Application-Specific Data Path for Highly Efficient Computation of Multi-Standard Video Codecs

Oscal T.-C. Chen, *Member, IEEE,* Li-Hsun Chen, Nai-Wei Lin, and Chih-Chang Chen

***Abstract*—A novel mechanism that flexibly adapts data flows and configures computational units is proposed to establish an application-specific data path in the Digital Signal Processor (DSP) that can efficiently perform multi-standard video codecs. Based on this mechanism, the proposed application-specific data path, using the Very Long Instruction Word (VLIW) architecture with eight computational units of five Arithmetic Logic Units (ALUs), one multiplier and two load/store units, is designed to perform five adaptive operations according to the characteristics of the low-level functions of MPEG-2, MPEG-4 and H.264/AVC video codecs. Using these adaptive operations, the proposed application-specific data path reduces the number of clock cycles required by the TI TMS320C64x data path to perform the low-level functions of the MPEG-2 video encoder and the H.264/AVC video decoder by 23.10% and 28.43%, respectively, for 30 352×288-pixel Foreman frames. Additionally, considering the operating frequency, the proposed application-specific data path reduces the computation time required by the TI TMS320C64x data path to realize the abovementioned encoder and decoder by 19.86% and 25.41%, respectively. Based on the TSMC 0.18μm CMOS cell library, the proposed application-specific data path is implemented, and exhibits the highest ratio of computational power to hardware cost among all of the data paths associated with the conventional DSPs in implementing the low-level functions of video codecs.**

***Index Terms*—digital-signal-processing-chips, video-coding, parallel-processing, pipeline-processing, VLSI.**

I. Introduction

VIDEO compression is a key technique for storing, communicating and manipulating video, and is adopted in many multimedia applications. Various video compression standards such as MPEG-2, MPEG-4, H.264/AVC and VC-1 have been established to meet numerous demands [1]-[3]. A cost-effective processor or co-processor that can run many video codecs is required to promote rapid deployment and develop competitive products that satisfy all of these demands using a single platform. A dual-core processor, like TI TMS320DM644x with a C64x DSP and an ARM CPU, is employed to implement video codecs, in which the DSP and CPU handle the low-level and high-level functions of video codecs, respectively [4]. The low-level functions include intra prediction, Quantization (Q)/Inverse Quantization (IQ), Discrete Cosine Transform (DCT)/Inverse Discrete Cosine Transform (IDCT), motion estimation, motion compensation and others, whereas the high-level functions include memory access, high-level decision-making, variable length coding/decoding, the multiplexing/demultiplexing of data streams and others. Meanwhile, the architecture developed by Berekovic *et al*. used a VLIW macroblock engine and a CPU to implement a video codec [5]. The Motorola MSC8101 processor also comprises a CPU, a SC140 DSP and peripheral controllers for multimedia applications [6].

The VLIW data path with numerous computational units offers high instruction-level parallelism, which is extensively adopted to design high-efficiency DSPs such as TI TMS320C64x, Motorola/Lucent StarCore140 and NEC SPXK5 processors [7]-[9]. As for the special applications, special instructions are typically designed to increase the computational efficiency of the VLIW data path. If the operating modes and interconnections of the computational units in the VLIW data path can be adaptively and dynamically changed to perform the special functions efficiently, then the computation efficiency of the data path can be further enhanced. In this work, the adaptive mechanism, which is used to increase the flexibility of data delivery in a VLIW data path, is proposed to improve the processing parallelism for a specific application [10]-[12]. The increased flexibility of data delivery enables an input/output port of the computational unit to access data for itself or other computational units. Given such an adaptive design, with increased flexibility of data access, the input/output ports of the computational units can be effectively used to increase the number of hardware components being executed at the same time. As well as computing data accessed from a register file, a computational unit can perform calculations on data obtained from other computational units, like data forwarding, to increase the utilization of its hardware components without the need for additional input/output ports [13]. Hence, by increasing the flexibility of data delivery, the proposed adaptive mechanism can be applied to reduce the number of the idle hardware components in the computational units. In addition to increasing the flexibility of data delivery, the proposed adaptive

Manuscript received October 9, 2005. This work was partially supported by the National Science Council of Taiwan under Contract No. NSC 93-2219-E-194-004.

Oscal T.-C. Chen is with the Department of Electrical Engineering, National Chung Cheng University, Chiayi, 621 Taiwan, R.O.C. (phone: +886-5-2720411 ext 33207; fax: +886-5-2720862; e-mail: oscal@ee.ccu.edu.tw).

Li-Hsun Chen and Chih-Chang Chen are with the Department of Electrical Engineering, National Chung Cheng University, Chiayi, 621 Taiwan, R.O.C. (e-mail: lhchen@samlab.ee.ccu.edu.tw; ccchen@samlab.ee.ccu.edu.tw).

Nai-Wei Lin is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, 621 Taiwan, R.O.C. (e-mail: naiwei@cs.ccu.edu.tw).

mechanism can configure the computational units of the data path to become special computational units for effectively computing a specific function.

The characteristics of the low-level functions of the MPEG-2, MPEG-4 and H.264/AVC video codecs are such that the proposed adaptive mechanism is adopted to implement an application-specific data path, involving the execution of five adaptive operations, which effectively performs these low-level functions. The proposed application-specific data path is designed by referring to the TI TMS320C64x data path. Additionally, the hardware implementation of the five adaptive operations is based on the available computational units allocated in the proposed application-specific data path to reduce the hardware complexity. Hence, by simply adjusting the configurations of the computational units, and the paths along which data are delivered among computational units, the proposed application-specific data path can execute the five adaptive operations at a low hardware cost. The results of the implementation herein reveal that the hardware cost of executing the five adaptive operations represents only 5.06% of the overall data path when the TSMC 0.18μm CMOS cell library is used to implement the application-specific data path. These adaptive operations enable the proposed application-specific data path to conserve 23.10% and 28.43% of the clock cycles demanded by the TI TMS320C64x data path for executing the low-level functions of the MPEG-2 video encoder and H.264/AVC video decoder, respectively. Therefore, the proposed application-specific data path with a high utilization of hardware resources is established to yield the highest ratio of computation power to hardware cost than the data paths of conventional DSPs.

II. Proposed Adaptive Mechanism

A VLIW architecture with numerous computational units can execute instructions in a highly parallel manner. However, a computational unit may comprise several hardware components that cannot easily be integrated, so some hardware components of this computational unit tend to be idle during the execution of a specific instruction, wasting hardware resources. The TI TMS320C64x data path with two sets of .L, .S, .M and .D units is taken as an example to explicate further the above phenomenon [7]. The .L and .S units execute arithmetic, logic, shifting, comparing, packing/unpacking and other instructions. The hardware component used to execute arithmetic, logical and comparative instructions primarily consists of an adder and some logic gates. This hardware component cannot easily be integrated with the one executing shifting and packing/unpacking instructions, most of which involve multiplexers. Accordingly, each of the .L and .S units has two hardware components. The .M unit consists of two identical hardware components, each of which performs one 16×16-bit multiplication, one multiplication on four pairs of 8-bit data and one inner product on four pairs of 8-bit data. Combining these two hardware components enables the .M unit to perform a further 32×16-bit multiplication, one multiplication on two pairs of 16-bit data and one inner product on two pairs of 16-bit data. The .D unit can compute the address of the data and execute some addition/subtraction operations. The execution of these operations is associated only with adders, so the .D unit is treated as an independent hardware component. With two sets of .L, .S, .M and .D units, the TI TMS320C64x data path can be regarded as having 14 independent hardware components (2×(2+2+2+1)=14). Summing the numbers of hardware components used in each clock cycle to execute a program enables the degree of utilization of hardware components in the TI TMS320C64x data path that runs the program to be formulated as follows;

$$\begin{aligned} \textit{Degree of Utilization of Hardware Components} &= \frac{1}{m}\sum_{i=0}^{m-1}\frac{N_i}{14} \\ &= \frac{1}{m\times 14}\sum_{i=0}^{m-1}N_i \end{aligned} \quad (1)$$

where m is the number of clock cycles required to run the program and N_i is the number of hardware components used in the i^{th} clock cycle.

(1) yields the degrees of utilization of hardware components in the TI TMS320C64x data path used to run various programs. However, each of the .L, .S and .M units has a pair of hardware components, so these hardware components may not be used simultaneously and thus the degree of utilization of the hardware components is reduced. Hence, in maximizing the degree of utilization of hardware components, the TI TMS320C64x data path is assumed to have 14 input/output ports associated with its 14 hardware components, so 14 instructions at most are to be executed in parallel. This new architecture with 14 input/output ports has the same number of hardware components as TI TMS320C64x data path, but has a higher bandwidth of data access than has TI TMS320C64x data path. Based on the architecture with 14 input/output ports, the programs can be re-coded to determine their assembly codes. (1) is then applied to estimate the maximum degrees of utilization of the hardware components. Figure 1 shows the degrees of utilization of the hardware components of the TI TMS320C64x data path and the architecture with 14 input/output ports used to run various programs. This figure reveals that, excluding the Finite Impulse Response (FIR), the multiplication of two matrices and the weighted vector sum, the TI TMS320C64x data path has lower degrees of utilization of the hardware components than the architecture with 14 input/output ports, indicating that the utilization of hardware components associated with TI TMS320C64x data path can still be improved by adapting data delivery.

Embedding the proposed adaptive mechanism enables a VLIW data path, as presented in Fig. 2, to exhibit highly flexible data delivery and configurable computational units. The data delivery by the adaptive mechanism can be classified into two modes. An interconnection circuit between the register file and the computational units can be used to enable the first mode to be implemented such that an input/output port accesses data for its corresponding or other computational unit(s) to increase the flexibility of access to data. In the second mode

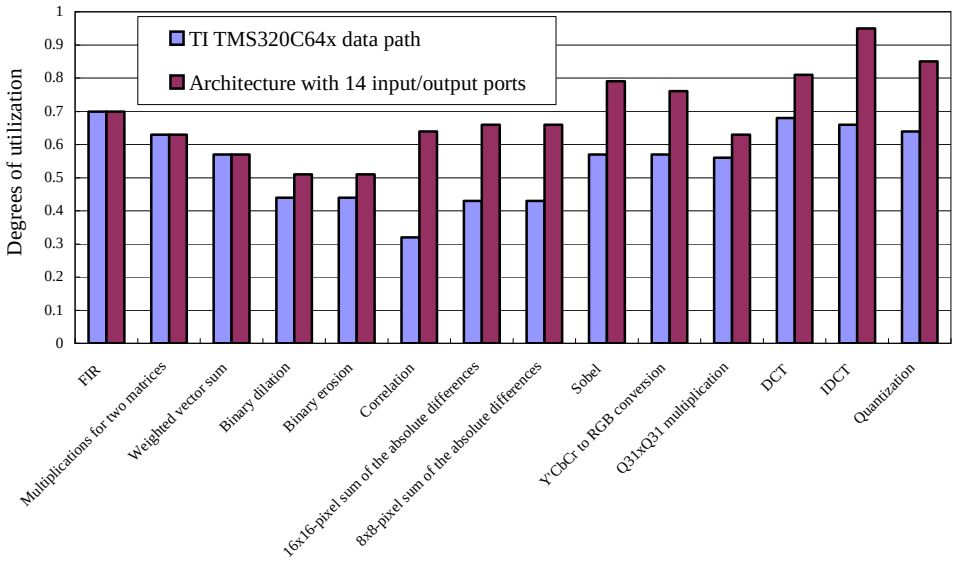

Fig. 1. Degrees of utilization of TI TMS320C64x data path and the architecture with 14 input/output ports.

like data forwarding, an interconnection circuit and some pipelined latches located after computational units can deliver the results among the computational units and improve the computational efficiency. As well as computing the data accessed from the register file, a computational unit can simultaneously compute the data generated by the other computational units to increase the utilization of its hardware components. Moreover, the computational units of the data path can be configured into special computational units for effective computing of a specific function. This study uses the term "adaptive operation" to interpret the delivery of data and the configuration of computational units by using the adaptive mechanism, and then performs computations on these data. The proposed adaptive mechanism is based on the characteristics of the low-level functions of the MPEG-2, MPEG-4 and H.264/AVC video codecs, to implement a VLIW data path with high computational efficiency.

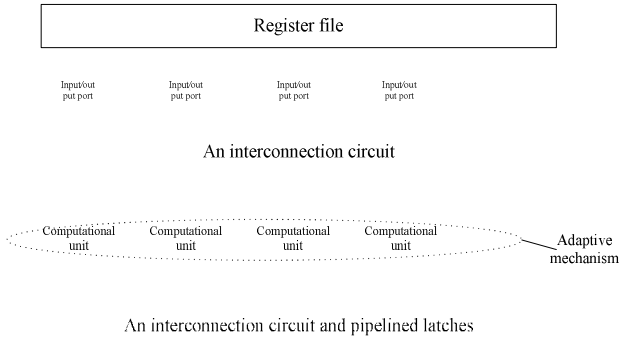

Fig. 2. A VLIW data path using the proposed adaptive mechanism.

III. Proposed Application-Specific Data Path

To effectively implement multi-standard video codecs, the dual-core platform with the CPU, DSP, memory and input/output device is proposed, as shown in Fig. 3, in which the CPU manipulates input/output access, memory management and high-level functions of video codecs, while the DSP computes the low-level functions of video codecs. The proposed adaptive mechanism is implemented in the data path of this DSP to dynamically adapt the flow of input and output data and configure computational units. Figure 4 illustrates the proposed application-specific data path, which comprises two register files and eight computational units, with reference to the data path of the TI TMS320C64x DSP. The following introduces the proposed application-specific data path in detail.

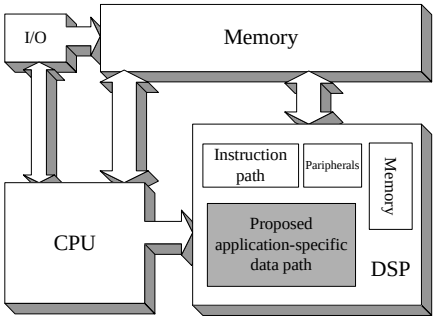

Fig. 3. A dual-core platform for multi-standard video codecs.

A. Computational Units

With reference to the configurations of computational units in the data paths of the conventional DSPs, TI TMS320C64x data path comprises two load/store units, two 32-bit ALUs, two 40-bit ALUs, and two 32×16-bit multipliers, one of which can be treated as two 16×16-bit multipliers [7]. Motorola/Lucent StarCore140 data path is established with two address generation units, four 40-bit ALUs and four 16×16-bit multipliers; NEC SPXK5 data path includes two 32-bit data address units, two 40-bit ALUs and two 16×16-bit multipliers [8], [9]. In these data paths, the number of multipliers is close to the number of ALUs. However, computations of popularly used

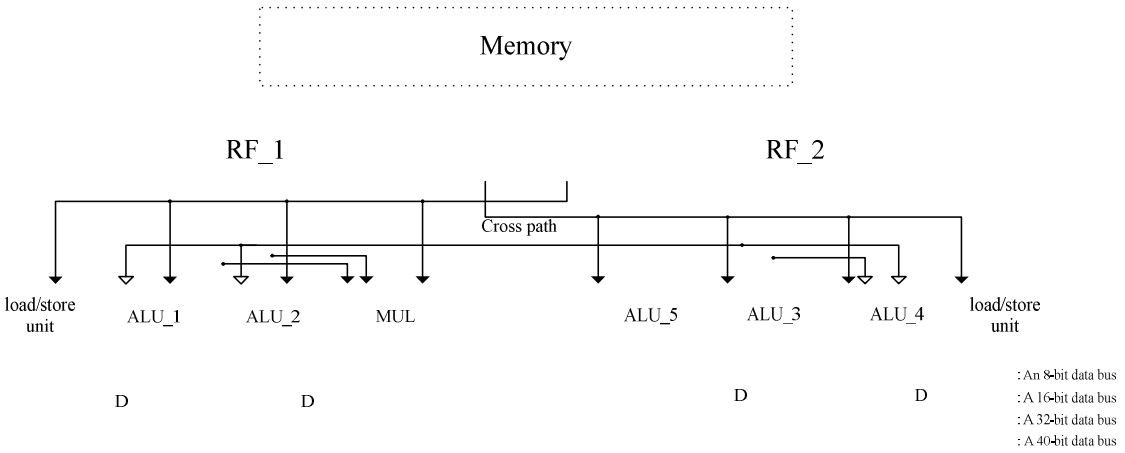

Fig. 4. Proposed application-specific data path.

video codecs require the multiplications to be fewer than the arithmetic and logical operations. In particular, the H.264/AVC video codec performs addition, subtraction and shifting operations to execute simple multiplication operations in the integer transform and inverse integer transform functions. Hence, with the reduced number of multipliers and the increased number of ALUs, the data path has an enhanced ability to parallel process the video codecs. In Fig. 4, the proposed application-specific data path includes five ALUs, one multiplier (MUL) and two load/store units.

With referring to the TI TMS320C64x data path, each of the load/store units in the proposed application-specific data path can determine the address at which up to 64-bit data are loaded or stored. It can also perform 32-bit addition/subtraction. In ALU_1, ALU_2, ALU_3 and ALU_4, they can perform 40-bit arithmetic, 32-bit logic, comparative and shifting operations. ALU_5 can execute 32-bit arithmetic, logic, shifting and branch operations. All five ALUs have the Single Instruction Multiple Data (SIMD) operations. Moreover, these five ALUs can perform multiple packing/unpacking operations to align the data for the SIMD operations. The carry select adder structure is employed to implement the adders in the five ALUs, as displayed in Fig. 5, to minimize the computational delay. With the same multiplier as that of the TI TMS320C64x data path, the proposed MUL can perform the multiplication and the inner product operations on 8, 16 and 32-bit data where an 8×8-bit multiplier is regarded as a fundamental unit. Changing the interconnection among eight 8×8-bit multipliers and hardware components enables the MUL shown in Fig. 6 to execute several multiplication operations.

B. Register File (*RF*)

In the proposed register file, two clusters of RF_1 and RF_2 are designed with reference to the TI TMS320C64x data path's register file. Each cluster includes 32 32-bit registers to support the operations of computational units. Two additional 32-bit read ports are accommodated to implement the cross path between two clusters and thus eliminate the moving of data between two clusters. Accordingly, this register file can effectively access data to execute general and adaptive instructions.

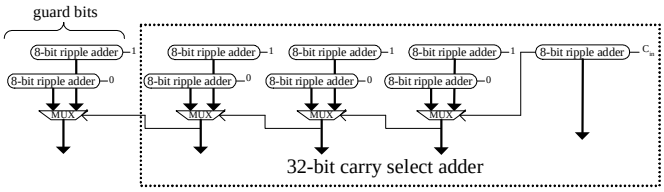

Fig. 5. A 40-bit carry select adder.

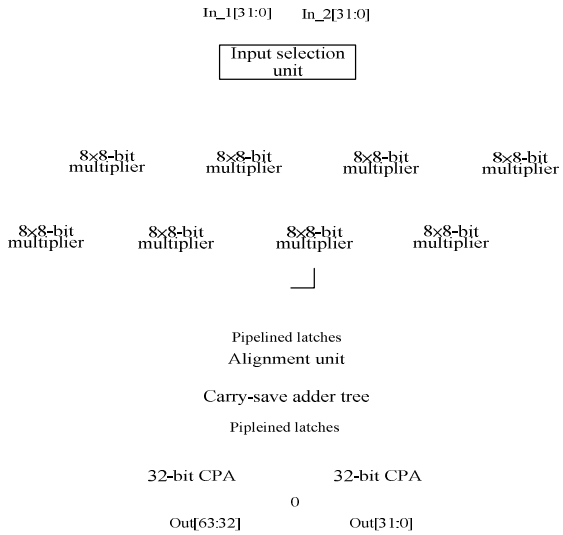

Fig. 6. The MUL architecture.

C. Adaptive Operations

According to MPEG-2, MPEG-4 and H.264/AVC video codecs, five adaptive operations are developed to accelerate the computation of their low-level functions with high computational complexities, including motion estimation, motion compensation, DCT, IDCT, integer transform, inverse integer transform, quantization, inverse quantization, mode decision, intra prediction and inter prediction. The structures of eight computational units are well designed, as described above, to implement easily adaptive operations. Additionally,

TABLE I
FEATURES OF THE FIVE ADAPTIVE OPERATIONS

Computational units / Adaptive operations	ALU_1	ALU_2	ALU_3	ALU_4	ALU_5	MUL
SAD adaptive operation	Performing four of 8-bit subtraction and absolute operations	Performing four of 8-bit subtraction and absolute operations	Performing four of 8-bit subtraction and absolute operations	Performing four of 8-bit subtraction and absolute operations	Not used	Performing the addition on 16 8-bit data and a 32-bit accumulation
MAS adaptive operation	Performing an 8×8-bit multiplication	Performing an 8×8-bit multiplication	Performing an 8×8-bit multiplication	Performing an 8×8-bit multiplication	Adding the four results from ALU_1, ALU_2, ALU_3 and ALU_4 with a 32-bit value to complete a multiplication and accumulation operation (MAC), and then performing a shifting operation on the result of MAC	Not used
AoS adaptive operation	Not used	Two input ports used for a CPA of MUL	Not used	Not used	Not used	Performing an addition or subtraction operation
AA adaptive operation	Performing an alignment operation and an SIMD arithmetic operation	Performing an alignment operation and an SIMD arithmetic operation	Performing an alignment operation and an SIMD arithmetic operation	Performing an alignment operation and an SIMD arithmetic operation	Not used	Not used
AFP adaptive operation	Performing four average operations on four sets of four 8-bit data	Performing four average operations on four sets of four 8-bit data	Not used	Not used	Not used	Not used

effectively using the input/output ports of computational units to access data, and sending the outputs of computational units to each other, the designs of the five adaptive operations do not increase the number of ports on the register file. Therefore, only the configurations of computational units and the paths along which data are delivered among computational units must be adjusted, and the proposed application-specific data path can implement the five adaptive operations at a low cost. Table I lists the features of the five adaptive operations, which are further illustrated as follows.

1) Sum-of-Absolute-Difference (*SAD*) *Adaptive Operation*: Motion estimation and mode decision of H.264/AVC involve primarily SAD operations, which are highly computationally complex, so accelerating the SAD operation can effectively improve the computational performance of video codecs [14]. The SAD adaptive operation, shown in Fig. 7, is designed to achieve this goal. Figure 7(a) presents the data flow associated with the SAD adaptive operation. As indicated in Table I, ALU_1, ALU_2, ALU_3 and ALU_4 can simultaneously perform 16 8-bit subtraction and absolute operations, to obtain 16 values that are stored at the latches for subsequent accumulation. The MUL is then applied to add these 16 values, which in turn are accumulated with a 32-bit value to accomplish the SAD operations on 16 pixels in parallel. During the SAD adaptive operation, the 40-bit carry select adders in ALU_1, ALU_2, ALU_3 and ALU_4 must be converted into something that can execute four 8-bit subtraction and absolution operations, as shown in Fig. 7(b). Two 8-bit ripple adders and one multiplexer (MUX) are required in an 8-bit subtraction and absolution operation, so that if the input data are *X* and *Y*, then one 8-bit ripple adder executes the subtraction *X*-*Y*, while the other executes the subtraction *Y*-*X*. The carry-out bit from *Y*-*X* is then used to control the MUX to choose the positive result. This design can cause that the four ALUs generate simultaneously 16 8-bit subtraction and absolute values. The 16 8-bit values yielded by the four ALUs are treated as the partial products of the MUL, and are added in the MUL, as displayed in Fig. 7(c). Adding these 16 8-bit values involves four 8×8-bit multipliers to add these 16 8-bit values under the same precision. The other four 8×8-bit multipliers are set to generate zero outputs. The results accumulated from the eight 8×8-bit multipliers are summed by the carry-save addition manner, and thus this summed result and the other 32-bit input value are added to generate the carry and sum values. Finally, a 32-bit CPA is employed to add the carry and sum values to complete the SAD adaptive operation. Based on the characteristics of additions on multiple values, the MUL can be easily adapted to perform the SAD adaptive operation at a low cost.

2) Multiplication-Addition-Shifting (*MAS*) *Adaptive Operation*: Although few multiplication operations are required in H.264, the multiplication operations for some functions of MPEG-2 and MPEG-4 are still used commonly; these include, for instance, quantization/inverse quantization and DCT/IDCT. Accordingly, the MAS adaptive operation can combine some adders as a 16×16-bit multiplier to increase the number of multipliers and thus effectively execute these functions, which require extensive multiplications. Figure 8 displays this adaptive operation based on ALU_1, ALU_2, ALU_3, ALU_4 and ALU_5. Figure 8(a) presents the block diagrams of a 40-bit carry select adder and an 8×8-bit array multiplier. Using a 7-bit ripple adder as a basic unit, an 8×8-bit array multiplier is constructed by eight 7-bit ripple adders that come from a 40-bit carry select adder. Additionally, apart from adding logic gates to generate partial products, some multiplexers can be allocated to change two input data in each of eight 7-bit ripple adders of a 40-bit carry select adder, so that a 40-bit carry select adder can be simply converted into an 8×8-bit array multiplier.

The four 40-bit adders in ALU_1, ALU_2, ALU_3 and ALU_4 are transformed into four 8×8-bit array multipliers. Through a cross path from the cluster RF_2 of the register file, the 8×8-bit array multipliers in ALU_1 and ALU_2 can access

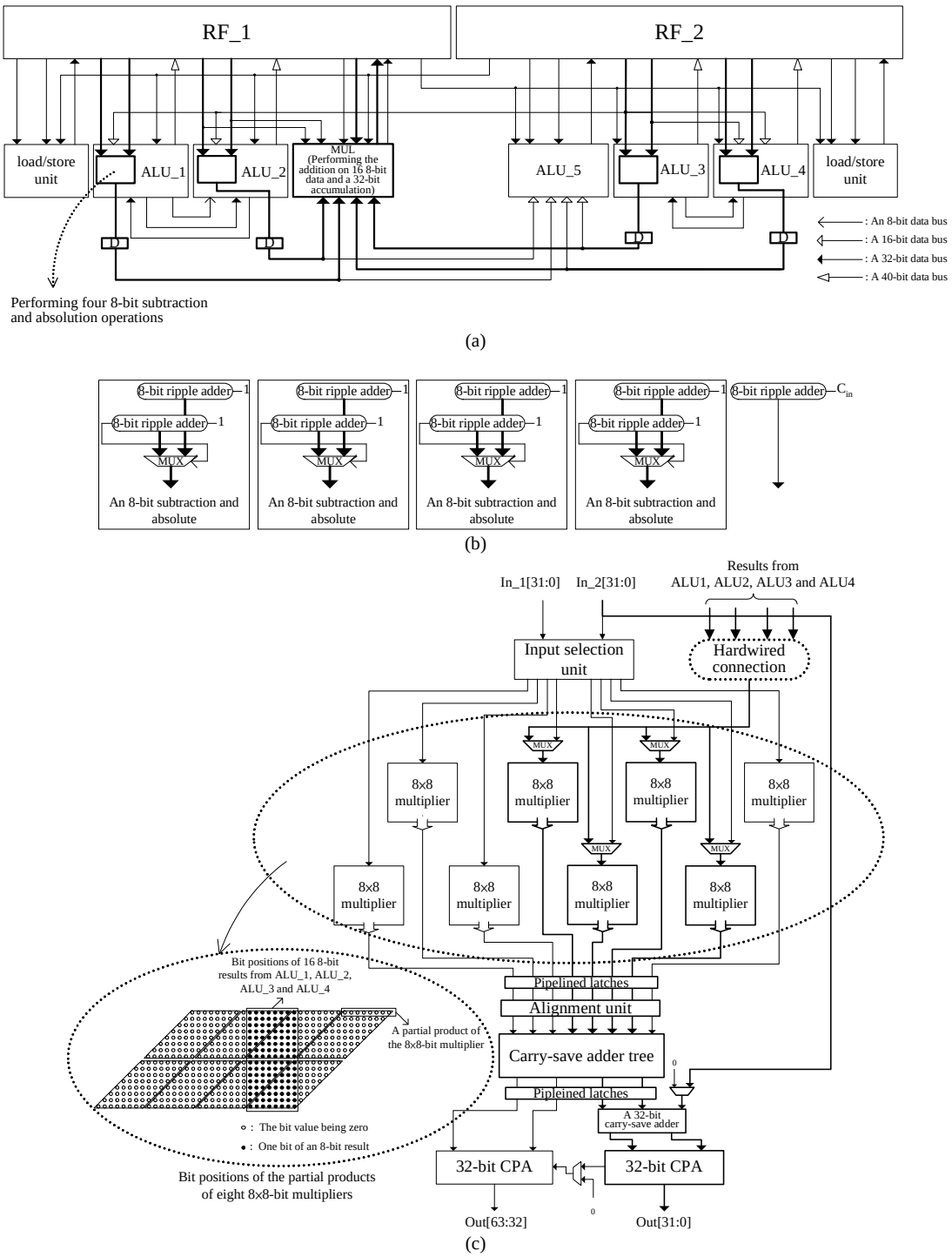

Fig. 7. SAD adaptive operation. (a) Data flow. (b) Configuration of the adder. (c) Configuration of the MUL.

the same data as those in ALU_3 and ALU_4, which in turn execute four 8×8-bit multiplication operations to yield the four 16-bit results. These four results are latched and then inputted into ALU_5. Figure 8(b) depicts the data flow of this adaptive operation, and the configuration of ALU_5. In ALU_5, the carry-save addition is executed to add the four 16-bit results and a 32-bit datum, which is accessed from the register file, to derive the sum and carry values. The sum and carry values are added using the 32-bit adder in ALU_5 to complete a multiplication and accumulation operation. However, in case this 32-bit datum from the register file is zero, this adaptive operation becomes a 16×16-bit multiplication. Additionally, by changing the configuration of ALU_5, the adder's output can be sent to the programmable shifter, as shown in Fig. 8(b). Apart from a 32-bit adder and a programmable shifter, ALU_5 needs a carry-save addition unit, a pipelined latch and some multiplexers, to perform this adaptive operation. In Fig. 8, the input/output ports and hardware components of ALU_3 and ALU_5 are employed to implement the MAS adaptive operation. Additionally, ALU_1, ALU_2 and ALU_4 use

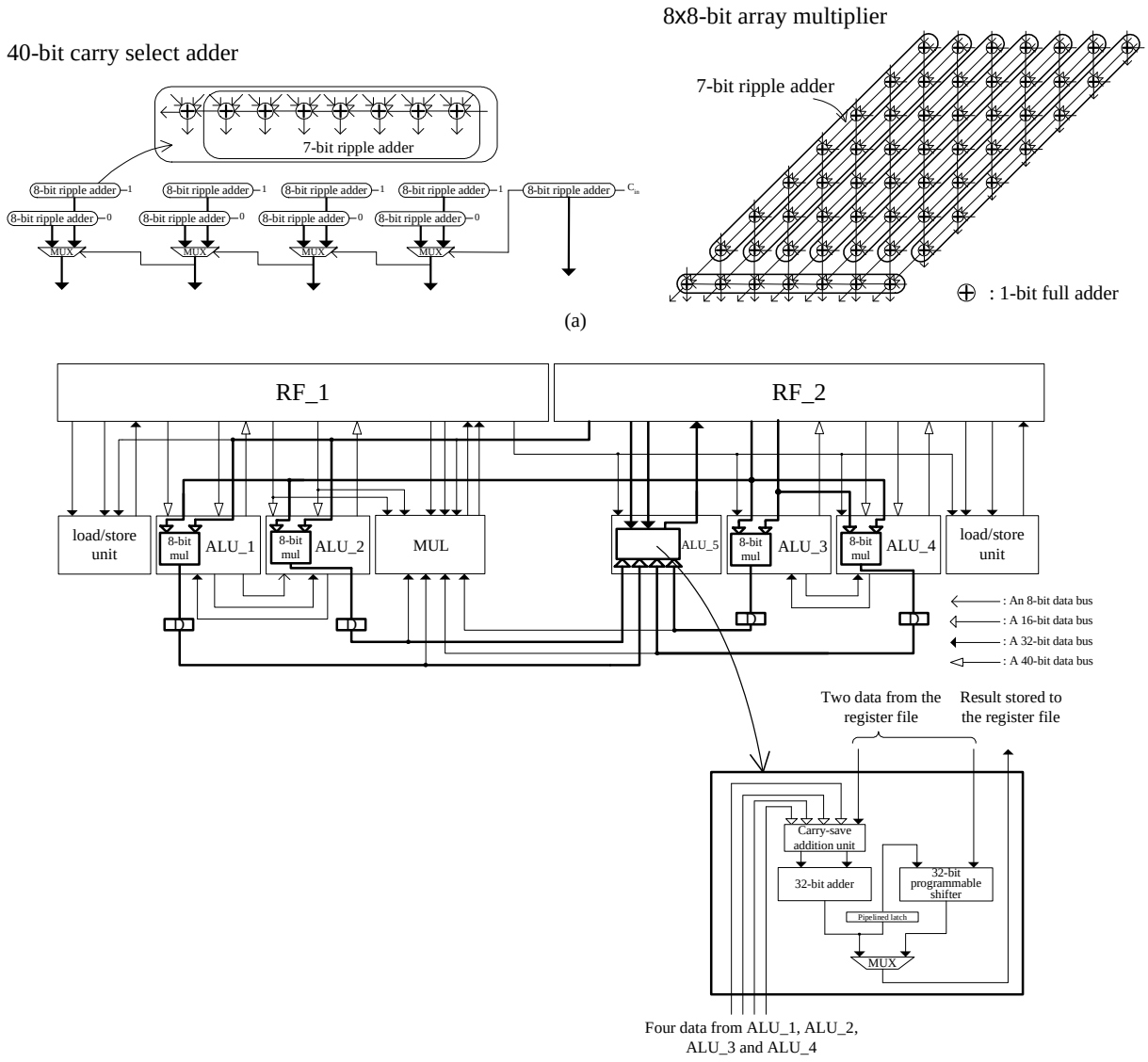

Fig. 8. MAS adaptive operation. (a) Block diagrams of a 40-bit carry select adder and an 8×8-bit array multiplier. (b) Data flow.

only their adders to execute the MAS adaptive operation, so they can also execute the shifting, the logical or the packing/unpacking instructions that do not involve the adder. When this adaptive operation is performed in quantization and inverse quantization, the proposed application-specific data path can improve the parallel processing of their multiplication, addition/subtraction and shifting operations. Similarly, the MAS adaptive operation can be also applied in DCT and IDCT to enhance the computational performance.

3) Addition-or-Subtraction (*AoS*) *Adaptive Operation*: As presented in Fig. 6, MUL has two CPAs, which add the sum and carry values generated by the carry-save adder tree. However, when the MUL executes such instructions as one 16×16-bit multiplication and the inner products, only a single CPA is used while the other is idle. Additionally, when executing the MAS adaptive operation, the ALU_1, ALU_2, ALU_3 and ALU_4 cannot execute the instructions that are associated with their adders, such as addition, subtraction and comparing. Hence, for these two reasons, the AoS adaptive operation, as shown in Fig. 9, is designed to include an extra addition or subtraction. In this figure, when ALU_2 is used to perform the MAS adaptive operation, its two 32-bit input ports can be used by a 32-bit CPA of the MUL to access the data. Therefore, the proposed application-specific data path can execute an additional addition or subtraction instruction and thus improve its computational performance. As well as using the input ports of the other computational unit to access data, this adaptive operation can also efficiently exploit hardware components.

4) Alignment-Addition (*AA*) *Adaptive Operation*: In video codecs, the word length of input data is primarily 8 or 16 bits. Designing the SIMD operations simultaneously to process multiple 8-bit or 16-bit data in computational units can effectively increase the computational efficiency of video codecs. This SIMD design has been extensively used in many conventional data paths. However, before the SIMD operation has been performed, the data paths must execute extra packing operations to align the data for the SIMD operations, so that the packing operations degrade the computational efficiency

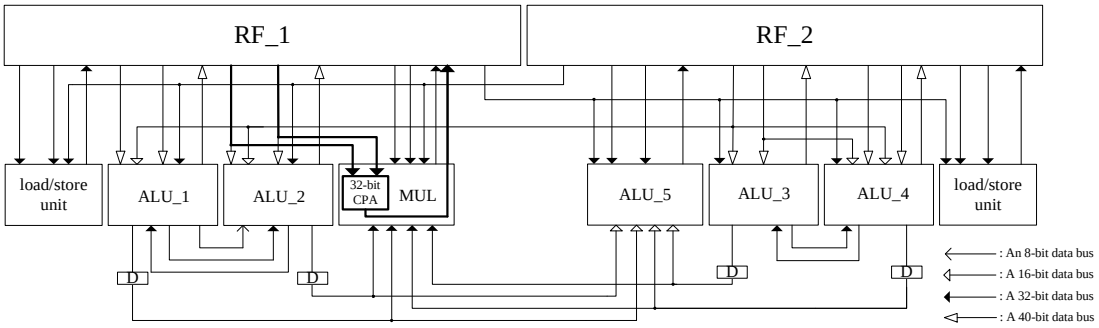

Fig. 9. AoS adaptive operation.

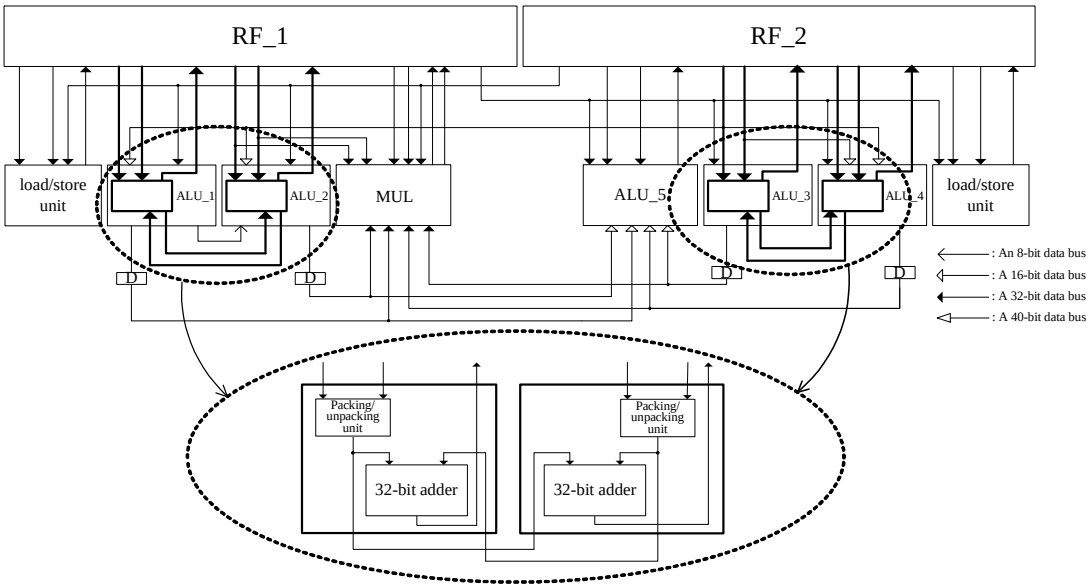

Fig. 10. AA adaptive operation.

of the SIMD. The AA adaptive operation was developed to address this issue; Fig. 10 shows its data flow. Herein, ALU_1 and ALU_2 are configured to enable their packing/unpacking units deliver outputs to the 32-bit adders to accomplish two alignments and two SIMD arithmetic operations. Like ALU_1 and ALU_2, ALU_3 and ALU_4 can also execute such an adaptive operation. With a packing/unpacking unit and a 32-bit adder to execute alignment and arithmetic operations, respectively, each of ALU_1, ALU_2, ALU_3 and ALU_4 uses only some multiplexers to change its configuration. Hence, the proposed application-specific data path can simultaneously perform two of the AA adaptive operations.

5) Average-on-Four-Point (*AFP*) *Adaptive Operation*: Interpolation and intra prediction are used mainly by average operations on two 8-bit data and four 8-bit data. Conventional data paths employ special instructions to perform average operations on multiple pairs of 8-bit data in parallel, such as the AVG4 instruction of TI TMS320C64x data path and the QUADAVG instruction of Philips PNX1500 data path [15], [16]. Nevertheless, these data paths cannot effectively perform the average operation on four 8-bit data in parallel, the equation of which can be described as follows;

$$t = (x0+x1+x2+x3+2) >> 2. \quad (2)$$

When addition operations are executed on four 8-bit data, *x0*, *x1*, *x2* and *x3*, and a constant value, two, the word length of the summed result exceeds 8 bits. Although the summed result is shifted right by 2 bits to yield the final 8-bit value, the conventional data paths still execute four 16-bit addition operations and a single shifting operation to compute (2), and thereby waste hardware resources. (2) is further modified as follows, to use effectively hardware resources;

$$\begin{aligned} t &= (x0+x1+x2+x3+2) >> 2 \\ &= ((x0+x1+1)+(x2+x3+1)) >> 2 \\ &= (y0+y1) >> 2. \end{aligned} \quad (3)$$

y0 and *y1* equal (*x0*+*x1*+1) and (*x2*+*x3*+1), respectively. From (3), the averaging operation on four 8-bit data comprises two steps. The first is to compute the two values, (*x0*+*x1*+1) and (*x2*+*x3*+1), which are determined using two 8-bit adders with carry-in bits being one to generate two 9-bit outputs, including two carry-out bits. In the second step, the operation, (*y0*+*y1*)>>2, is implemented using a 9-bit adder with a carry-in bit being zero where *y0* and *y1* are the two 9-bit outputs of the two 8-bit adders in the first step. The AFP adaptive operation, as displayed in Fig. 11, is developed according to (3), to perform effectively the averaging operation on four 8-bit data. The two 40-bit carry select adders in ALU_1 and ALU_2 that

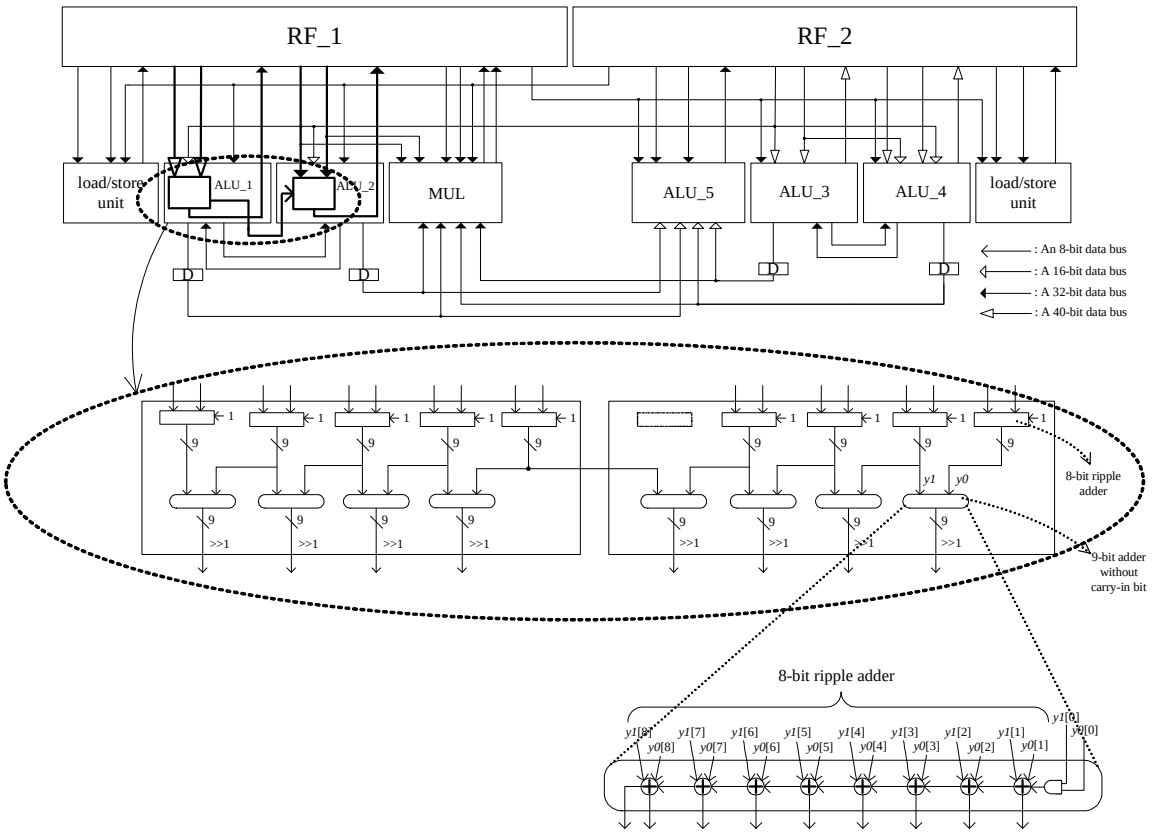

Fig. 11. AFP adaptive operation.

consist of 18 8-bit ripple adders can be configured as a special unit that can perform averaging operations on eight sets of four 8-bit data in parallel. The second step involves eight 9-bit additions with zero carry-in and eight added results shifted right by two bits, each of which eight addition and shifting operations can be simply implemented by using an 8-bit adder and a 1-bit AND gate to generate an average value of four 8-bit data. Accordingly, as shown in Fig. 11, by adapting the output paths and the carry-in bits of 8-bit ripple adders, the 8-bit ripple adders in ALU_1 and ALU_2 can be effectively used to accelerate the averaging operations to ensure the efficiency of computation.

D. Code Generation

This sub-section discusses the effects of introducing the application-specific data path on a compiler, and in particular, on the code generation phase of a compiler. Each of the five adaptive operations can be implemented by a sequence of primitive instructions, called adaptive instructions, which are characterized in Table II. The instruction format for the application-specific data path is based on that for the TMS320C64x data path [15]. Since the number of operation types associated with each computational unit in the TMS320C64x data path is under 64, sufficient spaces remain to accommodate these adaptive instructions.

A simple and effective way to retarget a compiler to a new architecture is to provide intrinsics for the new features of the new architecture. For simplicity, the organization of a C compiler for a VLIW architecture can be abstracted, as displayed in Fig. 12. It comprises a C compiler for a general architecture, which translates the C code with intrinsics for adaptive operations into a linear assembly code, and an instruction scheduler for a VLIW architecture, which schedules the linear assembly code into a parallel assembly code in a manner that meets the resource constraints of adaptive instructions. Table III lists the intrinsics designed for the SAD, MAS, AA and AFP adaptive operations. The AoS adaptive operation is unspecified in the C source code and is introduced by the instruction scheduler. Therefore, no intrinsic exists for the AoS adaptive operation. The handling of the AoS adaptive operation is discussed later. Since the translation of intrinsics is quite straightforward, the rest of this sub-section discusses only the instruction scheduler.

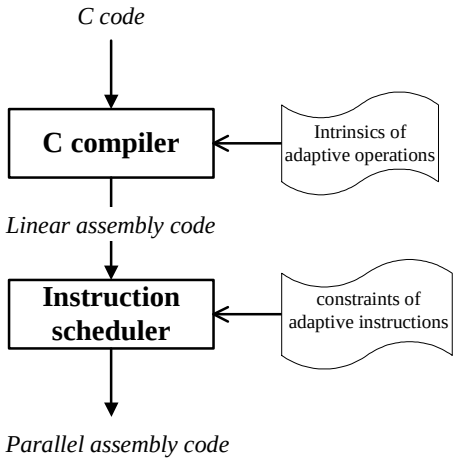

Fig. 12. Organization of a compiler for a VLIW architecture.

TABLE II
CHARACTERISTICS OF ADAPTIVE INSTRUCTIONS

Adaptive operations	Adaptive instructions	Used computational unit	Latency (clock cycles)	Description	Resource constraints
SAD	SAD_AD4 src1, src2	ALU_1, ALU_2, ALU_3 or ALU_4	1	Perform the subtraction and absolution on one pair of four packed 8-bit values	(1) Four SAD_AD4 instructions must be simultaneously performed in ALU_1, ALU_2, ALU_3 and ALU_4 (2) SAD_AD4 and SAD_SUM must be performed in two successive clock cycles
	SAD_SUM src, dst	MUL	3	Perform the addition on 16 8-bit values from ALU_1, ALU_2, ALU_3 and ALU_4, and perform a 32-bit accumulation	
MAS	MAS_MULL src1, src2	ALU_3	1	Perform four 8×8-bit multiplication on the values extracted from lower or higher half words of src1 and src2	(1) ALU_1, ALU_2 and ALU_4 only perform the instructions not involving the adder such as shifting or packing/unpacking, when MAS_MULx is performed (2) MAS_MULx and MAS_ADD or MAS_ADD_SHx must be performed in two successive clock cycles
	MAS_MULH src1, src2	ALU_3	1		
	MAS_ADD src1, dst	ALU_5	1	Perform the addition on the four 8×8-bit multiplication results from ALU_1 ALU_2, ALU_3 and ALU_4 with a 32-bit value (src1), and store the result to the register dst	
	MAS_ADD_SHL src1, src2, dst	ALU_5	2	Perform the addition on the four 8×8-bit multiplication results from ALU_1 ALU_2, ALU_3 and ALU_4 with a 32-bit value (src1) then shift left or right src2 bits, and store the result to the register dst	
	MAS_ADD_SHR src1, src2, dst	ALU_5	2		
AoS	AoS_ADD src1, src2, dst	ALU_2	1	Perform addition or subtraction on src1 and src2	MUL can't perform the two 16×16-bit or one 32×16-bit multiplication operation, when these adaptive instruction are performed
	AoS_SUB src1, src2, dst	ALU_2	1		
	AoS_ADD2 src1, src2, dst	ALU_2	1	Perform addition or subtraction on one pair of packed 16-bit values	
	AoS_SUB2 src1, src2, dst	ALU_2	1		
AA	AA_PLL_ADD2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1	Perform an alignment operation on src1 and src2, and then perform an addition or subtraction with a value from the other computational unit	Two of these adaptive instructions must be simultaneously performed in ALU_1 and ALU_2 or ALU_3 and ALU_4
	AA_PLH_ADD2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
	AA_PHL_ADD2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
	AA_PHH_ADD2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
	AA_PLL_SUB2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
	AA_PLH_SUB2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
	AA_PHL_SUB2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
	AA_PHH_SUB2 src1, src2, dst	ALU_1, ALU_2, ALU_3 or ALU_4	1		
AFP	AFPL src1, src2, dst	ALU_2	1	Perform four average operations on four 8-bit values	AFPL and AFPH must be performed simultaneously

TABLE III
INTRINSICS OF THE ADAPTIVE OPERATIONS

Adaptive operations	C compiler intrinsics	Pseudo instructions	Adaptive instructions
SAD	int dst = _sad (int a0, int a1, int b0, int b1, int c0, int c1, int d0, int d1)	SAD	SAD_AD4 a0, a1 SAD_AD4 b0, b1 SAD_AD4 c0, c1 SAD_AD4 d0, d1 SAD_SUM dst, dst
MAS	int dst = _mas_mulx_add (int m0, int m1, int a0), where x can be h or l	MAS_MULx_ADD	MAS_MULx m0, m1 MAS_ADD a0, dst
	int dst = _mas_mulx_add_shl (int m0, int m1, int a0, int s0), where x can be h or l	MAS_MULx_ADD_SHL	MAS_MULx m0, m1 MAS_ADD_SHL a0, s0, dst
	int dst = _mas_mulx_add_shr (int m0, int m1, int a0, int s0), where x can be h or l	MAS_MULx_ADD_SHR	MAS_ MULx m0, m1 MAS_ADD_SHR a0, s0, dst
AA	double dst = _aa_pxx_add2_pxx_add2 (int a0, int a1, int b0, int b1), where xx can be ll, lh, hl and hh	AA_Pxx_ADD2_Pxx_ADD2	AA_Pxx_ADD2 a0, a1, upper word of dst AA_Pxx_ADD2 b0, b1, lower word of dst
	double dst = _aa_pxx_add2_pxx_sub2 (int a0, int a1, int b0, int b1), where xx can be ll, lh, hl and hh	AA_Pxx_ADD2_Pxx_SUB2	AA_Pxx_ADD2 a0, a1, upper word of dst AA_Pxx_SUB2 b0, b1, lower word of dst
	double dst = _aa_pxx_sub2_pxx_add2 (int a0, int a1, int b0, int b1), where xx can be ll, lh, hl and hh	AA_Pxx_SUB2_Pxx_ADD2	AA_Pxx_SUB2 a0, a1, upper word of dst AA_Pxx_ADD2 b0, b1, lower word of dst
	double dst = _aa_pxx_sub2_pxx_sub2 (int a0, int a1, int b0, int b1), where xx can be ll, lh, hl and hh	AA_Pxx_SUB2_Pxx_SUB2	AA_Pxx_SUB2 a0, a1, upper word of dst AA_Pxx_SUB2 b0, b1, lower word of dst
AFP	double dst = _afp(long a0, long a1, int b0, int b1)	AFP	AFPH a0, a1, upper word of dst AFPL b0, b1, lower word of dst

The modulo scheduler is one of the most effective instruction schedulers for VLIW architectures [17]. Using modulo scheduling, the iterations of a loop are executed in a pipeline: the execution of each iteration needs not wait until the completion of the preceding one. The delay between iterations is fixed in modulo scheduling and is called the Initiation Interval (II). II is determined by both the data dependence and resource constraints. The initial II can be estimated only using resource constraints and is given in the following formula

$$\text{Initial II} = \max_{i \in \text{types of computational unit}} \left\{ \left\lceil \frac{\text{Inst}(i)}{\text{CU}(i)} \right\rceil \right\} \tag{4}$$

where Inst(i) is the number of instructions that are executed in the computational units of type i and CU(i) is the number of

computational units of type *i*. Figure 13 shows a simplified modulo scheduling algorithm.

```
Modulo-Scheduling (C) {
    Construct the data dependence graph D of the linear assembly code C;
    Set the initial II according to the resource constraints of C;
    Completed = false;
    while (!Completed) {
        Set the set S of scheduled instruction to ;
        D' = D;
        Initialize the resource reservation table T
        Completed = true;
        while (D' is not empty) {
            Select a ready instruction I from D';
            Schedule I as early as possible without violating the resource constraints in T;
            if (I cannot be scheduled) {
                Completed = false; break;
            } else {
                S = S ∪ {I}; Update T; D' = D' - {I}
            }
        }
        II = II + 1;
    }
    emit S;
}
```

Fig. 13. A simplified modulo scheduling algorithm.

The characteristics of adaptive instructions raise two issues in the modulo scheduler. First, some adaptive instructions must be executed simultaneously or consecutively. For instance, the SAD adaptive operation must be implemented as follows. Four SAD_AD4 adaptive instructions must be first executed simultaneously on ALU_1, ALU_2, ALU_3 and ALU_4, and then an SAD_SUM adaptive instruction must be executed in the next clock cycle. Second, some adaptive instructions may use some of the functionality of more than one computational unit. For example, the MAS_MULL and MAS_MULH adaptive instructions use ALU_3 and the adders of ALU_1, ALU_2 and ALU_4. Accordingly, in the same clock cycle, ALU_1, ALU_2 and ALU_4 cannot be adopted to execute instructions that involve adders.

The proposed solution to the first issue is discussed first. A pseudo instruction is designed for each intrinsic to enable the adaptive instructions that correspond to an intrinsic to be scheduled as a unit. To this end, the C compiler first translates intrinsics into pseudo instructions. After a pseudo instruction has been scheduled by the modulo scheduler, it is then translated into adaptive instructions. Since a pseudo instruction represents several adaptive instructions, it may exploit different resources in different clock cycles during its execution. Hence, the update operation of the resource reservation table is modified to accommodate the more complicated resource constraints on pseudo instructions. Additionally, because adaptive pseudo instructions incur stronger resource constrains, ready adaptive pseudo instructions are given higher priority than other ready instructions in scheduling.

The proposed solution to the second issue is now presented. The representation of the resource reservation table is modified to enable an instruction to use only some of the functionality of a resource, allowing the rest of the resource to be used by another instruction. This modification in the data structure is quite straightforward. Additionally, when the MAS_MULL or MAS_MULH adaptive instruction is executed, the three ALUs, ALU_1, ALU_2 and ALU_4, cannot be adopted to execute instructions that involve adders. At the same time, if the adder in MUL is unused, an addition or a subtraction instruction can be replaced with an AoS adaptive instruction and executed on the adder of MUL to increase the parallelism. When determining the initial II in such a case, the number of instructions executed in ALU, Inst(ALU), should be adjusted accordingly. Let MAS be the number of MAS_MULx instructions; NADD be the number of instructions that do not involve the adder, and ADD be the number of addition or subtraction instructions. Then, Inst(ALU) can be defined as

$$\text{Inst(ALU)} = \text{Inst}_0\text{(ALU)} + (3\times\text{MAS}) - \min(\text{NADD}+\min(\text{ADD}, \text{MAS}), 3\times\text{MAS}), \quad (5)$$

where Inst_0(ALU) is the original number of instructions executed in ALU.

A set of benchmarks was employed to evaluate the performance of such a compiler. Each of the benchmarks has a linear assembly code for the TMS320C64x data path and one for the proposed application-specific data path. The linear assembly code for the TMS320C64x data path is scheduled by the TI compiler for TMS320C64x, while that for the proposed application-specific data path is scheduled by the proposed modulo scheduler. Table IV presents the results of the performance evaluation. In benchmarks SAD, Inverse Integer Transform (IIT) and Diagonal Half-Pixel Interpolation (DHPI), since SAD, AA and AFP adaptive instructions are successfully generated, the proposed application-specific data greatly outperformed the TMS320C64x data path. However, in benchmarks DCT and quantization, since their codes involve many multiplications, the proposed application-specific data path with a single multiplier performs no better than the TMS320C64x data path with two multipliers. Since SAD, AA and AFP adaptive instructions do not degrade the functionality of other computational units, they can generally be more effectively scheduled than MAS and AoS adaptive instructions.

TABLE IV
PERFORMANCE EVALUATION OF THE INSTRUCTION SCHEDULERS FOR THE PROPOSED APPLICATION-SPECIFIC AND TI TMS320C64X DATA PATHS

Functions \ Data paths	TMS320C64x data path	Proposed application-specific data path	
		Clock cycles	Saving ratio
16×16 SAD	64	32 (using SAD adaptive operation)	50.00%
8×8 DCT	130	138 (using MAS and AoS adaptive operations)	-6.15%
8×8 quantization	66	66 (using MAS and AoS adaptive operations)	0.00%
4×4 inverse integer transform	20	16 (using AA adaptive operation)	20.00%
Diagonal 16×16 half-pixel interpolation	224	64 (using AFP adaptive operation)	71.43%

IV. ANALYSES OF COMPUTATIONAL PERFORMANCE

A. Analyses on Low-Level Functions of Video Codecs

The computational performance of the proposed application-specific and conventional data paths, as listed in Table V, in executing low-level functions of MPEG-2, MPEG-4 and H.264/AVC video codecs, is studied [18]-[20]. Since different code generation techniques result in different computation efficiencies, the hand coding is adopted to translate the programs of the low-level functions to ensure fairness of comparison. Table VI lists the numbers of clock cycles required to execute the low-level functions by using the proposed application-specific and conventional data paths. Additionally, the scheduling of the data flows associated with the low-level functions executed in the proposed application-specific and conventional data paths are described as follows.

Motion estimation in MPEG-2, MPEG-4 and H.264/AVC video codecs and the mode decision in the H.264/AVC video codec depend strongly on the SAD operation. The proposed application-specific data path, using the SAD adaptive operation, can execute the SAD operation on 16 pixels in a clock cycle, so it computationally outperforms the conventional data paths, as presented in Table VI. The execution of special instructions causes the TI TMS320C64x data path to outperform the Motorola/Lucent StarCore140 data path in performing SAD operations.

As for DCT and IDCT in MPEG-2 and MPEG-4 video codecs, these two functions require many multiplication, addition/subtraction, shifting and packing/unpacking operations. The inner product instructions can accelerate the execution of the DCT and IDCT [5], [16]. Apart from the MUL used to implement an inner product, the proposed application-specific data path can execute the MAS adaptive operation to accelerate the determination of inner products. The AoS adaptive operation can also be executed to cooperate with the MAS adaptive operation to enable that an additional arithmetic operation can be executed in parallel. In considering the addition/subtraction, the shifting and the packing/unpacking operations of DCT and IDCT, the proposed application-specific data path with five ALUs can execute five operations simultaneously under general instructions. Therefore, the proposed application-specific data path has a similar computational performance to that of TI TMS320C64x data path, which has two multipliers and four ALUs to execute DCT and IDCT.

The MPEG-2, MPEG-4 and H.264/AVC video codecs perform quantization and inverse quantization on the transformed coefficients and the variable-length-decoded data, respectively. The MAS adaptive operation can be executed to accelerate their multiplication, addition/subtraction and shifting operations, so the proposed application-specific data path computationally outperforms the conventional data paths, as illustrated in Table VI.

With respect to the 4×4-pixel integer transform, the 4×4-pixel inverse integer transform and the 4×4-pixel Hadamard transform in H.264/AVC video codec, these three functions require many 16-bit addition, subtraction and shifting operations. Although conventional data paths perform SIMD operations to accelerate the computation of 16-bit data, they need additional operations on packing and unpacking to align these data, degrading the computational performance. The proposed application-specific data path executes the AA adaptive operation that includes alignment operations before arithmetic operations to increase the computational performance.

Motion compensation in the MPEG-2 and MPEG-4 video codecs needs diagonal half-pixel interpolation. The diagonal half-pixel interpolation uses averaging operations on four 8-bit data, so the proposed application-specific data path exploits the AFP adaptive operation to accelerate its computation, optimizing computational performance. The half-pixel interpolation of the H.264/AVC video codec depends on the six-tap filter, according to the following equation [20];

$$b = (E-5F+20G+20H-5J+K+16) >> 5$$
$$= (E+20(G+H)-5(F+J)+K+16) >> 5, \quad (6)$$

where E, F, G, H, J and K are the six input pixels and b is the output pixel. 20(G+H)-5(F+J) can be computed by executing two addition operations and one inner product. Accordingly, a six-tap filter includes one inner product, five addition operations and one shifting operation. The six-tap filter is fulfilled at a word length of 16 bits, so the AA adaptive operation, based on SIMD, can be employed to accelerate this function. Moreover, the proposed application-specific data path with five ALUs can effectively execute the addition and shifting operations of the 6-tap filter. Hence, as presented in Table VI, the proposed application-specific data path computationally outperforms the conventional data paths to perform half-pixel interpolation.

The 4×4-pixel intra prediction in the H.264/AVC video codec involves nine modes in most of which averaging operations are applied mainly to two 8-bit and four 8-bit pixels. The proposed application-specific data path can use ALU_3 and ALU_4 to average multiples of two 8-bit pixels. Simultaneously, the proposed application-specific data path combines ALU_1 and ALU_2 as a special unit to execute the AFP adaptive operation, efficiently to average multiple sets of four 8-bit pixels. Hence, the proposed application-specific data path can execute the maximum number of averaging operations in a clock cycle in most modes, minimizing the number of clock cycles required to accomplish the 4×4-pixel intra prediction.

In addition to the low-level functions in Table VI, the other low-level functions in the video codecs mainly take addition/subtraction, shifting, comparing and logic operations, which depend more strongly on the use of ALUs, for example in the de-blocking filter or in the summation of residual data and motion-compensated data. When executing these functions, the proposed application-specific data path with more ALUs can computationally outperform than the conventional data paths.

B. Analyses on FIR, IIR and FFT

Besides the low-level functions of video codecs, other functions like Finite Impulse Response (FIR) filter, Infinite

TABLE V
FEATURES OF THE PROPOSED APPLICATION-SPECIFIC AND CONVENTIONAL DATA PATHS

Features \ Data paths		Proposed application-specific data path	TMS320C64x data path	StarCore140 data path
Number of instructions parallel executed		8	8	6
Computational units		1 of 32×16-bit multiplier 4 of 40-bit ALUs 1 of 32-bit ALU 2 of 32-bit load/store units Hardware components for adaptive operations	2 of 32×16-bit multipliers 2 of 40-bit ALUs 2 of 32-bit ALUs 2 of 32-bit load/store units	4 of 16×16-bit multipliers with 40-bit accumulators 4 of 40-bit ALUs 2 of 32-bit address arithmetic units
Number of registers		32 of 32-bit registers (two clusters)	32 of 32-bit registers (two clusters)	16 of 40-bit data registers 24 of 32-bit address, offset and modulo registers
No. of ports in the register file	Read ports	RF_1: 11 of 32-bit ports and 4 of 8-bit ports RF_2: 11 of 32-bit ports and 4 of 8-bit ports	Each of two clusters having 11 of 32-bit ports 2 of 8-bit ports	16 of 40-bit ports for data registers 9 of 32-bit ports for address, offset and modulo registers
	Write ports	RF_1: 7 of 32-bit ports and 2 of 8-bit ports RF_2: 6 of 32-bit ports and 2 of 8-bit ports	Each of two clusters having 7 of 32-bit ports 2 of 8-bit ports	8 of 40-bit ports and 2 of 64-bit ports for data registers 5 of 32-bit ports for address, offset and modulo registers

Impulse Response (IIR) filter and Fast Fourier Transform (FFT) that are widely used in communication applications are also explored. Table VII lists the clock cycles of the proposed and conventional data paths for executing 16-tap FIR filter on 64 samples, fourth-order IIR filter on 64 samples and mixed-radix 64-point FFT. When computing the FIR filter, the proposed data path executes the MAS adaptive operation to increase the multiplication operations. The proposed data path can complete three multiplication and accumulation operations in a clock cycle, compared to four multiplication and accumulation operations in a clock cycle for the TI TMS320C64x and Motorola/Lucent StarCore140 data paths with two 32×16-bit and four 16×16-bit multipliers, respectively [21], [22]. Consequently, to execute functions involving a number of multiplication operations close to or exceeding the number of arithmetic and logic operations, the proposed data path underperforms the conventional data paths with two or more multipliers.

The IIR filter has the prior output feedback to result in loop-carried dependence, thus limiting the computational efficiency of the data path. When the TI TMS320C64x data path implements the fourth-order IIR filter, four clock cycles are required to execute this loop-carried dependence computation, which comprises a multiplication operation at two clock cycles, an addition operation at one clock cycle and a shifting operation at one clock cycle. Accordingly, the TI TMS320C64x data path takes four clock cycles to obtain a result of the fourth-order IIR filter due to the pipelining latency [21]. The Motorola/Lucent StarCore140 data path can perform a multiplication and accumulation operation and a shifting operation in two clock cycles to accomplish the loop-carried dependence computation. Since the maximum number of instructions parallel executed is six, the Motorola/Lucent StarCore140 data path needs three clock cycles to complete the fourth-order IIR filter [22]. By using the MAS adaptive operation combining multiplication and addition as a multiplication-accumulation operation, the proposed data path can finish the computation of the fourth-order IIR filter in three clock cycles, exceeding and equaling the requirements of the TI TMS320C64x and Motorola/Lucent StarCore140 data paths, respectively.

In computing the FFT that requires many 16-bit addition/subtraction operations, the proposed data path can employ the AA adaptive operations to process the real and imaginary parts in the 16-bit addition and subtraction operations. Additionally, the MAS adaptive operations are used to implement the inner products for complex multiplications of the FFT. Hence, the computation efficiency of the proposed data path is superior to that of the TI TMS320C64x and Motorola/Lucent StarCore140 data paths in executing FFT [21], [22].

TABLE VI
NUMBERS OF CLOCK CYCLES REQUIRED BY THE PROPOSED APPLICATION-SPECIFIC AND CONVENTIONAL DATA PATHS

Functions \ Data Paths	Proposed application-specific data path	TMS320C64x data path	StarCore140 data path
16×16 SAD	16	32	256
8×8 DCT	130	126	231
8×8 IDCT	150	154	231
8×8 quantization	54	58	62
8×8 inverse quantization	62	66	90
4×4 integer transform / 4×4 inverse integer transform	12	16	20
4×4 Hadamard transform	10	12	16
Diagonal 16×16 half-pixel interpolation for MPEG-2 and MPEG-4	37	169	288
4×4 half-pixel interpolation for H.264/AVC	23	34	36

TABLE VII
NUMBERS OF CLOCK CYCLES REQUIRED BY THE PROPOSED APPLICATION-SPECIFIC AND CONVENTIONAL DATA PATHS COMPUTING FIR FILTER, IIR FILTER AND FFT

Functions \ Data paths	Proposed application-specific data path	TMS320C64x data path	StarCore140 data path
16-tap FIR filter on 64 samples	576	447	524
Fourth-order IIR filter on 64 samples	213	277	225
Mixed-radix 64-point FFT	294	304	388

V. ANALYSES OF COMPUTATIONAL POWER AND HARDWARE COST

Based on the TSMC 0.18μm CMOS cell library [23], the proposed application-specific data path, including five ALUs, an MUL, two load/store units, a register file, and hardware

components and interconnections for adaptive operations, is designed and implemented using the Cadence tool. Figure 14 shows the layout of the proposed application-specific data path, of which features are listed in Table VIII. The proposed application-specific data path has five ALUs each of which occupies an average of 4.90% of the overall data path. The register file occupies 53.31%. The MUL and two load/store units occupy 14.40% and 2.72%, respectively, and the hardware components and interconnections used for adaptive operations occupy 5.06%. The adaptive operations adapt data flows of the data path, so most of the hardware components are multiplexers. Circuits such as the Pass Transistor Logic (PTL) circuit or the Complementary Pass-transistor Logic (CPL) circuit used to implement the multiplexers may further reduce the area of the hardware components required to perform adaptive operations.

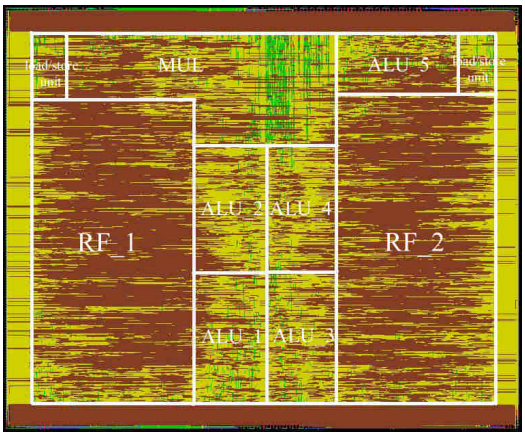

Fig. 14. Layout of the proposed application-specific data path.

TABLE VIII
FEATURES OF THE PROPOSED APPLICATION-SPECIFIC DATA PATH

Features		Proposed application-specific data path
Technology		TSMC 0.18μm CMOS
Design scheme		SAGE-X™ standard cell library
Supply voltage		1.8 V
Critical delay		6.92 ns
Area (1.77mm×1.45mm)	Five ALUs	0.63 mm^2 (24.51%)
	An MUL	0.37 mm^2 (14.40%)
	Hardware components and interconnections for adaptive operations	0.13 mm^2 (5.06%)
	Two load/store units	0.07 mm^2 (2.72%)
	Register file	1.37 mm^2 (53.31%)
	Total	2.57 mm^2 (100%)

The critical delay of the computational units is simulated by using the NanoSim tool, which occurs in an 8×8-bit multiplication, which is the fundamental multiplication operation. However, the eight 8×8-bit multipliers required by the MUL use compressors or counters to add their partial products in parallel and thus reduce the computational delay, so the critical delay in the proposed application-specific data path is at the 8×8-bit array multiplier for the MAS adaptive operation, as presented in Fig. 8. Figure 8(a) shows that an 8×8-bit array multiplier is composed of 56 1-bit full adders. Since a 1-bit full adder generates the sum and carry with different delays, the longer and shorter paths of a 1-bit full adder can be connected to the shorter and longer paths of the next 1-bit full adder(s), respectively, to minimize the critical delay [24]. Based on such hardware, the critical delay of the 8×8-bit array multiplier with some logic gates and multiplexers is 5.68ns. The control circuits of the register file, which decode the addresses of the registers, are established in the instruction path [13]. Hence, the register file only includes storage cells and tri-state buffers for reading and writing [25], [26]. Given such a design, the access time of the register file is 1.24ns. Summing the delays of the computational units and register file reveals that the critical delay of the proposed application-specific data path is 6.92ns (5.68ns+1.24ns).

The critical delay of the data path depends on the architectures and implementation manners of computational units. Since detailed information on the conventional data paths is not publicly available, the proposed adaptive mechanism cannot be directly implemented in the conventional optimized data paths. However, the high-level architectures of ALUs and MUL in the proposed application-specific data path are designed to be the same as those of the conventional data path. Additionally, hardware components of the proposed application-specific and conventional data paths are implemented using the same cell library to do fair comparison. Table IX lists the hardware costs, the computational power and the ratios of computational power to hardware costs required by the proposed application-specific and conventional data paths. The normalized hardware cost of the proposed application-specific data path is assumed to be unity, and the hardware costs of the conventional data paths are divided using that of the proposed application-specific data path to determine the normalized hardware costs. The proposed application-specific data path includes five ALUs, a multiplier, two load/store units, and hardware components and interconnections for adaptive operations, whereas the TI TMS320C64x data path comprises four ALUs, two multipliers and two load/store units. According to Table VIII, an MUL consumes a larger hardware cost than that of an ALU such that the TI TMS320C64x data path with two MULs is more complicated than the proposed application-specific data path. Four 16×16-bit multipliers can be formed as two 32×16-bit multipliers so the hardware complexity of the computational units in the Motorola/Lucent StarCore140 data path is similar to that in the TI TMS320C64x data path. From Table VIII, the hardware cost of the register file in the proposed application-specific data path is 0.53 times the total hardware cost of the data path. According to Table V, the register files of the conventional data paths can be implemented using the same design, and their normalized hardware costs are listed in Table IX.

When the circuits are designed to be similar to those in the computational units of the proposed application-specific data path, the computational units of the conventional data paths exhibit critical delays in 8×8-bit multiplication using 3:2, 4:2, 5:2 and 6:2 compressors and half adders, of around 5.4ns when

TABLE IX
HARDWARE COST, COMPUTATIONAL POWER, AND COMPUTATIONAL POWER OVER HARDWARE COST OF THE PROPOSED APPLICATION-SPECIFIC AND CONVENTIONAL DATA PATHS

Features \ Data paths		Proposed application-specific data path	TMS320C64x data path	StarCore140 data path
Hardware cost (Normalized values)	Computational units	0.47	0.54	0.56
	Register file	0.53	0.53	0.38
Computational power (Normalized values)	16×16 SAD	1.00	0.52	0.07
	8×8 DCT	1.00	1.07	0.59
	8×8 IDCT	1.00	1.01	0.68
	8×8 quantization	1.00	0.97	0.91
	8×8 inverse quantization	1.00	0.98	0.72
	4×4 integer transform / 4×4 inverse integer transform	1.00	0.78	0.63
	4×4 Hadamard transform	1.00	0.87	0.52
	Diagonal 16×16 half-pixel interpolation for MPEG-2 and MPEG-4	1.00	0.23	0.12
	4×4 half-pixel interpolation for H.264/AVC	1.00	0.70	0.67
	4×4 intra prediction at mode 5	1.00	0.57	0.33
Computational power/hardware cost	16×16 SAD	1.00	0.49	0.07
	8×8 DCT	1.00	1.00	0.63
	8×8 IDCT	1.00	0.94	0.72
	8×8 quantization	1.00	0.91	0.97
	8×8 inverse quantization	1.00	0.92	0.77
	4×4 integer transform / 4×4 inverse integer transform	1.00	0.73	0.67
	4×4 Hadamard transform	1.00	0.81	0.55
	Diagonal 16×16 half-pixel interpolation for MPEG-2 and MPEG-4	1.00	0.21	0.13
	4×4 half-pixel interpolation for H.264/AVC	1.00	0.65	0.71
	4×4 intra prediction at mode 5	1.00	0.53	0.35

the TSMC 0.18μm CMOS cell library is used. The critical delay of the conventional data paths is about 6.64ns (5.4+1.24), including the access time of the register file. According to the data in Table VI, the computational power of the proposed adaptive and conventional data paths executing a function is given by

$$\text{computational power} = \frac{1}{\text{number of clock cycles} \times \text{critical delay}}. \quad (7)$$

Additionally, Table IX presents the normalized values of computational power. The proposed application-specific data path takes a similar number of clock cycles to that required by the TI TMS320C64x data path to perform the DCT and IDCT. However, due to a longer critical delay, the proposed application-specific data path has slightly less computational power than the TI TMS320C64x data path. With respect to the other low-level functions, by effectively reducing the numbers of clock cycles required, the proposed application-specific data path has a higher computational power than the TI TMS320C64x data path. As compared to the Motorola/Lucent StarCore140 data path, the proposed application-specific data path shows a higher computational power in executing these low-level functions. Finally, the normalized computational power is divided by the normalized hardware cost of the data path as listed in Table IX. When executing the DCT demanding extensive multiplications, the proposed application-specific data path has the same ratio of computational power to hardware cost as the TI TMS320C64x data path. In executing the other low-level functions, the proposed application-specific data path shows the highest ratio of computational power to hardware cost than the conventional data paths. Therefore, the application-specific data path proposed herein is computationally efficient and cost-effective in realizing multi-standard video codecs.

TABLE X
PERFORMANCE OF THE PROPOSED APPLICATION-SPECIFIC AND TI TMS320C64X DATA PATHS EXECUTING THE LOW-LEVEL FUNCTIONS OF THE MPEG-2 VIDEO ENCODER AT 30 352×288-PIXEL FRAMES.

Functions \ Data paths			Proposed application-specific data path	TMS320C64x data path
Motion estimation (three-step search)	Cycles/block	P frame	654	1,086
		B frame	1,308	2,172
	Number of blocks	P frame	3,564	3,564
		B frame	7,128	7,128
	Total clock cycles		11,654,280	19,352,520
DCT	Cycles/block		130	126
	Number of blocks		47,520	47,520
	Total clock cycles		6,177,600	5,987,520
IDCT	Cycles/block		150	154
	Number of blocks		47,520	47,520
	Total clock cycles		7,128,000	7,318,080
Quantization	Cycles/block		54	58
	Number of blocks		47,520	47,520
	Total clock cycles		2,566,080	2,756,160
Inverse quantization	Cycles/block		62	66
	Number of blocks		47,520	47,520
	Total clock cycles		2,946,240	3,136,320
Interpolation	Vertical / horizontal	Cycles/block	68	68
		Number of blocks	10,692	10,692
		Total clock cycles	727,056	727,056
	Diagonal	Cycles/block	37	169
		Number of blocks	10,692	10,692
		Total clock cycles	395,604	1,806,948
Total number of clock cycles			31,594,860	41,084,604
Estimated run time (μs)			218,636	272,802

In the proposed and TI dual-core hardware platforms, 30 352×288-pixel frames of the Foreman video sequence are processed by the proposed application-specific and TI TMS320C64x data paths that focus on executing low-level functions. The operations related to memory access, high-level decisions, variable length coding/decoding functions and multiplexing/demultiplexing the data stream are performed by the CPU. Table X lists the clock cycles and run time required for

motion estimation, DCT/IDCT, quantization/inverse quantization and interpolation in the MPEG-2 video encoders where the three-step search scheme is used to determine the motion vector. To implement these six functions in 30 frames with the coding condition of (1 intra-frame, 3 predictive and 6 bi-directional inter-frames), the proposed application-specific data path can achieve a 23.10% reduction compared to the TI TMS320C64x data path in the clock cycles required to implement these six functions. When the critical delays of 6.92ns and 6.64ns taken by the proposed and TI TMS320C64x data paths are considered, respectively, the proposed application-specific data path at 144MHz needs 218,636μs to execute these six functions in 30 352×288-pixel Foreman frames compared to 272,802μs for the TI TMS320C64x data path at 150MHz. Hence, the proposed application-specific data path conserves 19.86% of computation time requested by the TI TMS320C64x data path.

TABLE XI
PERFORMANCE OF THE PROPOSED APPLICATION-SPECIFIC AND TI TMS320C64X DATA PATHS EXECUTING THE LOW-LEVEL FUNCTIONS OF THE H.264/AVC VIDEO DECODER AT 30 352×288-PIXEL FRAMES

Functions \ Data paths			Proposed application-specific data path	TMS320C64x data path
Intra_4×4	mode 0 and 1	Cycles/block	3	3
		Number of blocks	2,005	2,005
		Total clock cycles	6,015	6,015
	mode 2	Cycles/block	8	8
		Number of blocks	2,054	2,054
		Total clock cycles	16,432	16,432
	mode 4, 5, 6, 7 and 8	Cycles/block	6	11
		Number of blocks	5,797	5,797
		Total clock cycles	34,782	63,767
Intra_16×16	mode 0 and 1	Cycles/block	17	17
		Number of blocks	68	68
		Total clock cycles	1,156	1,156
	mode 2	Cycles/block	24	27
		Number of blocks	58	58
		Total clock cycles	1,392	1,566
	mode 3	Cycles/block	268	268
		Number of blocks	49	49
		Total clock cycles	13,132	13,132
Intra prediction of chroma block	mode 0 and 1	Cycles/block	9	9
		Number of blocks	1,604	1,604
		Total clock cycles	14,436	14,436
	mode 2	Cycles/block	15	17
		Number of blocks	1,006	1,006
		Total clock cycles	15,090	17,102
	mode 3	Cycles/block	67	67
		Number of blocks	238	238
		Total clock cycles	15,946	15,946
Inverse integer transform		Cycles/block	12	16
		Number of blocks	253,216	253,216
		Total clock cycles	3,038,592	4,051,456
Hadamard transform		Cycles/block	10	12
		Number of blocks	175	175
		Total clock cycles	1,750	2,100
Half-pixel and quarter-pixel interpolation	6-tap filter	Cycles/block	23	34
		Number of blocks	202,570	202,570
		Total clock cycles	4,659,110	6,887,380
	Bilinear interpolation	Cycles/block	6	6
		Number of blocks	70,163	70,163
		Total clock cycles	420,978	420,978
Total number of clock cycles			8,238,811	11,511,466
Estimated run time (μs)			57,013	76,436

Table XI lists the number of clock cycles required by the proposed application-specific and TI TMS320C64x data paths to implement the intra prediction, inverse transform and half-pixel and quarter-pixel interpolation of the H.264/AVC video decoder. The proposed application-specific data path performs the adaptive operations to speed up these low-level functions. Accordingly, the proposed application-specific data path can conserve 28.43% of the clock cycles required by the TI TMS320C64x data path to execute these functions. The proposed application-specific and TI TMS320C64x data paths take 57,013μs and 76,436μs, respectively, to implement these low-level functions of the H.264/AVC decoder and thus 25.41% computation time is conserved in the proposed application-specific data path. Therefore, the proposed application-specific data path requires less computation time than the TI TMS320C64x data path to execute the low-level functions of video codecs.

VI. CONCLUSION

This work proposes an adaptive mechanism for increasing the flexibility of data delivery and configuring computational units in a VLIW data path, so the hardware components and input/output ports of computational units can be effectively used to improve computational performance. Based on the characteristics of the low-level functions in MPEG-2, MPEG-4 and H.264/AVC video codecs, the proposed application-specific data path with eight computational units is designed to perform five adaptive operations, and is established to perform computationally at a level equal to or better than the conventional data paths, in executing the low-level functions. By using these adaptive operations, the proposed application-specific data path can conserve 23.10% and 28.43% of clock cycles demanded by the TI TMS320C64x data path to execute the low-level functions of the MPEG-2 video encoder and H.264/AVC video decoder, respectively. The TSMC 0.18μm CMOS cell library is used to implement the proposed application-specific data path, to achieve the highest ratio of computational power to hardware cost than is achieved by conventional data paths. Therefore, the proposed adaptive mechanism can be applied to a wide range of DSP architectures to improve their computational efficiencies in particular application domains.

ACKNOWLEDGEMENT

Valuable comments and suggestion from reviewers are highly appreciated. Additionally, the authors would like to thank Prof. John Lach, Electrical and Computer Engineering Department, University of Virginia, USA, for his valuable suggestion on the adaptive architecture design. Dr. B. J. Sheu, Nassda Corp., USA, commending on VLSI architecture design is thanked very much. The Chip Implementation Center, HsinChu, Taiwan, providing the service of CAD tools is also highly appreciated.

REFERENCES

[1] J. Xin, C. W. Lin and M. T. Sun, "Digital video transcoding," *Proceedings of the IEEE*, vol. 93, issue 1, pp. 84-97, Jan. 2005.

[2] T.-H. Chiang and D. Anastassiou, "Hierarchical coding of digital television," *IEEE Communications Magazine*, vol. 32, issue 5, pp. 38-45, May 1994.
[3] P. Pirsch and H. J. Stolberg, "VLSI implementations of image and video multimedia processing systems," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 8, issue 7, pp. 878-891, Nov. 1998.
[4] *TMS320DM6443 Digital Media System-on-Chip*, Texas Instruments, June 2006.
[5] M. Berekovic, H. Stolberg and P. Pirsch, "Multicore system-on-chip architecture for MPEG-4 streaming video," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 688-699, Aug. 2002.
[6] *MSC8101 User's Guide*, Motorola, June 2001.
[7] *TMS320C64x Technical Overview*, Texas Instruments, Sep. 2000.
[8] *SC140 DSP Core Reference Manual*, Motorola, April 2001.
[9] T. Kumura, M. Ikekawa, M. Yoshida and I. Kuroda, "VLIW DSP for mobile application," *IEEE Signal Processing Mag*., pp. 10-21, July 2002.
[10] L. H. Chen, O. T.-C. Chen, T. Y. Wang and C. L. Wang, "An adaptive DSP processor for high-efficiency computing MPEG-4 video encoder," *Proc. of IEEE Int. Symposium on Circuits and Systems*, vol. 2, pp. 23-26, May 2004.
[11] L. H. Chen, W. L. Liu, O. T.-C. Chen and R. L. Ma, "A reconfigurable digital signal processor architecture for high-efficiency MPEG-4 video encoding," *Proc. of IEEE Int. Conf. on Multimedia and Expo*, vol. 2, pp. 165-168, Aug. 2002.
[12] L. H. Chen and O. T.-C. Chen, "Design Methodology of a hardware-efficiency VLIW architecture with highly adaptable data path," *Proc. of IEEE 48th Midwest Symposium on Circuits and Systems*, pp. 1223-1226, Aug. 2005.
[13] P. Lapsley, J. Bier, A. Shoham and E. Lee, *DSP Processor Fundamentals: Architecture and Features*, New York: IEEE Press, 1997.
[14] O. T.-C. Chen, "Motion estimation using one-dimensional gradient descent search," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, no. 4, pp. 608-616, June 2000.
[15] *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, Texas Instruments, June. 2005.
[16] *PNX15xx Series Data Book: Connected Media Processor*, Philips Semiconductors, Dec. 2004.
[17] B. R. Rau, "Iterative modulo scheduling," *Int. J. Parallel Program*., vol. 24, pp. 3-64, 1996.
[18] "Generic coding of moving pictures and associated audio information: Video," Int. Standards Org./Int. Electrotech. Comm. (ISO/IEC) and Int. Telecommun. Union-Telecommun. (ITU-T), 13 818-2 and ITU-T Recommendation H.262 (MPEG-2), 1996.
[19] "Coding of audio-visual objects – Part 2: Visual," Int. Standards Org./Int. Electrotech. Comm. (ISO/IEC), ISO/IEC 14 496-2, 1999.
[20] Joint Video Team, Int. Telecommun. Union-Telecommun. (ITU-T) and Int. Standards Org./Int. Electrotech. Comm. (ISO/IEC), "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification," *ITU-T Recommendation H.264 and ISO/IEC 14496-10 AVC*, May 2003.
[21] *Signal Processing Examples Using TMS320C64x Digital Signal Processing Library (DSPLIB)*, Texas Instruments, Sept. 2003.
[22] *Speech Coder Filters Using the StarCore™ SC140/SC1400 Cores*, Freescale Semiconductor, April 2005.
[23] *TSMC 0.18μm Process 1.8 Volt SAGE-X™ Standard Cell Library Databook*, Artisan components, Sept. 2003.
[24] P. F. Stelling, C. U. Martel, V. G. Oklobdzija and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Trans. on Computer*, vol. 47, iss. 3, pp. 273-285, March 1998.
[25] W. Hwang, R. V. Joshi and W. H. Henkels, "A 500-MHz, 32-word×64-bit, eight-port self-resetting CMOS register file," *IEEE J. Solid-State Circuits*, vol. 34, issue 1 pp. 56-67, Jan. 1999.
[26] R. Kumar, "Scalable register file organization for a multiple issue microprocessor," *IEE Electronics Letters*, vol. 30, issue 1, pp. 16-17, Jan. 1994.

Oscal T.-C. Chen was born in Taiwan in 1965. He received the B.S. degree in electrical engineering from National Taiwan University in 1987, M.S. and Ph.D. degrees in electrical engineering from University of Southern California, Los Angeles, USA, in 1990 and 1994, respectively.

Dr. Chen worked in Computer Processor Architecture Department of Computer Communication & Research Labs. (CCL), Industrial Technology Research Institute (ITRI), for serving a system design engineer, project leader, and section chief from 1994 to 1995. He had contributed significantly to many industrial applications including the fuzzy chip, neural networks, speech recognition system, and digital signal processor. Dr. Chen was an Associate Professor in Department of Electrical Engineering, National Chung Cheng University (NCCU), Chiayi, Taiwan from Sept. 1995 to Aug. 2003. Additionally, Dr. Chen served a Director, Academic Development Division, Office of Research and Development, NCCU from July 2001 to July 2004, and a Director, Technology Transfer Center, NCCU, from July 2003 to July 2004. After Aug. 2003, he becomes a Professor in Department of Electrical Engineering, NCCU. Currently, he also serves a technical consultant in Electronics and Optoelectronics Research Labs. ITRI, Hsinchu, Taiwan.

In the technical society, he was an Associate Editor of IEEE Circuits & Devices Magazine from Aug. 2003, and a founding member of the multimedia systems and applications technical committee of IEEE Circuits and Systems Society. Dr. Chen participates the Technical Program Committee of many IEEE International Conferences and Symposiums. Dr. Chen was the co-recipient of the Best Paper Award of IEEE Transactions on VLSI Systems in 1995. His research interests include video/audio processing, DSP processors, VLSI systems, RF IC, microsensors and communication systems. He is a member of IEEE, and a life member of Chinese Fuzzy Systems Association.

Li-Hsun Chen was born in Taiwan, R.O.C., in 1976. He recieved the B.S. degree in electrical engineering from National Chung Cheng University in 1998. Currently, he is working toward the Ph. D. degree in electrical engineering. His research interests include digital filter design, reconfigurable architectures, DSP processors and VLSI systems.

Nai-Wei Lin received the B.S. and M.S. degrees in Electrical Engineering from Tatung University, Taiwan in 1981 and 1983. He received the Ph.D. degree in Computer Science from University of Arizona, U.S.A. in 1993. He is currently an Associate Professor in the Department of Computer Science and Information Engineering, National Chung Cheng University, Taiwan. His research interests include programming languages, compilers, software testing, and software tools.

Chih-Chang Chen was born in Taiwan, R.O.C., in 1972. He received the B.S. degree in electronic engineering from Feng Chia University, Taiwan, R.O.C., in 1994. He is currently working toward the PH.D. degree at electrical engineering from National Chung Cheng University, Taiwan, R.O.C.. His research interests include image processing, image and video compression, especially in ROI encoding of the JPEG 2000 and memory reduction of the H.264/AVC video codec.