false) becomes a pressing problem. To solve

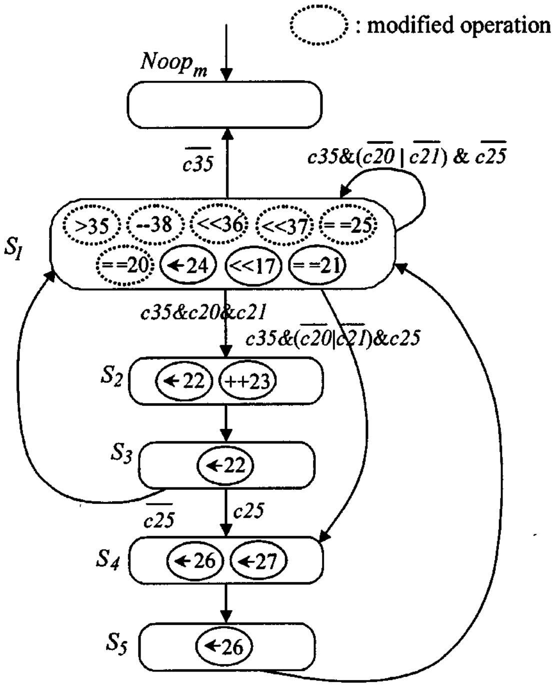


Fig. 9. State transition graph STG_n.

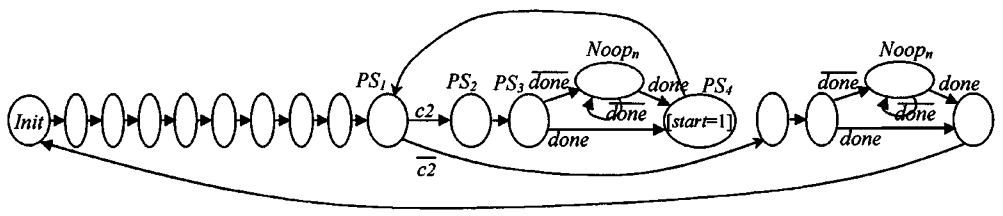
the problem, some new control signals will be generated and relative circuits will be designed. Let L_O, SL_R, SL_C2, and C_numb be the original loading, shifting, or down-counting control signals of registers: $output$, R, $C2$, and counter: $numb$, respectively. Signals L_O, SL_R, SL_C2, and C_numb are enabled and set to 1 by the controller when the circuit is at state S_1. To prevent the speculative overruns from occurring, some new control signals are designed to replace them to control respective registers or counters. Let their corresponding new control signals be L_O', SL_R', SL_C2', and C_numb', respectively. In addition, let signal In_1 be a control signal that indicates whether the circuit is at state S_1 and is set to 1 if it is at state S_1; otherwise, signal In_1 is set to 0. Then, signal L_O' is generated by

$$L\_O' = \begin{cases} L\_O, & \text{if } In_1 = 0 \\ c21, & \text{otherwise.} \end{cases} \tag{5}$$

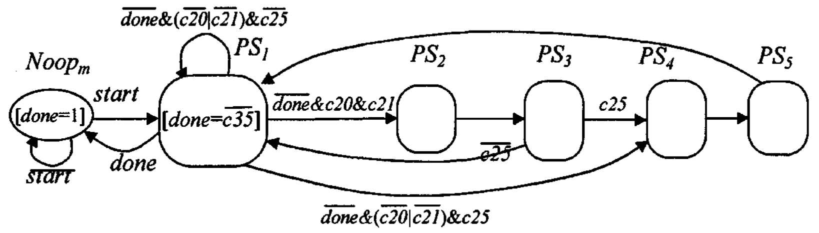
Signals SL_R', SL_C2', and C_numb' can be generated by the similar way as signal L_O'.

By analyzing the execution condition, the STG_n of the normalization phase shown in Fig. 9 will complete its operations using 1 clock cycle in the fastest case or using 11 clock cycles in the worst case.

4) Final Interacting STG's Generation: Before designing the dynamic pipeline controller, initial STG_m and initial



(a)



(b)

Fig. 10. (a) Final STG_m. (b) Final STG_n.

	*	+	−	<<<	Fir1	==
PS_1	X	X	X			X
PS_2	X					
PS_3		X	X			
PS_4				X	X	

Fig. 11. Activated times of operations in the repeating loop body.

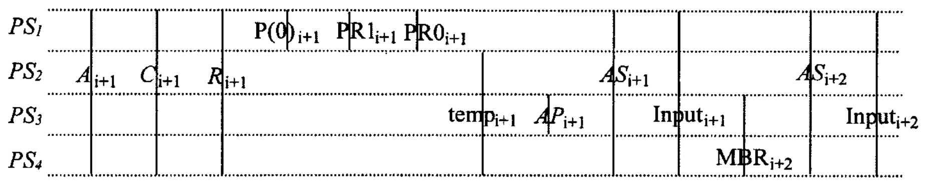


Fig. 12. Lifetimes of data values in the repeating pipeline body.

STG_n must be appropriately modified and integrated to form two interacting STGs for dynamic pipeline control. Two interacting signals $start$ and $done$ are added between them for their interacting communications. In STG_m, the null state $Noop_n$ has added to it two outgoing edges labeled with signal condition $done$ to state PS_4 and $\overline{done}$ to itself, respectively. Signal $done$ equal to $\overline{c16}$ is enabled (disabled), i.e., set to 1(0), by STG_n to represent the finish (unfinish) of the normalization phase's work and to inform STG_m the condition, and the prelude and postlude of the dynamic pipeline are also inserted into STG_m. Moreover, another signal $start$ must be added and be set to 1 at state PS_4 of STG_m to initiate the execution of STG_n, and is disabled (set to 0) at other states of STG_m. In STG_n, on the other hand, another null state $Noop_m$ that represents the operating of STG_m is also added two outgoing edges labeled with signal conditions $start$ to the next state PS_1 and $\overline{start}$ to itself, respectively. The final STG_m and final STG_n are shown in Fig. 10(a) and (b), respectively.

5) Data Path Design: Subsequently, the datapath allocation then is performed according to Fig. 5, STG_m and STG_n to construct the dynamic pipelined datapath. The datapath allocation techniques proposed in [17], [18] are applied to design the datapath by assigning each operation to a functional unit, by assigning each data value to a storage element, and by providing interconnections among functional units and storage elements using multiplexers and/or buses. The storage elements are allocated by a lifetime analysis of data values, and functional units are allocated by the activated time of operations. In general, any hardware unit can be shared if all allocation processes about it don't generate any time conflict. In the allocation process, efforts are made to allow maximal sharing of functional units and storage elements. However, sometimes resource sharing is not performed when the relative multiplexing (or busing) and interconnection cost exceeds the shared gain. Additionally, to save the circuit cost, a less-area but faster fixed-width multiplier [21] is developed and is applied to implement all multiplication operations.

The activated times of operations in the repeating loop body of Fig. 7 are shown in Fig. 11. Based on Fig. 11, the functional units allocated contain one multiplier, one adder, one subtractor, two-barrel shifters, one Fir1 to produce $numb$, and one comparator. On the other hand, the lifetimes of data values in the repeating pipeline body are shown in Figs. 12 and 10 registers or counter-based registers are allocated. After the functional units

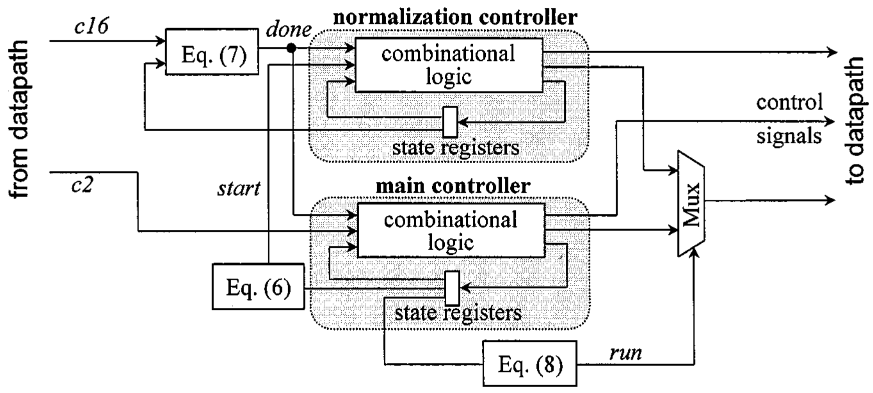


Fig. 13. Basic architecture of the controller.

and registers have been allocated, the interconnection among them is constructed by tracing the execution of the operations in Fig. 10.

6) Dynamic Pipeline Controller Design: A controller is designed to sequence datapath hardware units according to the dynamic pipelined scheduling as represented in final STG_m and STG_n. The controller consists of two parts: the main and normalization controllers. The main (normalization) controller is derived from STG_m (STG_n) with the sequential circuit design techniques, and is optimized with logical synthesis tools. The interacting signals $start$ and $done$ described previously are applied to concert the activations of the two controllers. The signal $start$ used to initiate the execution of STG_n is enabled by the main controller when it is at the state PS_4. More formally

$$start = \begin{cases} 1, & \text{if } STG_m \text{ is at state } PS_4 \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

When signal $start$ is set to 1, it means that the main controller will go into a concurrent state and then state $Noop_n$, and the control over datapath operation will be transferred from the main controller to the normalization controller. The signal $done$ is enabled by STG_n when it reaches state $Noop_m$ or the state S_1 producing $c35$ and $c35 = 0$. More formally

$$done = \begin{cases} 1, & \text{if } STG_n \text{ is at state } Noop_m \\ & \text{or at state } S_1 \text{ and } c35 = 0 \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

When signal $done$ is set to 1, it means that the STG_n has finished its operations and will go into state $Noop_m$. Meanwhile, the control over datapath operation will be transferred to the main controller.

Since the datapath is controlled by the main and normalization controllers simultaneously sometimes, both of them may send different control signals to the same hardware unit of datapath at the same time. The phenomena are called control conflicts. In the design, registers (or counters) $numb$, $C2$, and R have control conflicts. For one register X with control conflicts, a multiplexer is used to overcome the problems by multiplexing the control signals from the main or normalization controllers. Assume that the multiplexer uses signal run as its select signal. When run is 0 (1), the control signals from the main (normalization) controller are selected to control the unit X. Then, signal run will be enabled by the main controller when it is at state $Noop_n$, since the normalization controller has the control over datapath operation at that time. In addition, run is also enabled by the main controller when it reaches one of the sets of the concurrent states Δ at which the register X is written by the operations in STG_n. That is, signal run for solving the control conflict of register X, denoted as $run(X)$, is generated by

$$run(X) = \begin{cases} 1, & \text{if } STG_m \text{ is at state } Noop_n \text{ or at } \Delta \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

Fig. 13 shows the basic architecture of the controller.

B. Dynamic Pipeline Decoding

For the decoding part, its pipelining execution can be similarly designed to that of encoding. However, another new design problem occurs in decoding: the output value of iteration i, denoted as output$_i$, which is used to determine the AS_{i+1}, is produced after the reading $\mathbf{Ad}$ operation of iteration $i+1$. Splitting the SRAM that stores the content of table $\mathbf{Ad}$ into two subblocks as shown in Fig. 14 can solve this problem. In the SRAM structure of Fig. 14, based on the fact of AS_{i+1} being obtained by shifting the output$_i$ into AS_i, the possible $\mathbf{Ad}[AS_{i+1}]$ can be read by $AS_i\langle 8 : 0\rangle$ from SRAM0 and SRAM1 before the output$_i$ is generated. The actual $\mathbf{Ad}[AS_{i+1}]$ then is selected by output$_i$. Subsequently, the dynamic pipelining technique introduced in Section III-A can be used to pipeline the execution of the three phases of the decoding process.

IV. Experimental Results

We have designed and implemented both of the proposed dynamic pipelined and the sequential [7] architectures of the ADBAC algorithm in Verilog HDL, and achieved their layouts using Cadence tools [22] with the standard cells of 0.8-μm SPDM technology [9], respectively. The brief layout design flow is as follows. First, we use the Preview tool in OPUS [22] to floorplan blocks in the design; then, Preview's Block Ensemble and Cell Ensemble are used to perform placement and routing; finally, we use Dracula [22] to verify the layout. A prototype chip of the sequential architecture has been implemented and fabricated [7], and the chip occupies a silicon area of 4.2 $*$ 4.5 mm^2. The compression speed of it under 25 MHz clock rate is about 3 Mb/s. On the other hand, the chip layout of the dynamic pipelined architecture occupies a silicon area of 4.5 $*$ 5.0 mm^2 and is shown in Fig. 15, but which was not fabricated due to chip turnaround time and cost. The

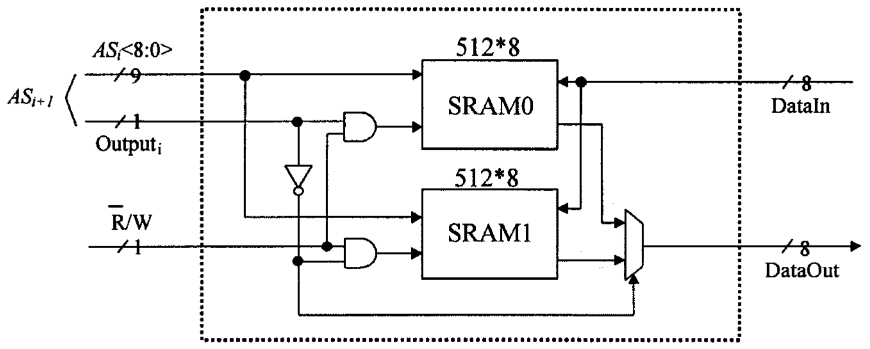


Fig. 14. Structure of SRAM for dynamic pipelining of decoding.

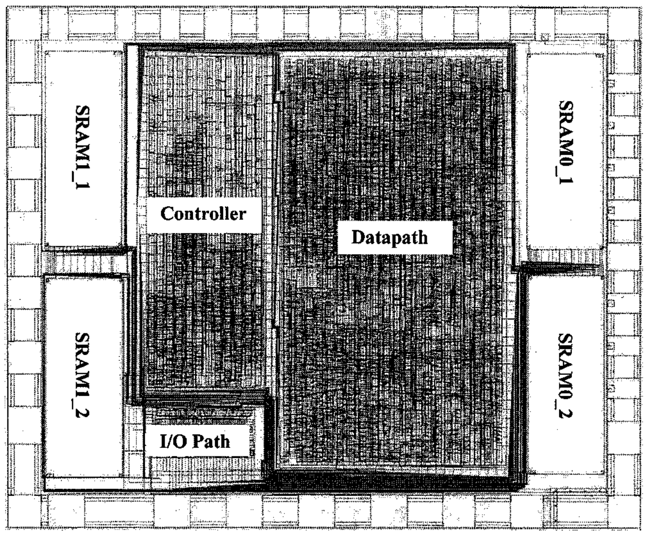


Fig. 15. Layout of the dynamic pipelined architecture of arithmetic coding.

main functional blocks of it containing four SRAMs, Datapath, Controller, and I/O Path, are also labeled, and they occupy about 31%, 46%, 20%, and 3% of the area of the chip layout, respectively. In the design, the dynamic pipelined architecture needs more hardware resources including one extra Fir1 unit, two special barrel shifters, more pipelined registers, and more complex interconnection. In addition, the control path also is complex because that it contains two controllers: a normalization controller and a main controller to achieve dynamic pipelining execution.

The simulation results of the dynamic pipelined architecture show that the normalization phase is frequently completed within 3 clock cycles and, therefore, the latency is 4 frequently. In practice, more than 90% of the normalization loop phases can be completed within 3 clock cycles so that the real compression speed is very close to the ideal compression speed. Fig. 16 shows the percentage distributions of the values of *numbs*, which is the number of iterations of the normalization loop executed in each coding cycle. The simulated compression speeds under 25 and 50 MHz are about 6 and 12.5 Mb/s, respectively. Table I shows the compared results of the compression speed of the dynamic pipelined and the sequential architectures for different type files under 25 MHz clock rate. The test files including text, binary, and image files, which are the manuals of commands on a UNIX system, the execution codes on a UNIX system, and some RGB images, respectively. In addition, "C-ratio" represents the compression ratio, and "C-speed (S)" and "C-speed (P)" represent the compression speed of the sequential and the dynamic pipelined architectures, respectively.

TABLE I
COMPARING RESULTS OF COMPRESSION SPEED OF THE ARITHMETIC CODERS

files	file size (bytes)	C-rate	C-speed (S) (Mbit/sec)	C-speed (P) (Mbit/sec)	speedup
Text1	73622	49.1%	2.99	6.10	2.04
Text2	51740	52.8%	3.00	6.12	2.04
Text3	108501	51.5%	3.00	6.12	2.04
Text4	23680	49.6%	2.97	6.13	2.06
Binary1	163840	23.2%	2.86	6.01	2.10
Binary2	147456	34.8%	2.91	6.03	2.07
Binary3	1064960	36.3%	2.94	6.05	2.06
Binary4	98304	40.9%	2.93	6.06	2.07
Image1	345600	42.0%	2.99	6.01	2.01
Image2	245760	13.9%	2.85	5.98	2.10
Image3	921856	41.8%	2.94	6.01	2.04
Image4	345600	28.1%	2.96	5.98	2.02

TABLE II
CHARACTERISTIC COMPARISONS OF LAYOUTS OF OUR SEQUENTIAL AND DYNAMIC PIPELINED ARCHITECTURES

characteristic	sequential	dynamic pipelining
technology	TSMC 0.8 μm SPDM	TSMC 0.8 μm SPDM
supply voltage	5V	5V
operation frequency	25 MHz	25 MHz
compression speed	3 Mbits/sec	6 Mbits/sec
chip area	4.2*4.5 mm^2	4.5*5.0 mm^2

Table II summarizes the characteristic comparisons of layouts of our sequential and dynamic pipelined architectures. The results listed in Tables I and II show that the dynamic pipelined architecture obtains about two times speedup than the sequential architecture. From the point of theoretical analysis, we can find that from Fig. 6 the execution time of each iteration of the sequential ADBAC architecture [7] needs 9 cycles, but the proposed dynamic pipelined architecture needs on average only 4.5 cycles, which is the average value of latencies gotten from experiments (see Figs. 16 and 7), to run each iteration of ABDAC. Therefore, we get the theoretical speedup equal to 9/4.5 (=2), which is almost the same the experimental values shown in

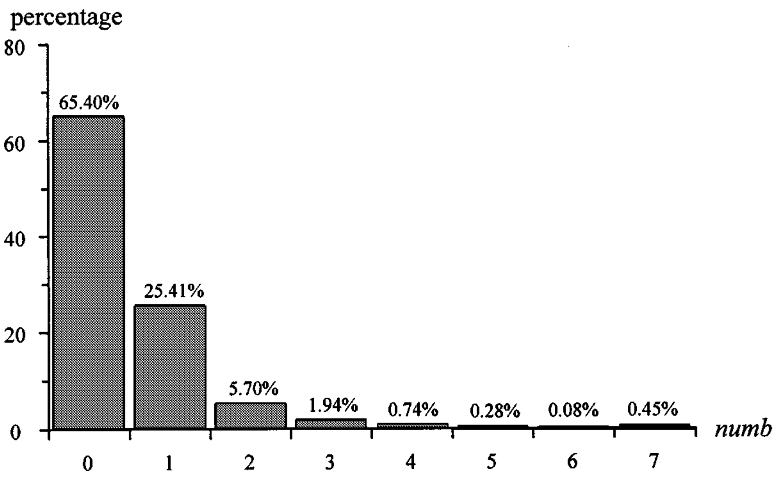


Fig. 16. Percentage distributions of $numb$ during the normalization loop phase.

Table I. Moreover, the layout area of the dynamic pipelined architecture is about 1.3 times of the one of the sequential architecture. The penalty is acceptable for its speedup.

Finally, we also make the simple comparison between our dynamic pipelined ADBAC design and the ABIC chip designed by Arps *et al.* [23]. In general, the compression ratio of our ADBAC design is higher (about 15%$\sim$25%) than the ABIC chip based on our implementation of it since our model for probability estimation is universal but the one of the ABIC chip is for the bilevel image only. But design in [23] is faster (about 1.5 times) than our dynamic pipelined ADBAC design, because it uses the customized ALU designs and employs a multiplication-free approach. We think that the performance of the ABIC chip can be further enhanced if this dynamic pipelining technique is applied.

V. Conclusion

In this paper, we have presented a dynamic pipelined design of adaptive binary arithmetic coding ADBAC for lossless data compression and decompression. To the best our knowledge, it is one of the first, if not the first, designs in which the ADBAC algorithm is executed in dynamic pipeline. Initially, the algorithm is modeled as a hardware behavioral description and then is unwound. The data and/or control dependencies in its initial pipeline pattern are analyzed and loosened by modifying their corresponding operations and with techniques similar to ASAP-speculative scheduling. Then, the novel dynamic pipeline scheme is developed and applied, and a high throughput dynamic pipeline design with variant latencies for ADBAC is constructed, the pipeline latencies of the dynamic pipelined ADBAC design are run-time depended on the execution length of the normalization phase and are unfixed naturally. The architecture and its layout have been designed, implemented, and verified with Cadence tools based on of the 0.8-μm SPDM standard cell technology. Experimental results show that the dynamic pipelined architecture gets about two times speedup with an acceptable area overhead.

Acknowledgment

The authors wish to thank the anonymous reviewers for their valuable suggestions and careful review that helped to enhance the quality of the manuscript. The authors are in the process of applying for a USA patent for the proposed systematic approach, which is to be used to design the dynamic pipelined architecture of an adaptive binary arithmetic coder.

References

[1] G. G. Langdon and J. J. Rissanen, “Compression of black-while image with arithmetic coding,” *IEEE Trans. Commun.*, vol. COM-29, pp. 858–867, June 1981.
[2] I. H. Witten, R. M. Neal, and J. G. Cleary, “Arithmetic coding for data compression,” *Commun. ACM*, vol. 30, no. 6, pp. 520–540, June 1987.
[3] G. G. Langdon and J. J. Rissanen, “A simple general binary source code,” *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 800–803, Sept. 1982.
[4] J. L. Mitchell and W. B. Pennebaker, “Optimal hardware and software arithmetic coding procedures for the Q-coder,” *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 727–736, Nov. 1988.
[5] G. Feygin, P. G. Gulak, and P. Chow, “Architectural advances in the VLSI implementation of arithmetic coding for binary image compression,” in *Proc. Data Compression Conf.*, 1994, pp. 254–263.
[6] B. Fu and K. K. Parhi, “Two VLSI design advances in arithmetic coding,” in *Proc. ISCAS*, 1995, pp. 1440–1443.
[7] S. R. Kuang, J. M. Jou, and Y. L. Chen, “The design of an adaptive on-line binary arithmetic coding chip,” *IEEE Trans. Circuits Syst. I*, vol. 45, pp. 693–706, July 1998.
[8] D. E. Knuth, “Dynamic Huffman coding,” *J. Algorithms*, vol. 6, pp. 163–180, 1985.
[9] *0.8 μm SPDM Technology Manual*, Computer & Communication Laboratory, Industry Technology Research Institute, Taiwan R.O.C., 1993.
[10] E. M. Circzyc, “Loop winding—A data flow approach to functional pipelining,” in *Proc. ISCAS*, May 1987, pp. 382–385.
[11] N. Park and A. C. Parker, “Sehwa: A software package for synthesis of pipelines from behavioral specifications,” *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 356–370, Mar. 1988.
[12] A. Aiken and A. Nicolau, “Optimal loop parallelization,” in *Proc. ACM SIGPLAN’88 Conf. Program Language Design and Implementation*, pp. 308–317.
[13] L. F. Chao, A. LaPaugh, and E. H. M. Sha, “Rotation scheduling: A loop pipelining algorithm,” in *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 566–572.
[14] M. Lam, “Software pipelining,” in *Proc. ACM SIGPLAN’88 Conf. Program Language Design and Implementation*, pp. 318–328.
[15] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, “Percolation based synthesis,” *Proc. ACM/IEEE Design Automation Conf.*, pp. 444–448, 1990.
[16] H. S. Jun and S. Y. Hwang, “Automatic synthesis of dynamically configured pipelines supporting variable data initiation intervals,” *IEEE Trans. VLSI Syst.*, vol. 4, pp. 279–285, June 1996.
[17] M. C. McFarland, A. C. Parker, and R. Camposano, “The high-level synthesis of digital systems,” *Proc. IEEE*, vol. 78, pp. 301–318, Feb. 1990.

[18] J. M. Jou, C. Chin, and Y. R. Li, “PASS: A package for automatic scheduling and sharing pipelined data paths,” in *Proc. ISCAS*, 1991, pp. 1769–1772.

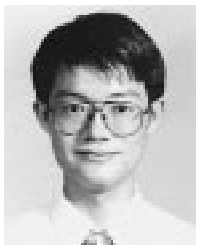
[19] U. Holtmann and R. Ernst, “Experiments with low-level speculative computation based on multiple branch prediction,” *IEEE Trans. VLSI Syst.*, vol. 1, pp. 262–267, Sept. 1993.

[20] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “Dynamic speculation and synchronization of data dependencies,” in *Proc. 24th Annu. Int. Symp. Computer Architecture*, 1997, pp. 181–193.

[21] J. M. Jou, S. R. Kuang, and R. D. Chen, “Design of low-error fixed-width multipliers for DSP applications,” *IEEE Trans. Circuits Syst. II*, vol. 46, pp. 836–842, June 1999.

[22] *Design Framework II User Guide*, Cadence Design Systems, Inc., San Jose, CA, Sept. 1992.

[23] R. Arps, T. Truong, D. Lu, R. Pasco, and T. Friedman, “A multi-purpose VLSI chip for adaptive data compression of bilevel images,” *IBM J. Res. Develop.*, vol. 32, no. 6, pp. 775–794, Nov. 1988.



Shiann Rong Kuang received the B.S. degree from National Center University, Taiwan, R.O.C., in 1990, and the M.S. and Ph.D. degrees from National Cheng Kung University, Taiwan, R.O.C., in 1992 and 1998, respectively, all in electrical engineering.

He is currently an assistant professor in the Department of Electronic Engineering, Southern Taiwan University of Technology, Taiwan, R.O.C. His research interests include VLSI chip design, high-speed digital circuit design, and data compression.

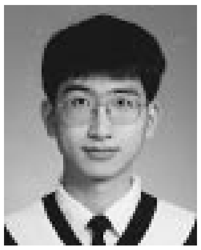
Jer Min Jou received the Ph.D. degree in electrical engineering and computer science from National Cheng Kung University, Tainan, Taiwan, R.O.C., in 1987.

In 1989, he was an associate professor in the Department of Electrical Engineering and Computer Science, National Cheng Kung University, Tainan, Taiwan, R.O.C. Currently he is a professor at the same university. His research interests include ASIC design/synthesis, SoC hardware-software codesign, System design, VLSI CAD, and asynchronous circuit design.

Dr. Jou was the recipient of a Distinguished Paper Citation at the 1987 IEEE ICCAD Conference in Santa Clara, CA. He received the Longterm Best paper award from Acer Foundation in 1998 and 1999, and was a reviewer of many journals including the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS II.

Ren Der Chen received the B.S. degree in electrical engineering from National Cheng Kung University, Taiwan, R.O.C., in 1992, and is currently working toward the Ph.D. degree in electrical engineering.

His research interests include automatic synthesis and decomposition of speed-independent circuits and VLSI chip design.



Yeu Horng Shiau received the B.S. degree in electrical engineering from National Cheng Kuang University, Taiwan, R.O.C., in 1997, and is currently working toward the Ph.D. degree in electrical engineering.

His research interests include video coding system and VLSI chip design.