Complex Streamed Instructions: Introduction and Initial Evaluation

Stamatis Vassiliadis Ben Juurlink Edwin Hakkennes
Computer Engineering, Electrical Engineering Department
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract

An architectural paradigm intended to improve the performance of streaming operations is introduced. The proposed complex streamed instructions *perform setting, controlling and executing vector operations simultaneously. That is, each of the instructions has the capability of sectioning a vector in addition to performing complex memory accesses and SIMD execution. To provide an initial validation, a video compression application is considered. We identify 5 streamed operations and perform simulations using the SimpleScalar toolset. The experiments show the following: the cycle count diminishes by a factor of 3.1, the number of executed instructions reduces by a factor of 3.0 to 3.2, and the CPI stays about the same. These results show that substantial performance improvements can be expected when complex streamed instructions are employed.*

1. Introduction

Multimedia as well as many other applications often apply the same computation on an entire stream of data, which make them well suited for a Single Instruction Multiple Data (SIMD) architectural paradigm. Hardware manufacturers recognized this property and added SIMD-like instructions targeted to multimedia applications to their instruction sets. Examples include Intel's MMX [12], HP's PA-RISC MAX-2 [9] and SUN's UltraSparc VIS [15]. Another feature common to multimedia applications is that their performance is often limited by memory bandwidth. Data is often read only once, which implies that temporal locality may not be exploited. Furthermore, accessing data for streaming operation may follow a sub-matrix pattern, implying a complex addressing mechanism. Finally, it can be observed that some complex SIMD operations could be captured with a single instruction not requiring prohibitive hardware implementations (see, e.g., [16, 6]). In this paper we propose an architecture that incorporates all of the above observations in a single instruction. We call this approach *complex streamed instructions*.

These instructions provide a number of advantages. First, they increase the amount of data parallelism that can be exploited. Second, because streamed instructions contain all necessary information to complete the entire operation (like base address, length and stride), data can be fetched before it is needed. Another advantage that they completely eliminate the need for sectioning and branching, since they replace the instructions needed for vector sectioning by a hardware mechanism. Finally, a fast execution time for the operation is achieved as multiple simple operations are combined to perform a more complex function in less machine cycles with no additional cycle time penalties.

We investigate the applicability and quantify the performance potential of complex streamed instructions using the MPEG2 encoder of the MPEG Software Simulation Group [10] as a benchmark. Our initial evaluation shows that streamed instructions can be applied in many circumstances and that a significant performance improvement can be obtained. In particular, we show the following:

- Compared to the baseline superscalar architecture, the number of execution cycles is reduced by a factor of 3.1.
- The number of executed instructions improves by a factor of 3.0 to 3.2.
- Even though streamed instructions may many cycles, the CPI remains about the same.

These results strongly suggest that complex streamed instructions are promising in delivering improved performance for applications requiring streaming data operations.

This paper is organized as follows. In Section 2 we consider one operation, the *sum of absolute differences* (SAD), and devise a streamed instruction and possible implementation for it. In Section 3 experimental results are presented that show the performance gain resulting from adding the streamed SAD instruction to the instruction set architecture as well as other streamed instructions. Section 4 describes related work and Section 5 contains our conclusions.

2. Complex Streamed Instructions

As a typical example of an application with many streamed operations, consider video compression. This application consists of many stages, but a large part of the execution time is needed for motion estimation. This stage compares a block in the current frame with blocks in the reference frame to find the block resembling the current block the most. Because searching all blocks is too expensive and because movements are usually small, the search is often limited to blocks in proximity of the current block.

The most popular metric for determining the resemblance of two blocks is the *sum of absolute differences* (SAD). For 16×16 blocks, it is defined as follows:

$$SAD(x, y, r, s) = \sum_{i=0}^{i=15} \sum_{j=0}^{j=15} |A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)}|, \quad (1)$$

where (x, y) is the position of the current block A and (r, s) denotes the motion vector, i.e. the displacement of the current block relative to the reference block B. The corresponding C code, taken from the MPEG Software Simulation Group [10], is depicted in Figure 1.

An operation like the SAD operates on an entire *stream of data*. In general, the stream elements do not have to be stored consecutively in memory. In this case, the row elements are stored consecutively, but to locate the first element in the next row, the width `lx` of the frame is also needed. This implies that accesses to consecutive elements in the same row will exhibit good spatial locality, but when the next row is accessed, a cache miss is likely to occur. That is one of the reasons why we propose complex streamed instructions that contain all necessary information to complete the entire operation, so that the hardware can start fetching the next row before it is needed. Furthermore, in addition to operating in a vector-like fashion, stream operations may require multiple simple operations to be performed on each element. In our proposal these operations are performed by one complex instruction.

```
s = 0;
for (j=0; j<h; j++){
  if ((v = p1[0]-p2[0])<0)   v = -v; s+= v;
  if ((v = p1[1]-p2[1])<0)   v = -v; s+= v;
  ...
  if ((v = p1[15]-p2[15])<0) v = -v; s+= v;

  if (s >= distlim) break;

  p1+= lx;
  p2+= lx;
}
```

Figure 1: C code for sum of absolute differences.

```
$L465:
   lbu  $3,0($4)     # $3 = p1[0]
   lbu  $2,0($12)    # $2 = p2[0]
   #nop              # delay slot
   subu $5,$3,$2     # v = p1[0]-p2[0]
   bgez $5,$L466     # if (v>=0)
                     # skip next instr.
   subu $5,$0,$5     # v = -v
$L466:
   lbu  $3,1($4)     # $3 = p1[1]
   lbu  $2,1($12)    # $2 = p2[1]
   addu $8,$8,$5     # s += v
   subu $5,$3,$2     # v = p1[1]-p2[1]
   bgez $5,$L467     # ...
   subu $5,$0,$5     #
$L467:
   ...
$L481:
   addu $8,$8,$5     # s += v
   slt  $2,$8,$10    # if (s>=distlim)
   beq  $2,$0,$L484  #   break;
   addu $4,$4,$6     # p1 += lx
   addu $12,$12,$6   # p2 += lx
   addu $13,$13,1    # j++
   slt  $2,$13,$15   # if (j<h)
   bne  $2,$0,$L465  #   repeat
```

Figure 2: Assembly code for sum of absolute differences using conventional instructions.

Figure 2 shows the relevant parts of the assembly code generated from the sum of absolute differences routine. Note that the accumulation occurs after loading the next pixels in order to fill the load delay slot. In addition, we remark that in order to perform the entire operation on two 16×16 blocks, 1664 instructions are required (6 per pixel plus 8 for controlling the loop).

In order to improve the execution of such streamed operations, we propose instructions that shorten the code in terms of instructions to be executed and that improve substantially the execution time of applications requiring streamed operations. These goals are achieved by including complex streamed instruction that generally operate as follows:

- First, identify streamed operations that can be implemented by a complex instruction with non-prohibitive hardware requirements (in terms of area and delay).
- Augment the complex instruction definition to include sectioning, loading/storing and branching.

In the next sections, these concepts are illustrated using the SAD as an example.

2.1. An Efficient Implementation of the SAD

To establish that the SAD operation can be implemented with non-prohibitive hardware requirements, we first note that a direct approach consists of the following steps:

- Compute $A_i - B_i$ for all 16x16 pixels in the two blocks A and B.
- Take the absolute values of these differences.
- Accumulate all 16x16 absolute values.

A straightforward implementation of this approach requires 4 cycles[1] per row: 1 cycle to compute $A_i - B_i$ for all 16 pixel pairs, 1 cycle for possibly negating the result, and two cycles to accumulate the absolute values. Alternatively, one can compute the absolute value in 1 cycle by computing $A_i - B_i$ and $B_i - A_i$ in parallel, and by selecting one of them using the sign-bit of $A_i - B_i$, but this requires two full subtractors and a multiplexor per pair of pixels.

To speed up the computation without adding much hardware, we eliminate the absolute-difference operation. In general this is not possible, because $\sum |A_i - B_i| \neq |\sum (A_i - B_i)|$. Our solution is as follows. We first observe that $|A_i - B_i| = \max(A_i, B_i) - \min(A_i, B_i)$. So, instead of actually computing the absolute difference, we could pass $\max(A_i, B_i)$ and $-\min(A_i, B_i)$ to the accumulate stage. Determining and negating $\min(A_i, B_i)$, however, requires more than 1 cycle. But if we instead pass its inverted value to the accumulate stage and let the accumulate stage correct for this, only 1 extra cycle is needed in addition to the 2 cycles needed for the accumulate.

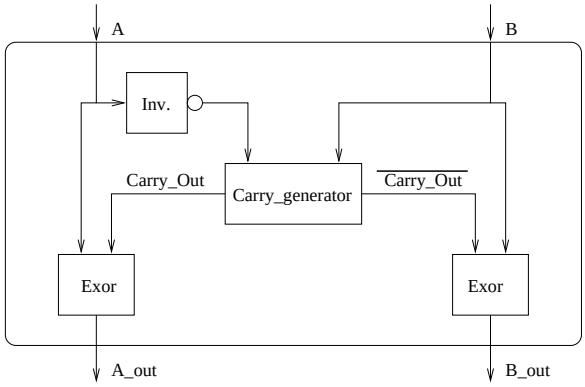
The following unit computes the SAD of 16 pixel pairs in parallel, where each pixel is represented by an unsigned byte.

Step 1.a (Determine the smallest operand): This is done by inverting one of the operands and by computing the carry-out which would arise from adding both operands. Notice that computing the carry-out is cheaper than performing the actual addition.

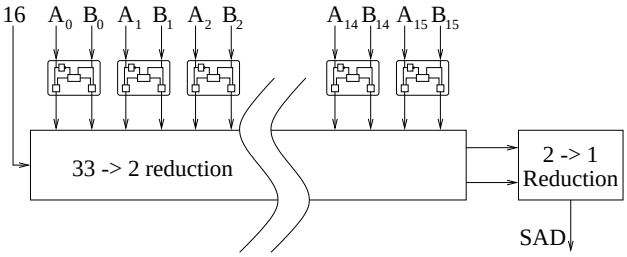
Step 1.b (Invert the smallest operand): The smallest operand is inverted, and both the inverted smallest and the largest value are passed to an adder-tree.

A graphical representation of these steps is depicted in Figure 3(a). The above two steps result in 32 8-bit values, which are passed to an adder-tree. In addition, a correction term of 16 is passed to the adder tree, to undo the effect of not passing the negated value of the smallest operand but its inverse. Then, the following steps are applied.

[1] Of course, if we make the cycle-time long enough, the entire operation can be performed in 1 cycle. We assume here that the cycle-time is comparable to that of a two-cycle 32-bit multiply, i.e., the cycle-time is just long enough to accommodate a 32-bit multiply in two cycles.



(a) First step of the SAD unit



(b) 16×1 SAD unit

Figure 3: Graphical representation of the 3-cycle SAD unit.

Step 2 (33-to-2 reduction): The 33 rows are reduced to 2 using a counter scheme, see, e.g., [18, 5, 17].

Step 3 (2-to-1 reduction): The two remaining rows are added together. The carry-out has to be discarded.

A schematic diagram for the SAD unit is given in Figure 3(b). A more thorough explanation and the correctness proof are given in [16]. Assuming that the cycle-time is comparable to that of a 2-cycle 32-bit multiply, the following can be stated about the complexity of the SAD unit:

- Step (1.a) and (1.b) require 16 carry generators and 32 XORs. This is of the same or less complexity than setting up the multiplier. Together, these steps require at most 1 cycle.
- The second step can be done using a total of 260 Carry Save Adders in 8 levels, and requires 1 cycle.
- The 2-1 addition of partial products is of less complexity than the equivalent reduction for a 32-bit multiplier.

The overall conclusion is that the hardware required to compute the SAD of two 16×1 blocks is comparable to

that of a multiplier, and that it can be performed in 3 cycles. We also note that the approach can be extended to compute the SAD of two 16×16 blocks in 4 cycles. Thus, we have introduced a flexible design that is scalable with appropriate hardware additions and with implementations that are not prohibitive in terms of area and delay.

2.2. The Streamed SAD Instruction

The previous discussion answers the first constraint of our proposal. The second is satisfied by the following. For illustration, we first consider a VIS-like [15] instruction set that contains a vector-like SAD instruction (v-SAD) that operates on two 64-bit FP registers. Then, the code depicted in Figure 4 computes the SAD of two 16×16 blocks.

Although a v-SAD instruction significantly reduces the code size, still 304 instructions are needed to process an entire 16×16 block. In fact, out of 19 instructions in each loop iteration, only two are actually used for computation. Eleven instructions can be classified as overhead; they are needed for aligning data, advancing pointers and loop control. Furthermore, the code is dependent upon the length of

```
   add       $2,$0,$0    # s = 0
   add       $8,$0,$0    # j = 0
   add       $13,$0,16   # $13 = 16
   la        $4, A
   la        $5, B
$L50:
   alignaddr $2,0($4)    # $2 = $4&~7
# load 1st and 2nd 8-byte block of A
   l.d       $f0,0($2)
   l.d       $f2,8($2)
   l.d       $f4,16($2)
   alignaddata $f0,$f0,$f2
   alignaddata $f2,$f2,$f4
   alignaddr $2,0($5)
# load 1st and 2nd 8-byte block of B
   l.d       $f4,0($2)
   l.d       $f6,8($2)
   l.d       $f8,16($2)
   alignaddata $f4,$f4,$f6
   alignaddata $f6,$f6,$f8
   v-sad     $2,$f0,$f4
# perform SAD operation and accumulate
# in integer register $2
   v-sad     $2,$f2,$f6
# advance pointers to next rows
   addu      $4,$4,$6
   addu      $5,$5,$6
   addu      $8,$8,1     # j++
   slt       $10,$8,$13  # if (j<16)
   bne       $10,$0,$L50 #   goto $L50
```

Figure 4: VIS-like code for sum of absolute differences.

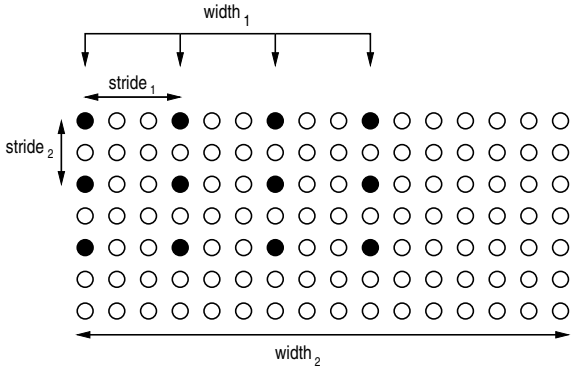


Figure 5: Operands required by the s-SAD instruction.

the vector operation (8 bytes in this case). We refer to this length as the machine's *section size*.

The instruction we propose, called the streamed SAD (s-SAD), replaces the entire loop by a single instruction. This is possible because all computations are rather simple and independent, thus they create no special control design problems.

We do not assume that the stream data is stored consecutively in memory. Figure 5 illustrates the most general case. There is a stride in the horizontal (stride_1) and vertical dimension (stride_2). Furthermore, in addition to the stream length, the instruction also needs to know the number of stream elements in a row (the horizontal stream length; width_1), as well as the row length (width_2) in order to calculate the address of the first stream element in the next row. In the example, $\text{stride}_1 = 3$, $\text{stride}_2 = 2$, $\text{width}_1 = 4$, $\text{width}_2 = 16$, and the stream length is 12.

So, the s-SAD instruction requires 8 operands (the 5 parameters mentioned above, the starting addresses of both streams and the destination register). Without special facilities, this will not fit into a 32-bit instruction format. We, therefore, first provide the general framework of the proposed architecture. It has the following facilities:

- A 64-bit stream status register (SSR).
- 32 32-bit stream control registers (SCR).
- 32 32-bit stream general registers (SGR).
- 32 stream registers (SR), whose lengths are implementation dependent.
- Stream bit vector registers for selective executions.

The instruction format of the s-SAD instruction is

s-SAD SGRi,SGRj,SGRk,CRUk,CRLl

where the SGR's are stream general registers, and CRUk and CRLl are stream control registers. The instruction can

be fitted into 32 bits by employing the following techniques. The opcode is 9 bits and the SGR-fields are 5 bits. The control registers are divided into 16 upper (denoted by CRU) and 16 lower register (denoted by CRL). So, they can be specified by 4 bits. The CRU registers are addressed as pairs of even-odd registers. The even-numbered registers are stride registers; the 16 most significant bits specify the horizontal stride whereas the 16 least significant bits specify the vertical stride. The corresponding odd-numbered register contains the horizontal stream length $width_1$. In order to increase the number of registers available, specifying an odd-numbered register, for example CRU3, implies that both operands have a stride of one in both dimensions and that CRU3 contains the value of $width_1$. The lower 16 control registers denoted by CRL are also addressed as pairs of even-odd registers. The even register contains the stream length and the odd register contains the row length ($width_2$).

Given this framework, the following code will compute the SAD of two 16×16 blocks:

1. LD CRU1, 16 (Load the horizontal stream length to an odd-numbered upper control register).
2. LD CRL2, 256 (Load the stream length to an even-numbered lower control register).
3. LD CRL3, frame_width (Load the width of the frame in the next odd-numbered lower control register).
4. LD SGR1, A (Load starting address of stream A)
5. LD SGR2, B (Load starting address of stream B)
6. s-SAD SGR3, SGR1, SGR2, CRU1, CRL2 (Perform the SAD operation on the entire data streams, i.e., (1) reduce CRL2 with the machine's section size, (2) set the condition code (in the stream status register) if the content of CRL2 is ≤ 0; this condition code indicates if there are more sections to process, (3) advance properly the addresses contained in SGR1 and SGR2, and (4) operate on the stream section and accumulate in SGR3. Repeat this until the condition code indicates that the entire stream has been processed.)

Thus, the entire loop is replaced by 1 instruction and the entire computation is reduced to 6 instructions. Furthermore, the code is independent of the machine's section size. There is only an implementation constraint on the section size but not an architectural one. Because of this, the code does not need to be recompiled if the vector length is increased.

The requirements for the new instructions to incorporate the memory interfacing are the following:

- Two address generation units are needed instead of one.
- Two ported caches may be required in order to achieve the required memory bandwidth. One ported cache or a sibling address generator will also be sufficient.

It should be noted that the s-SAD is an example and that other frequently occurring operations can be implemented as a streamed instruction. Example are DCT, IDCT, Huffman coding and various filters. In the next section we provide some examples and evaluate the performance improvement resulting from employing streamed instructions.

3. Evaluation

In order to quantify the performance potential of streamed instructions, we incorporated them into the **sim-outorder** simulator from the SimpleScalar toolset (release 2.0) [3]. This simulator emulates a superscalar processor with out-of-order issue and execution. The instruction set architecture is derived from the MIPS-IV ISA.

In order to isolate the effects of adding streamed instructions, we simulated an idealized memory system with a latency of one cycle (i.e., the equivalent of a perfect cache). However, we measured the impact of cache misses on the performance of streamed instructions by varying their latency. The performance estimations are, therefore, conservative, since increasing the memory latency will not affect the performance of the streamed instructions, but will negatively affect the performance of scalar instructions. The simulator parameters are listed in Table 1.

3.1. Streamed SAD

One of the advantages of streamed instructions is the ability to overlap the read latency with execution. With SIMD extensions like VIS [15] and MMX [12] an entire block needs to be fetched before the operation can be performed, but with streamed instructions, an operation can be performed while the next block is being fetched, since the instruction contains all necessary information (base address, length, stride) to complete the entire operation.

instruction decode	4 inst/cycle
instruction issue	4 inst/cycle
instruction commit	4 inst/cycle
register update unit (RUU) size	16
load/store queue (LSQ) size	8
number of integer ALU's	4
number of integer multiplier/dividers	1
number of memory ports	2
number of FP ALU's	4
number of FP multiplier/dividers	1

Table 1: Basic simulator parameters.

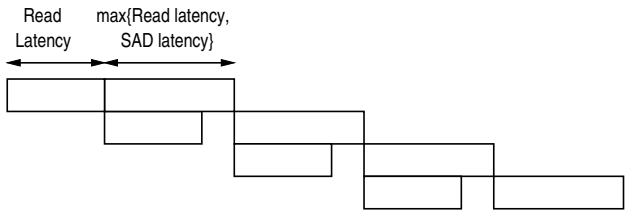


Figure 6: Streamed instructions have the potential of overlapping the read latency with execution times.

As depicted in Figure 6, the latency of the fetch is entirely hidden by the execution time if the fetch latency is smaller than or equal to the execution time. Otherwise, the time to process one block is equal to the read latency. The time to process an, e.g., 16×16 block (assuming the SAD functional unit is not pipelined) is therefore given by

$$\textit{Read latency} + \left\lceil \frac{256}{\textit{section size}} \right\rceil \times \max\{\textit{Read latency}, \textit{SAD latency}\}.$$

If the SAD functional unit is fully pipelined, the time may even be smaller, but we do not consider this most ideal case in this paper.

In the experiments, we assumed that the section size is 16 and that the *SAD latency* is equal to 3 cycles (as shown in Section 2) and varied the *Read latency* between 1, 2, 4 and 8 cycles (to model the effects of cache misses). So, in the first two cases the time to process a 1×16 block is equal to the *SAD latency*; in the last two cases it is equal to the *Read latency*.

Table 2 shows the total number of execution cycles needed to encode the `test` and `mei16v2` bit streams. The number of execution cycles required by the baseline superscalar architecture is also included. The streamed SAD instruction improves performance by a factor of 2.12 to 2.24 (when the read latency is 1 cycle) to 1.85 to 1.94 (read latency of 8 cycles). Furthermore, increasing the read latency from 1 to 2 cycles has a negligible effect, since the it remains completely hidden by the execution time. Even going 4 cycles has only a minor impact, even though this means that the read latency is no longer be completely hidden. Increasing the read latency to 8 cycles, however, incurs a performance penalty of 15-16% compared to a read latency of 1 cycle.

3.2. Streamed IDCT and FDCT

Besides motion estimation, another important stage in video compression is the discrete cosine transform (DCT). Video encoding performs both the forward DCT (FDCT) and the inverse DCT (IDCT). To speed up the IDCT, the encoder uses fixed-point arithmetic, but for the FDCT, it uses

	`test`	`mei16v2`
Read Latency (cycles)	Execution time (10^6 cycles)	Execution time (10^6 cycles)
baseline	92.6	795.4
1	43.6	354.7
2	43.7	355.7
4	45.1	367.4
8	50.1	410.2

Table 2: Simulated execution time in cycles versus read latency for two inputs. The latency of the SAD operation is assumed to be 3 cycles.

	`test`	`mei16v2`
s-IDCT Latency (cycles)	Execution time (10^6 cycles)	Execution time (10^6 cycles)
+ s-SAD (50)	43.7	355.7
10	42.7	347.8
20	42.7	347.9
40	42.7	348.1

Table 3: Simulated execution time versus the latency of the s-IDCT instruction. The 16×16 SAD is assumed to take 50 cycles.

FP arithmetic. In the experiments we varied the latency of the s-IDCT instruction from 10 cycles (1 cycle read latency to fetch the first row, 8 cycles to process each row/fetch the next row, and 1 cycle write latency), to 20 cycles (2 cycles per row plus the read and write latency), to 40 cycles. Because the FDCT uses FP arithmetic, we varied its latency from 20 to 60 cycles. In all cases, the latency of the SAD operation is assumed to be 3 cycles and the initial read latency 2 cycles, so that it takes 50 cycles to process two 16×16 blocks.

First, we consider the case in which only the streamed IDCT is added to the instruction set in addition to the s-SAD. Table 3 depicts the total number of execution cycles as a function of the latency of the s-IDCT instruction.

Adding the s-IDCT instruction does not yield a large performance improvement. Even under the optimistic assumption that the s-IDCT takes only 10 cycles, the performance improvement is only 2.2 to 2.3%. The reason for this is twofold. First, the IDCT stage is already very efficient, since it uses fixed-point arithmetic. Second, the IDCT constitutes only a small fraction of the total execution time.

Because the FDCT uses floating-point arithmetic, it is computationally more expensive than the IDCT. Table 4 shows the total number of execution cycles as a function of the latency of the s-FDCT instruction.

The inclusion of the s-FDCT instruction yields a significant performance improvement. For example, even if we assume that it takes 60 cycles to process one block, the exe-

	test	mei16v2
s-FDCT Latency (cycles)	Execution time (10^6 cycles)	Execution time (10^6 cycles)
+ s-IDCT (20)	42.7	347.9
20	30.7	262.6
40	30.7	262.7
60	30.7	262.9

Table 4: Simulated execution time versus the latency of the s-FDCT instruction. The SAD operation is assumed to take 50 cycles, the latency of the s-IDCT instruction is set to 20 cycles.

```
static void add_pred(pred,cur,lx,blk)
unsigned char *pred, *cur;
int lx;
short *blk;
{
  int i, j;

  for (j=0; j<8; j++){
    for (i=0; i<8; i++)
      cur[i] = clp[blk[i] + pred[i]];
    blk+= 8;
    cur+= lx;
    pred+= lx;
  }
}
```

Figure 7: C code for saturating add.

```
static void sub_pred(pred,cur,lx,blk)
unsigned char *pred, *cur;
int lx;
short *blk;
{
  int i, j;

  for (j=0; j<8; j++){
    for (i=0; i<8; i++)
      blk[i] = cur[i] - pred[i];
    blk+= 8;
    cur+= lx;
    pred+= lx;
  }
}
```

Figure 8: C code for calculating the differences.

	test	mei16v2
Streamed ALU Latency (cycles)	Execution time (10^6 cycles)	Execution time (10^6 cycles)
+ s-FDCT (40)	30.7	262.7
10	29.9	256.9
20	29.9	257.0
40	29.9	257.1

Table 5: Simulated execution time in cycles versus the latency of the s-SADD and s-SUB instructions. The 16×16 SAD operation is assumed to take 50 cycles, the s-IDCT 20 cycles, and the s-FDCT 40 cycles.

cution time is 28% smaller than that needed by the program that does not use the s-FDCT instruction. The reason is, of course, that the FDCT is computationally expensive.

3.3. Streamed Saturating ADD and Streamed SUB

Figure 7 depicts the C-code from the MPEG encoder that reconstructs the current block from the reference block and the calculated differences. This is done because the motion estimation stage uses the *decoded* frame as the reference frame to avoid unnecessary errors. The values are saturated (clipped) to 0 and 255, so that each pixel can be represented by one byte. This routine is another example of a streamed operation, which we refer to as the streamed saturating add (s-SADD). Figure 8 shows the code that calculates the differences between the current block and the reference block. For this piece of code we introduce the streamed SUB (s-SUB) instruction. Because of the similarity between these two instructions, we assume that they are performed by the same unit, a *streamed ALU*. We also assume that both instructions require the same amount of time.

Table 5 shows the total number of execution cycles required by the program after the streamed ALU has been added. Adding the s-SADD and s-SUB instructions yields a modest performance improvement. For example, if the latency of these instructions is 20 cycles, the program that employs them is about 2.2-2.6% faster than the program that does not use them. The reason is that not much execution time is spent in the `add_pred` and `sub_pred` functions.

3.4. Summary

The most important performance metric is the total number of execution cycles. Table 6 summarizes the performance achieved by each program variant. The input bit stream is `mei16v2`. In the first column of this table, the assumed latency is given between parentheses.

As can be seen, the streamed instructions significantly reduce the total execution time. Overall, a performance improvement by a factor of 3.1 is achieved. The biggest improvement is obtained when the s-SAD instruction is added, since motion estimation is the most time consuming part of the encoder.

In addition to improving the execution time, we also claimed that streamed instructions significantly reduce the number of executed instructions. To validate this claim,

Program variant	Execution time (10^6 cycles)	# of times streamed instr. is executed	# of executed instr. ($\times 10^6$)	CPI
original	795.4	N/A	1133.8	0.7015
+ s-SAD(50)	355.7	994280	541.4	0.6570
+ s-IDCT(20)	347.9	8448	523.5	0.6646
+ s-FDCT(40)	262.7	8448	388.2	0.6768
+ s-SADD,s-SUB(20)	257.0	8448	374.4	0.6864

Table 6: Summary for the `mei16v2` bits stream.

the second column in Table 6 contains the number of times each streamed instruction is executed, and the third column contains the total number of instructions executed. The s-SAD instruction is executed most often, and it reduces the number of executed instructions by a factor of 2.1. The s-IDCT, s-FDCT, s-SADD, and s-SUB instructions each are executed once for every 8×8 block (the input bit stream consists of 8448 blocks) and, together, they reduce the number of executed instructions by a factor of 3.0.

Finally, the fifth column contains the CPI measured for each program. Since streamed instructions may require many cycles, it is expected that the CPI increases when they are included. This is, however, not true. As shown in Table 6, the CPI remains about the same.

4. Related Work

Complex streamed instructions combine complex memory accesses (with implicit prefetching), program control for vector sectioning and complex computations on multiple data in a single operation. The architectural[2] family closest to our proposal is the vector architecture family (including sub-word SIMD multimedia architectures). There are several notable differences between vector processors and our proposal. We begin by considering mainframe vector architectures that include complex vector operations, such as multiply-add/subtract, multiply-accumulate, etc., see, e.g., [11, 2]. The amount of data operated upon in such architectures is limited by a fixed section size, usually less than 512 elements for mainframe computers[3]. Furthermore, the actual implementation fixes the section size, i.e., the code is dependent on the section size. This is the first notable difference to our proposal which assumes only an *implementation* constraint on the section size but not an *architectural* one. All proposals for multimedia architectures such as the Intel MMX and extensions [12, 7, 1, 14], the Sun VIS [15] and the HP MAX [8] also have architectural constraints on the section size with the additional constraint that it is usually at most 128 bits rather that 512 words.

In all cases, sectioning is common to all schemes and is implemented using loops with mixed scalar and vector instructions. There are two aspects to the controlling of the loop. One concerns the setting of the "variables" that control the execution, the other the "repetition" of the loop. This is achieved by setting certain registers (e.g., for vector length, starting address, etc.), usually using integer instructions or special instructions that set vector counts and condition codes[4], and the use of branches for repetition. To our knowledge, there is no other proposal in mainframe vector or sub-word vector processing that proposes single instructions that will perform all these operations at once. An additional difference between previous proposals and the one described here concerns the loading of operands. When executing memory-to-memory or memory-to-register operations, mainframe vector processors perform no prefetching or speculations. New multimedia (internet) instruction sets such as [7] may allow preprocessing of memory accessing via speculation and prefetching. This is also true for the complex streamed instruction set we propose, because memory accessing is operated upon with implicit speculation. Furthermore, our scheme also performs sub-matrix addressing. This is not only absent from other multimedia and mainframe vector processing instructions, it is also absent from the pure load/store instructions of these instruction sets. A final remark relates to multimedia instructions and in particular to complex instructions such as the SAD. There are new additions, such as the PERR instruction in the Alpha MVI [13] and the PSADBW instruction in the new Pentium III instruction set [6]. We differentiate from such proposals as we presented a fast implementation for this operation[5] and more new complex operations.

[2] Architecture denotes the attribute of a system as seen by the programmer, i.e., the conceptual structure and functional behavior of the processor. It is distinct from the organization of the data flow and physical implementation of the processor [11].

[3] For example, in the vector architecture described in [2], the section size is a power of 2 between 8 and 512.

[4] For an example, see the load vector count and update (VLVCU) in the IBM 370 vector architecture [2].

[5] Recently, an implementation for the Intel MMX PSADBW instruction has been introduced in [6]. Following the discussion of [6], we conclude that the described approach is not the best possible. This is because to determine the sum of absolute differences it is stated that their implementation performs the following three micro-operations: (1) determine for each pair of inputs the difference, (2) take the absolute value of this difference, (3) take the sum of these absolute differences. This is clearly more expensive than our approach.

Another related proposal is the *Matrix Oriented Multimedia* (MOM) extension proposed in [4]. MOM instructions can be viewed as vector versions of MMX-like instructions. Thus, they operate on an entire matrix at once. There are three main differences between MOM and our proposal. First, in MOM the entire matrix must be loaded into a MOM register (consisting of 16 64-bit elements) before an arithmetic operation can be performed. Thus, the read latency is not overlapped with execution (from the exposition in [4] it is not clear if chaining is performed). Second, although MOM does not require that the rows of the matrix are layed out in memory in consecutive locations, consecutive stream elements in the same row have to be stored consecutively. Third, in MOM the "horizontal stream length" is limited to 8 (in case the SIMD vector elements are bytes) and the "vertical stream length" is limited to 16. Computing the SAD of 16×16 blocks, for example, requires two MOM instructions. These limitations are not present in our proposal.

5. Concluding Remarks

In this paper we have introduced an architectural paradigm intended to improve the performance of streaming operations. The proposed complex streamed instructions perform setting, controlling and performing vector operations simultaneously. In addition, each instruction performs complex memory access and execution operations.

We have performed simulations using the SimpleScalar toolset. The experiments have shown that the performance improves by a factor of 3.1, that the number of executed instructions reduces by a factor of 3.0 to 3.2, and that the CPI remains the same. These results strongly suggest that complex streamed instructions are promising in delivering improved performance for applications requiring streaming operations.

In the future we intend to look at other benchmarks and applications and try to identify additional streamed instructions. Furthermore, we are currently performing simulations comparing our approach to sub-word parallel multimedia extensions like MMX and VIS.

References

[1] J. Able, K. Balasubramanian, M. Bargeron, T. Craver, and M. Phlipot. Applications Tuning for Streaming SIMD Extensions. *Intel Technology Journal*, May 1999.

[2] W. Buchholz. The IBM System/370 Vector Architecture. *IBM systems journal*, 25(1):51–62, 1986.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Univ. of Wisconsin-Madison, Comp. Sci. Dept., 1997.

[4] J. Corbal, M. Valero, and R. Espasa. Exploiting a New Level of DLP in Multimedia Applications. In *MICRO 32*, 1999.

[5] L. Dadda. Some Schemes for Parallel Multipliers. *Alta Frequenza*, 34:349–356, May 1965.

[6] K. Diefendorff. Pentium III = Pentium II + SSE, Internet SSE Architecture Boosts Multimedia Performance. *Microprocessor Report*, 13(3):5–11, March 1999.

[7] J. Keshava and V. Pentkovski. Pentium III Processor Implementation Tradeoffs. *Intel Technology Journal*, May 1999.

[8] R. Lee. Effectiveness of the MAX-2 Multimedia Extensions for PA-RISC 2.0 Processors. slides from HotChips IX, August 1997.

[9] R. B. Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[10] MPEG Software Simulation Group. MPEG-2 Video Codec. Available via `http://www.mpeg.org/MSSG/`.

[11] A. Padegs, B. Moore, R. Smith, and W. Buchholz. The IBM System/370 Vector Architecture: Design Considerations. *IEEE Trans. on Comp.*, 37(5):509–520, May 1988.

[12] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):24–38, January 1997.

[13] P. Rubinfeld, B. Rose, and M. McCallig. Motion Video Instruction Extensions for Alpha. http://www.digital.com/semiconductor/papers/pmvi-abstract.htm, October 1996.

[14] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, May 1999.

[15] M. Tremblay, J. M. O'Conner, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, August 1996.

[16] S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechanek. The Sum-Absolute-Difference Motion Estimation Accelerator. In *EUROMICRO 24*, volume II, pages 559–566. IEEE, August 1998.

[17] S. Vassiliadis, J. Hoekstra, and H.-T. Chiu. Array Multiplication Scheme Using (p,2) Counters and Pre-Addition. *Electronics Letters*, 31(8):619–620, April 1995.

[18] C. Wallace. A Suggestion for Parallel Multipliers. *IEEE Trans. Electron. Comput.*, EC-13:14–17, 1964.