Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Journal of Systems Architecture 49 (2003) 599–617
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Implementation of a streaming execution unit

Dmitry Cheresiz [a,b,*], Ben Juurlink [a], Stamatis Vassiliadis [a], Harry A.G. Wijshoff [b]

[a] *Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands*
[b] *Leiden Institute of Advanced Computer Science, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands*

Abstract

The Complex Streamed Instruction (CSI) set is an instruction set extension targeted at multimedia applications. CSI instructions process two-dimensional data streams stored in memory and the streams can be of any length. Sectioning (the process of splitting up arbitrary-length streams into fixed-size sections that fit in a vector register), data alignment, and conversion between different packed data types are all performed in hardware. It has been shown previously that CSI provides significant speedups compared to current media ISA extensions such as MMX and VIS. This paper presents a detailed design of a unit that can execute CSI instructions under the assumption that it is interfaced with the first-level data cache. In particular, it is shown that the complex, two-dimensional, address-generation calculations can be performed in a pipelined fashion and implemented using a three-stage pipeline with acceptable delay and hardware cost.

Keywords: Multimedia applications; Instruction set extension; Pipelining

1. Introduction

The growing importance of multimedia applications for the desktop market motivated major processor vendors to extend their instruction set architectures (ISAs) with instructions that can be used to implement key multimedia algorithms efficiently. Examples of such extensions are the *Visual Instruction Set* (*VIS*) for the *UltraSPARC* architecture and the *MultiMedia eXtension* (*MMX*) for the *x86* architecture [1,2]. These extensions are, essentially, load-store vector architectures with short (typically, 64-bit or 128-bit) vector registers called multimedia registers. In a 64-bit vector register, for example, a vector consisting of eight 8-bit, four 16-bit, or two 32-bit elements can be stored. The instructions provided in VIS and MMX take advantage of the fact that multimedia applications process small data types (for example, 8-bit pixels or 16-bit audio samples) and exploit the data-level parallelism present in these codes by operating on all vector elements in parallel.

The ISA extensions mentioned above have proven to provide significant performance benefits

* Corresponding author. Address: Computer Engineering Laboratory, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands. Tel.: +31-15-2787362; fax: +31-15-2784898.
E-mail address: cheresiz@dutepp0.et.tudelft.nl (D. Cheresiz).

1383-7621/$ - see front matter
doi:10.1016/j.sysarc.2003.06.003

(see, e.g., [3,4]). Further performance improvements may be limited, however, due to several characteristics of these ISA extensions.

First, the fixed size of the registers limits the number of operations performed in parallel by a single instruction. For example, since MMX registers are 64 bits wide and the smallest data type supported by MMX is a byte, the number of parallel operations performed by a single MMX instruction is at most 8. Much higher amounts of parallelism are present in many multimedia kernels where the same operation often has to be performed on data streams consisting of tens to hundreds of elements. To implement such kernels using MMX instructions, the streams have to be split into sections that fit into the 64-bit registers. This process, called *sectioning*, results in a large number of instructions to be executed.

Second, VIS and MMX implementations of multimedia kernels may require a significant number of overhead instructions for data alignment and conversion, increasing the instruction count even further. Data alignment instructions are needed if the data is not aligned at a byte-address that is a multiple of the vector register size in bytes. Data conversion instructions are needed to convert data from *storage format* (the number of bytes a data element occupies in memory) to *computational format* (the number of bytes used during computation) and vice versa. Often, the storage format is too narrow for intermediate computations to occur without overflow. According to [3], up to 41% of the total instruction count for VIS constitutes overhead.

With the number of instructions being fixed, more instructions have to be fetched, decoded, and executed in each cycle in order to increase the performance of MMX- or VIS-enhanced processors. Increasing the issue width, however, requires a substantial amount of hardware [5] and increases the cycle time [6]. Another possibility to improve performance is by increasing the multimedia register size. The drawback of this approach is, however, that it implies a change of the ISA and, therefore, requires recompiling or even rewriting existing codes. Furthermore, to increase the register size beyond 256 bits is not likely to provide much benefit, because many multimedia kernels process small two-dimensional sub-matrices and only a limited number of elements, typically 8 or 16, are stored consecutively.

The *Complex Streamed Instruction* (*CSI*) set has been proposed in order to overcome these limitations. Two-dimensional streams of arbitrary length can be processed by a single CSI instruction which performs the actual computation, the memory accesses, sectioning, as well as data alignment and conversion. CSI has proven to provide significant speedups on a wide variety of multimedia applications [7–10]. For example, using a near cycle-accurate simulator we have shown that a 4-way superscalar processor enhanced with a CSI unit capable of operating on 32 bytes in parallel outperforms the same processor enhanced with comparable amount of VIS execution hardware by factors of up to 7.4 on kernels and by factors of up to 1.54 on full applications.

In this paper a detailed design of a unit that can execute CSI instructions is presented. It is organized as follows. Section 2 provides a brief description of the CSI architecture. Section 3 describes the datapath of the streaming execution unit and discusses to which level of the memory hierarchy it should be connected. Section 4 is focused on the control organization of the unit. After that, Section 5 presents a detailed description of the address-generation hardware. It is shown that the complex, two-dimensional, address-generation calculations can be implemented using a three-stage pipeline with acceptable delay. In Section 6 we discuss how the results of this paper influence the performance results reported in our previous work, propose directions for future research, and present concluding remarks.

2. The CSI architecture

In this section we briefly sketch the CSI multimedia ISA extension. Further details on the CSI architecture can be found in [7].

CSI is a memory-to-memory architecture for two-dimensional streams. As illustrated in Fig. 1, the streams follow a matrix access pattern, with a fixed *horizontal stride* (distance between consecutive elements of the same row) and a fixed *vertical*

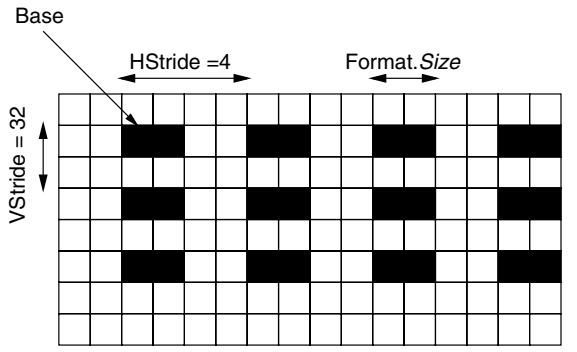

Fig. 1. Format of a stream. Each box represents a byte. Filled boxes are stream elements.

stride (distance between rows). The stream length is not fixed architecturally. In other words, CSI instructions can process streams of arbitrary length, since the stream length is an operand of the instruction.

Each stream is specified by a *set of stream control registers* (SCR-set) that consists of the following registers:

(1) The Base register holds the address of the first element of the stream.
(2) HStride contains the stride (in bytes) between consecutive stream elements in a row. It is assumed to be positive.
(3) HLength contains the number of stream elements in a row.
(4) VStride contains the distance in bytes between consecutive rows.
(5) VLength contains the number of rows in the stream.
(6) The Format register consists of three fields, *Size*, *ProcessingSize*, and *Scale-Factor*, that describe the storage and the computational format of the stream elements. It also contains some other fields, such as *Saturate*, which determines whether saturation or wrap-around arithmetic should be performed. Further details are given in [7].

Additionally, each SCR-set contains two more registers, CurrCol and CurrRow, which specify the position of the element currently being processed. They are used for interrupt-handling.

Most CSI instructions fetch two input streams from memory, perform arithmetic operations on corresponding elements, and store the resulting output stream back to memory. For example, the instruction `csi_add SCRSk, SCRSi, SCRSj` adds corresponding elements of the data streams described by the SCR-sets `SCRSi` and `SCRSj`, and writes the results to the stream specified by the SCR-set `SCRSk`. If the storage format of one of the source streams differs from the computational format, the elements are converted (*unpacked*) before being processed. Similarly, if the computational format differs from the storage format of the destination stream, the results are *packed* before being stored. Suppose, for example, that `SCRS1` and `SRCS2` describe streams with 8-bit and 16-bit elements, respectively, and that `SRCS3` specifies a stream with 8-bit elements. Then the instruction `csi_add SCRS3, SCRS1, SCRS2` will load both input streams, unpack the elements of the stream described by `SCRS1` from 8 to 16 bits, perform additions in 16-bit precision, pack the results back from 16 to 8 bits and write them to the destination stream.

The key advantages of the CSI architecture are its ability to exploit large amounts of parallelism using a single instruction, the reduction of the overhead associated with converting between different data types, and the possibility to run the same CSI code on an implementation with a wider SIMD datapath without recompilation. The first feature is achieved by having no restriction on the stream length and by supporting two-dimensional streams. The second one results from performing conversion internally in hardware and overlapping it with useful computations. The third characteristic, code compatibility, is a consequence of the fact that the number of elements that are actually processed in parallel is not part of the architecture.

3. Datapath of the CSI execution unit

A CSI instruction such as `csi_add` loads the source streams from memory, unpacks (if necessary) the stream elements from storage to computational format, performs a certain operation on corresponding elements, packs (again if necessary)

the results, and stores the resulting output stream back to memory. Since these operations are independent, they can be pipelined. The CSI execution unit is, therefore, organized as a pipeline in which stream data flows through a sequence of stages that perform these operations. The datapath of the streaming execution unit is depicted in Fig. 2. For clarity, some parts (for example, floating-point hardware) have been omitted. The control logic is described in Section 4.

The main hardware entities of the streaming execution unit are the *stream control register sets* (SCR-sets), *memory-interface unit* (MIU), the *stream input* and *stream output* buffers, the *pack* and *unpack* units, and one or more SIMD-like functional units. In Fig. 2, two SIMD functional units, medADD and medMUL, are shown that perform addition-related and multiply-related operations, respectively. In the remainder of this section we describe the parts of the streaming execution unit in detail.

The memory interface unit is responsible for transferring data between the memory hierarchy and the stream input buffers. In addition, if the source stream elements are not stored consecutively, it must also extract and store them consecutively in the stream buffers. If the destination stream elements are not stored consecutively, the unit must perform the reverse operation, scattering data into appropriate memory locations.

Each unpack unit converts stream data from storage format to computational format (if required). Unsigned fixed-point numbers are zero-extended (additional bits are filled with zeroes) while signed numbers are sign-extended. Additionally, the values can be scaled by means of shifting in order to position the binary point. These operations are controlled by the fields of the Format register of the corresponding SCR-set.

The functional units medADD and medMUL perform SIMD parallel operations on the data contained in the input latches. If these input lat-

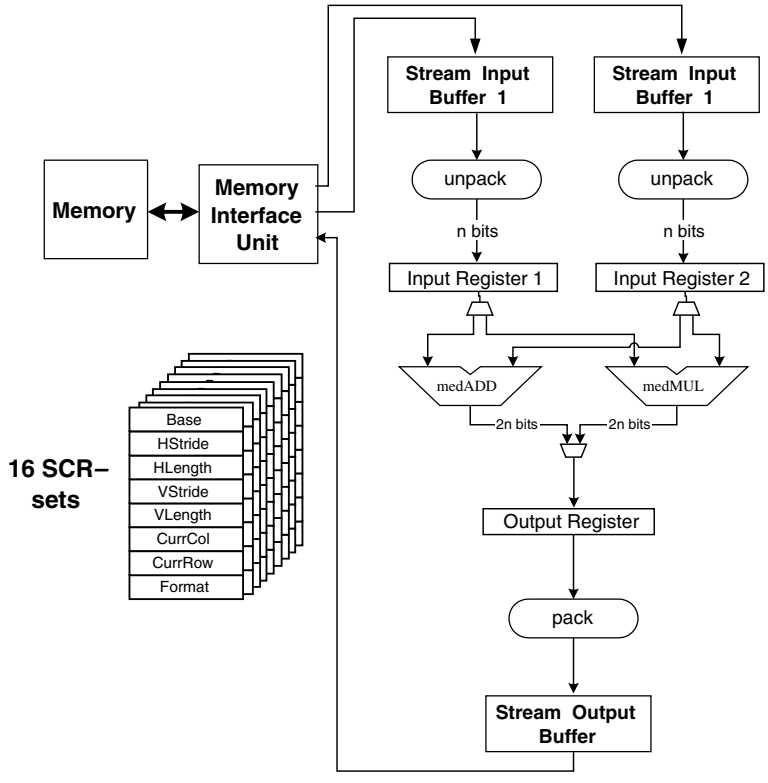

Fig. 2. Datapath of the streaming execution unit.

ches are n bits wide, these units process either $n/8$ bytes in parallel, $n/16$ halfwords, or $n/32$ words. The value of n is implementation dependent. It can be 64, 128, or even larger. As mentioned before, this feature allows CSI codes to take advantage of a wider datapath without recompilation. The output register is $2n$ wide so that no overflow occurs during computation. The medADD unit performs the usual addition, subtraction, and bitwise logical operations, as well as addition-related operations such as the *Paeth* operation [11]. It also expands the output to $2n$ bits by padding it with zeroes in order to produce the same number of bits as the medMUL unit. The medMUL unit performs the packed multiply operation as well as more complex media operations such as the Sum of Absolute Difference (SAD) [11].

From the output register, data flows to the stream output buffer via the pack unit. The pack unit converts, if necessary, the data from computational format to storage format under control of the Format register of the destination stream. When no conversion is needed, data is passed through the unit without being changed.

The design of the CSI execution unit is strongly influenced by the memory hierarchy level it is connected to. It can be connected to the first-level (L1) cache, or it can bypass the L1 cache and go directly to the L2 cache or even main memory. We decided to interface it to a 2-ported L1 cache. The motivation for this design decision is as follows. First, Ranganathan et al. [3] and Slingerland and Smith [12] have observed that with realistic L1 cache sizes, most multimedia applications for audio, video, speech and document processing achieve high hit rates. Our experiments [7,9,10] support this observation: with a 32 KB direct-mapped L1 data cache, JPEG and MPEG-2 coders/decoders as well as the 3D graphics benchmark *viewperf* exhibited hit rates of over 99%. Another motivation is that since the L1 cache is on-chip, it will not be expensive to widen the path between the cache and the streaming execution unit, so that an entire block can be transfered in a single access. Such a design can provide high data bandwidth without increasing the number of cache ports, which is undesirable since multi-ported caches are expensive and may increase the cache hit time. An additional advantage of attaching the CSI execution unit to the L1 cache is that it keeps the cache coherent with memory. Therefore, in the remainder of this paper we assume that the CSI execution unit is interfaced with the L1 cache.

4. Control organization

This section presents a detailed description of the CSI execution unit, concentrating on the design of the parts which are particular to its L1 cache interface and on the organization of the control. The design of the most complex part of the unit, the address-generators, is presented in Section 5.

Fig. 3 depicts the CSI datapath and the control lines. Thick lines represent paths through which data and addresses move and thin lines are control lines. Rounded rectangular and trapezium shapes are used for computational units and boxes for storage between pipeline stages. In this figure, the stream address-generators (boxes labeled AG), the load and store queues (LQ and SQ), and the extract and insert units represent collectively the implementation of the memory-interface unit (the box labeled MIU in Fig. 2). The latencies of the pipeline stages are estimated in terms of machine cycles, where one cycle is assumed to be approximately twice as long as the delay of a 32-bit adder.

4.1. General control organization

The control logic is distributed across the pipeline stages. Each control signal is associated with a storage entity located between two consecutive pipeline stages. The control organization guarantees that if storage entity S_i is full, no data generated by the preceding pipeline stage will be written to S_i and destroy valid data in it. Therefore, one signal sent by the control logic of S_i to the control logic of the previous storage entity S_{i-1} is, usually, the signal S_i_full. According to this control organization, the CSI pipeline operates in general as follows: at the beginning of each cycle the control logic associated with S_i receives the control signals from the control associated with the next storage element S_{i+1}. Based on these signals

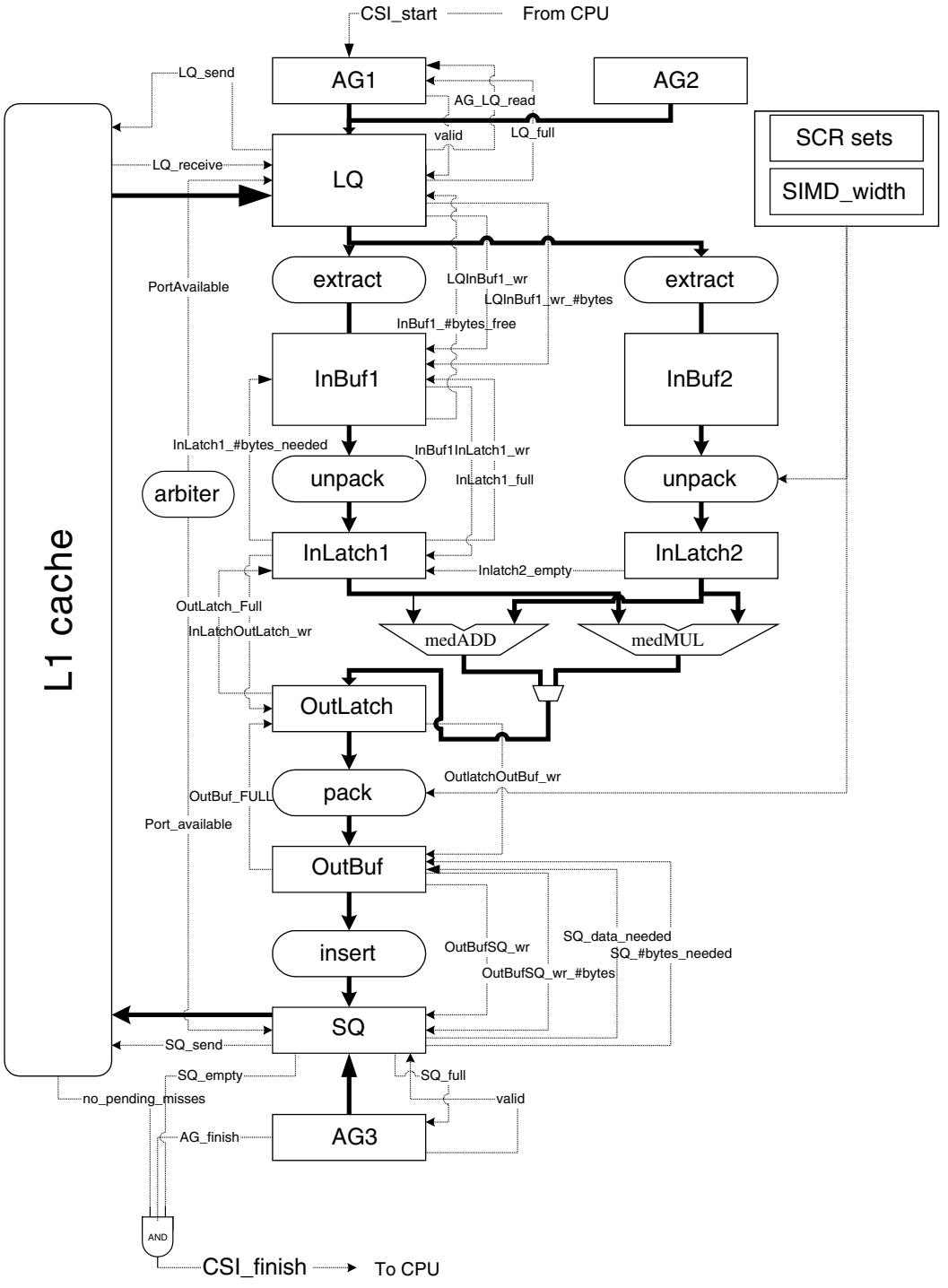

Fig. 3. The CSI datapath with the control lines added.

and its current state, the control logic associated with S_i updates its state and decides whether the data can be transferred to the next pipeline stage. If so, the transfer is performed and the control logic associated with S_{i+1} is notified. After that, it checks the notification control signal generated by the control logic associated with the previous storage entity S_{i-1} if data is transferred from S_{i-1} to S_i. Based on this signal, the control logic associated with S_i again updates its state.

4.2. Stream address generators

Each address generator associated with an input stream (AG1 and AG2) produces a sequence of records consisting of addresses and some bookkeeping information needed to extract stream elements from a cache block. As illustrated in Fig. 4, each AG record consists of the following fields.

- The *addr* field contains the address (aligned at a cache block boundary) of the cache block from which stream data should be extracted.
- Let *bsize* denote the size of an L1 data cache block in bytes. The *mask* field is a (*bsize*·$\log_2$ (*bsize*))-bit *position mask*. It indicates which bytes in the cache block contain stream data. If a byte in the block belongs to the stream, the corresponding $\log_2$ (*bsize*)-bit value is equal to the order of this byte among all stream bytes in the block. For example, for the first byte in the block that belongs to the stream, the corresponding position mask value is 1. For bytes that do not belong to the stream, the value is equal to zero.
- The *els* field contains the number of stream elements contained in the cache block.
- The *bytes* field contains the number of bytes belonging to the stream and contained in the cache block. It is equal to the value of the *els* field multiplied with the value of the *el_size* control register from the corresponding SCR-set.

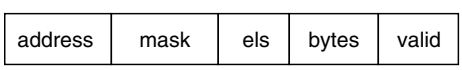

Fig. 4. Format of an AG record.

- Finally, the *valid* field is a 1-bit flag which signifies if the record is valid. Invalid records may be generated by an AG when the end of a row of a two-dimensional stream has been reached.

In Section 5 we describe how these fields are computed. Here we describe the control signals received and generated by each AG and how they influence their operation.

- The *CSI_start* signal is generated by the host CPU. If this signal is set, the fields of the first AG record for the cache block corresponding to the base address of the stream are calculated and the AG starts generating the records.
- The *LQ_full* control signal is generated by the load queue (LQ). When this signal is set, the AG pipeline is stalled. We remark that since each AG is organized as a 3-stage pipeline, up to three valid AG records can be in-flight when the signal is received. To guarantee that none of them is lost, each AG contains a FIFO buffer with three entries.
- The AG_i*_LQ_wr* signal is generated by the control of AG_i ($i = 1, 2$) to notify the LQ that an AG record is transferred. The signal is asserted if *LQ_full* is deasserted and the first record in the FIFO of AG_i is valid. In this case, the record is dequeued from the FIFO and sent to the LQ.

4.3. The load queue

The Load Queue (LQ) fetches data from the L1 data cache and passes it further through the pipeline. Fig. 5 depicts its organization. It is organized as a circular queue where each entry consists of an AG record and three extra fields, *data*, *ready*, and *stream_num*. The *data* field is *bsize* bytes large and contains the cache block fetched from the L1 cache. The *ready* field is a 1-bit flag indicating that the data has arrived. The *stream_num* field indicates to which input stream (1 or 2) the entry belongs. In the experiments reported in [7,13], an 8-entry LQ was used.

The internal state of the LQ consists of registers that hold information needed to implement a circular queue (LQ_head, LQ_tail, LQ_length, and LQ_size), and of the LQ_mem register which points

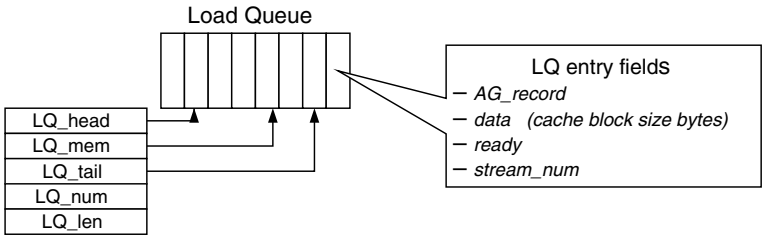

Fig. 5. Organization of the Load Queue.

to the first entry that has not yet been sent to the L1 cache. We now describe the signals received by the control associated with the LQ from other parts of the CSI pipeline, the actions they initiate, and the signals generated by the LQ control and sent to other parts of the pipeline. The four sequences of actions described below are performed in parallel during each cycle.

(1) The control associated with the LQ checks the number of the input stream associated with the first LQ entry. Suppose it is the first stream. Then the LQ control receives the *InBuf1_#bytes_free* signal from the control of InBuf1 and generates the *LQ_InBuf1_wr* signal. This signal is asserted if the cache block for the first LQ entry has arrived and InBuf1 has enough free space to accommodate all bytes in the block that belong to the stream. In this case, the *data* and *mask* fields of the first LQ entry are sent to the extract unit which extracts the stream data from the block according to the mask and places it into the input buffer InBuf1. The $\log_2(bsize)$-wide control signal *LQInBuf1_wr_#bytes* notifies the control of InBuf1 of the number of stream bytes contained in the block. After this, the first LQ entry is discarded.

Since the calculations for generating the *LQInBuf1_wr* signal are simple, they will only take a fraction of a cycle. We therefore deduce that the LQ control is able to perform the same calculations for the second LQ entry in the same cycle and, therefore, pass two cache blocks to the extract stage during a single cycle. This makes the design balanced because, for a two-ported L1 cache, two cache blocks can be delivered to the LQ in one cycle.

(2) If there were less than two free entries in the load queue at the end of the previous cycle, then the *LQ_full* signal is asserted to stall the AGs. Two free entries are needed because two cache blocks can be delivered in a cycle. Otherwise, two entries are appended to the LQ, the *LQ_AG_read* signals are sent to AG1 and AG2, and the first valid AG record of each AG are transferred to the *AG_record* fields of the next two LQ entries.
(3) The *ports_available* signal is received from the cache port arbiter responsible for sharing the cache ports between the load and store queues. If the signal is set and there are two LQ entries which have not yet accessed the L1 cache, the LQ control sends their *address* fields to the cache ports and increments the LQ_mem register.
(4) The *data_available* signal is received from the L1 cache controller. If it is asserted, cache blocks are received from the ports and written to the *data* field of the corresponding LQ entries and the ready flags of these entries are set. Since we assume a 2-ported L1 cache, two cache blocks can be received in a single cycle.

4.4. Extract units and stream input buffers

Each extract unit receives data of a loaded cache block from an LQ entry, extracts the bytes belonging to the stream, and stores them consecutively in one of the input buffers (see Fig. 6). An extract unit is implemented as a $bsize \times bsize$

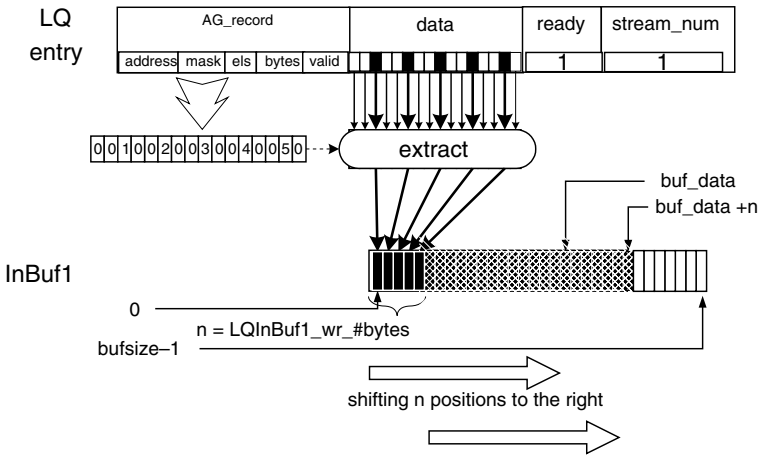

Fig. 6. Extracting the bytes that belong to the stream.

switch controlled by the *mask* field of the LQ entry. It is assumed that the delay of an extract unit is less than one cycle. Each $\log_2(bsize)$-bit entry of the *mask* field controls the position to which the corresponding byte is routed. If the entry is zero, the corresponding byte is discarded.

Each stream input buffer is a shift buffer that stores stream elements consecutively. From there the data is passed to the unpack units. Each input buffer has two control registers. The buf_data register points to the position of the first byte (least recently received from the extract unit) which is not yet passed to the *Unpack* stage. In the buf_size register the size of the buffer (in bytes) is hardwired. We now describe the control organization of the first input buffer and its interface with other parts of the CSI pipeline. The second buffer operates identically. In each cycle the following sequence of actions is performed.

(1) The *InLatch1_full* signal is received from the next storage entity in the CSI pipeline, InLatch1. This signal is asserted if InLatch1 is filled with stream data that has not yet been consumed by the SIMD execution units. If the signal is deasserted, the control logic calculates how many bytes of stream data are needed to fill InLatch1. If InBuf1 contains the required number of bytes, they are sent to the *Unpack* stage and removed from the buffer by adjusting the value contained in the buf_data register. The *InBufInLatch1_wr* signal is then asserted to notify InLatch1.
(2) The *LQInBuf1_wr* signal is received from the LQ. If it is asserted, the contents of InBuf1 are shifted *n* positions to the right and the *n* leftmost bytes are transferred from the extract unit to the *n* leftmost positions in InBuf1, where *n* is the value of the *LQInBuf1_wr_#bytes* control signal generated by the control of the LQ (cf. Fig. 6). After this, the buf_data register is incremented by *n* and the value of *buf_size-buf_data* is sent to the control of the LQ via the *InBuf1_bytes_free* signal.

4.5. Unpack units

Each unpack unit receives stream data from its corresponding input buffer and converts (unpacks) it, if needed, from storage to computational format. Unpacking consists of converting an *m*-bit value to a larger width by sign-extending (for signed fixed-point numbers) or zero-extending (for unsigned numbers) it. Additionally, the promoted value can be shifted, which is performed to reposition the fractional point. The converted data is stored in the input latches located in front of the SIMD functional units.

4.6. SIMD functional units

After the input data has arrived at the input latches in the computational format, the arithmetic or logical operation specified by the CSI instruction is performed by the SIMD functional units. Two input latches, InLatch1 and InLatch2, are located in front of the SIMD execution units, each of which is n bits wide, where $n = 8 \cdot SIMD\_width$ and where *SIMD_width* is the number of bytes processed in parallel by the SIMD functional units. The following sequence of actions is performed for InLatch1 in each cycle. The same actions are performed for InLatch2.

(1) The *OutLatch_full* signal is received from the output latch of the SIMD units, *OutLatch*. If *InLatch1_full* and *InLatch2_full* indicate that both input latches are full and *OutLatch_full* is deasserted, the SIMD computation on the data contained in the input latches is triggered and the *InLatchOutLatch_wr* is asserted to signal to the OutLatch that a transfer has taken place. After that, the *InLatch1_full* and *InLatch2_full* are reset.
(2) The *InBuf1InLatch1_wr* signal is received from the control of InBuf1. If it is asserted, the unpack unit writes data to InLatch1 and the *InLatch1_full* flag is set.

The results of the SIMD execution units flow to OutLatch, the storage entity located after the SIMD functional units. The control associated with OutLatch generates a 1-bit flag *OutLatch_full*. In each cycle the following actions are performed. The *OutBuf_full* signal is received from the next storage entity in the CSI pipeline, the stream output buffer Outbuf. If this signal is deasserted and the Outlatch contains a valid result, then the result is sent to the pack unit and the notification signal *OutLatchOutBuf_wr* is asserted. Simultaneously, the *OutLatch_full* flag is set to the value of the *OutBuf_full* signal and passed to InLatch1. If the flag was set, the input latch will stop triggering new SIMD operations in the next cycle.

We remark that the latency of the medMUL unit is assumed to be at least two cycles and that the unit is fully pipelined. Because of this, there can be two or more valid results in-flight at the moment the *OutLatch_full* signal is asserted. To guarantee that this data is not lost, the output latch is not organized as a single register but as a FIFO buffer with s entries, where s is the number of pipeline stages needed to implement the medMUL unit so that it has a throughput of one result per cycle.

4.7. Pack unit

The pack unit converts (*packs*), if required, the output stream data from computational to storage format. Packing is the reverse operation of unpacking: the elements are shifted (usually, to the right) and then truncated. Additionally, elements can be rounded prior to shifting and saturated when being truncated. Fig. 7 illustrates how the signed 16-bit fixed-point numbers with 4 fractional bits $129.75_{10} = 000100000001.1100_2$ can be packed to 8 bits. First 0.1_2 is added to round the result. After that, the result is rounded to the nearest integer by shifting out the fractional bits. Finally, it is saturated to the largest value representable as an 8-bit 2's complement value to obtain $127_{10} = 01111111_2$.

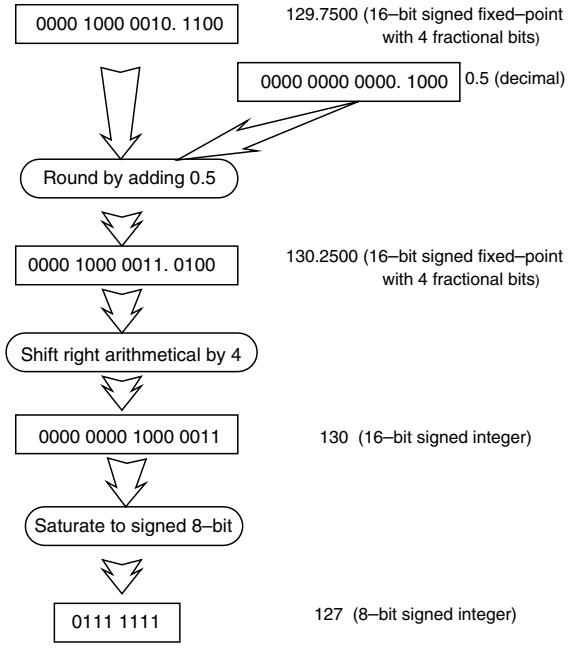

Fig. 7. Packing.

4.8. Stream output buffer and insert unit

The stream output buffer OutBuf operates similarly to the the stream input buffers. It receives the *SQ_data_needed* and *SQ_bytes_needed* signals from the store queue (SQ) in order to determine whether data should be transferred from OutBuf to the SQ through the insert hardware unit. If the transfer should occur, the control of the output buffer asserts the *OutBufSQ_wr* and *OutBufSQ_wr_#bytes* signals and sends them to the SQ. After this, the control logic associated with OutBuf generates the *OutBuf_full* signal and sends it to OutLatch. The signal is asserted if the number of free bytes in OutBuf is less than the number of bytes in the output latch. The insert unit performs the reverse operation of the extract units. It takes from the output buffer the stream bytes that belong to the same cache block and inserts them in the appropriate positions of the *data* field of the SQ entry which requested the insert operation. The positions are determined by the *mask* field of the entry. We note that only the elements which correspond to non-zero mask positions are written.

4.8.1. The store queue

The SQ is organized and operates similarly to the load queue. It is responsible for storing the output stream data in memory. During each cycle the following three sequences of actions are performed in parallel.

- The SQ control receives the *Port_available* signal from the cache arbiter. If it is active and there are entries in the queue which have already received data from the output buffer but is not yet submitted to the L1 cache, they are sent to the cache and the *SQ_send* signals are asserted. It is assumed that two entries can be transferred to L1 during one cycle.
- If there is a free entry in the SQ and a valid record in the FIFO of the output stream AG AG3, the *SQ_AG_read* is asserted and the first valid record from the FIFO is transferred to the *AG_record* field of the SQ entry.
- The *OutBuf_wr* and *OutBuf_wr_#bytes* signals are received from the output buffer. If they are set, *OutBuf_wr_#bytes* are transferred from the buffer to the *data* field of the appropriate SQ entry. After this, the *SQ_data_needed* and *SQ_#_bytes_needed* signals are generated. The first signal is asserted if there are SQ entries for which addresses are generated but data has not yet been received. The second one is set equal to the number of bytes needed by the oldest entry.

5. CSI address generators

The task of the CSI address generators is to produce the addresses of cache blocks containing stream elements and information needed to extract data from a cache block. In this section we describe how these AGs can be implemented. Each AG should produce the following information:

- The addresses $B_{1,1},\ldots,B_{1,k_1},B_{2,1},\ldots,B_{2,k_2},\ldots,B_{m,1},\ldots,B_{m,k_m}$ of all cache blocks that contain stream elements. Each address $B_{i,j}$ is a multiple of the L1 cache block size *bsize*. k_i is the number of cache blocks containing elements of row i of the two-dimensional stream.
- The sequence $mask_{1,1},\ldots,mask_{m,k_m}$ of position masks. Each mask $mask_{i,j}$ consists of $bsize\log_2(bsize)$-bit values. The k-th value is equal to zero if the cache block does not belong to the stream. Otherwise, it indicates the order of the k-th byte among all bytes that belong to the stream.

Consider, for example, the two-dimensional stream illustrated in Fig. 8 and assume that the block size is 8 bytes. The base address of this stream is 804, the horizontal stride is 3, the row length is 4, the element size is 1, and the number of rows is 2. For this stream the AG should generate successively the addresses 800, 808, 904, 912, and the mask vectors (0,0,0,0,1,0,0,2), (0,0,1,0,0,2,0,0), (1, 0, 0, 2, 0, 0, 3, 0), and (0, 1, 0, 0, 0, 0, 0, 0).

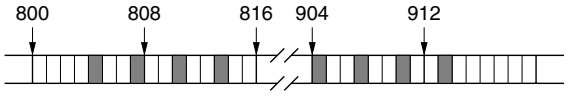

Fig. 8. A stream stored in memory.

This section is structured as follows. Section 5.1 presents formulas for calculating the addresses and the mask vectors. Since these calculations are rather complex and do not appear to fit in one machine cycle, a pipelined implementation capable of generating one AG record each cycle is developed and presented in Section 5.2.

5.1. Formulas for calculating addresses and mask vectors

Let *hstr* denote the horizontal stride of a stream. Throughout this section we assume that $hstr \leqslant bsize$. If this is not the case each cache block contains only one stream element and, therefore, CSI instructions will not achieve a higher throughput than scalar instructions. We also assume that $hstr > 0$.

We introduce the following notations and definitions.

(1) $A_{i,j}$ denotes the address of the first stream element in the cache block starting at address $B_{i,j}$.
(2) $ofs_{i,j}$ is the offset of the first stream element to the block boundary: $ofs_{i,j} = A_{i,j} - B_{i,j}$.
(3) $els_{i,j}$ is the number of stream elements contained in the block starting at $B_{i,j}$.
(4) By $rel_{i,j}$ (*Row Elements Left*) we denote the number of stream elements contained in the blocks starting at $B_{i,j}, \ldots, B_{i,k_i}$: $rel_{i,j} = \sum_{l=j}^{k_i} els_{i,l}$.
(5) Let x and y be integers. We use the symbol % for the modulo operation ($x\%y = x \bmod y$) and the symbols ˜ and & for the bitwise NOT and AND operations.

The calculations performed by each address generator are based on the following formulas.

Proposition 1. *Let SCRS be the stream control register set for which an address generator generates addresses. Then the following equations hold*:

$$A_{1,1} = SCRS.Base,$$
$$A_{i+1,1} = A_{i,1} + SCRS.Vstride \tag{1}$$

$$B_{i,1} = A_{i,1}\&\tilde{}(bsize - 1), \quad B_{i,j+1} = B_{i,j} + bsize \tag{2}$$

$$ofs_{i,1} = B_{i,1}\&(bsize - 1) \tag{3}$$

$$els_{i,j} = \lceil (bsize - ofs_{i,j})/hstr \rceil \tag{4}$$

$$rel_{i,1} = SCRS.HLength, \quad rel_{i,j+1} = rel_{i,j} - els_{i,j} \tag{5}$$

Given the offset $ofs_{i,j}$ of the first stream element in a block to the block boundary, the oset $ofs_{i,j+1}$ in the next block can be calculated using the following theorem.

Theorem 1. *If $ofs_{i,j}$ is the offset of the first stream element in a block to the block boundary, then*

$$ofs_{i,j+1} = (ofs_{i,j} + hstr - bsize\%hstr)\%hstr \tag{6}$$

Proof. Let k be the smallest integer such that

$ofs_{i,j} + k \times hstr \geqslant bsize$.

Then

$$\begin{aligned} ofs_{i,j+1} &= (ofs_{i,j} + k \times hstr)\%bsize \\ &= ofs_{i,j} + k \times hstr - bsize. \end{aligned}$$

Because $ofs_{i,j+1}$ must be smaller than *hstr*, we can take the modulo *hstr* on both sides and obtain

$$\begin{aligned} ofs_{i,j+1} &= ofs_{i,j+1}\%hstr \\ &= (ofs_{i,j} + k \times hstr - bsize)\%hstr \\ &= (ofs_{i,j} - bsize)\%hstr. \end{aligned}$$

In order to implement it in hardware, we want to add a positive term to $ofs_{i,j}$. We, therefore, exploit the fact that $(a + b)\%n = (a\%n + b\%n)\%n$ and add the term *hstr* to the inner expression to obtain

$$ofs_{i,j+1} = (ofs_{i,j} + hstr - bsize\%hstr)\%hstr. \qquad \square$$

Generation of block addresses. Eqs. (1), (2), and (5) are sufficient to construct the sequence of the block addresses $B_{i,j}$. Suppose that $B_{i,j}$ is calculated. If the next block containing stream elements belongs to the same row, then its address is obtained using the second part of (2). Otherwise, it is the first block of the following row. In this case, first, the address $A_{i+1,1}$ is obtained using the second part of (1). The required address $B_{i+1,1}$ is then given by the first equation of (2). The row termination de-

cision and which calculations should be performed can be based on (5). The row termination condition is simply $rel_{i,j+1} \leqslant 0$.

Generation of mask vectors. The mask $mask_{i,j}$ can be derived from the base mask *bmask* which is constant for a given stream and is determined by the stream horizontal stride and the element size. The calculations are based on (6), (4), (5), and the following definitions.

Definition 1. $msb\_mask(bsize, x)$ denotes the $bsize \cdot \log_2(bsize)$-bit binary number for which the x most significant bits (i.e., leftmost bits) are set to 1 and all other bits to 0.

Definition 2. Given a stream with base address 0, horizontal stride *hstr*, row length *HLength*, and element size *elsize*. Let $HLength \geqslant bsize$. By $bmask(bsize, hstr, elsize)$ we denote the mask for the cache block starting at address 0.

Consider, for example, the base mask $bmask(8, 3, 1)$. This mask is a 24-bit vector consisting of eight 3-bit numbers. Element 0, 3, and $6 = 2 \cdot 3$ are equal to 1, 2, and 3, respectively. All other elements are equal to 0 because the corresponding bytes do not belong to the stream. So

$$bmask(8, 3, 1) = 001\ 000\ 000\ 010\ 000\ 000\ 011\ 000.$$

Because the element size *elsize* is always a power of two, it is easy to show that if the mask for a stream with a given stride and element size of one (byte) $bmask(bsize, hstr, 1)$ is known, the mask for a stream with an element size *elsize* of 2 or 4 bytes, $bmask(bsize, hstr, elsize)$, can be obtained using a short sequence of binary shift and AND operations.

The following proposition presents formulas for calculating $mask_{i,j}$.

Proposition 2. *Let* $w = \log_2(bsize)$. $mask_{i,j}$ *can be calculated using the following equations*:

If the cache block is not the last block in a row, then

$$mask_{i,j} = bmask(bsize, hstr, elsize) \gg (ofs_{i,j} \cdot w), \tag{7}$$

where $\gg$ *stands for the shift right logical operation.*

If the cache block is the last block in a row, then

$$mask_{i,j} = (bmask(bsize, hstr, elsize) \gg (ofs_{i,j} \cdot w))\ \& \times msb\_mask(bsize, (y + ofs_{i,j}) \cdot w) \tag{8}$$

where $y = rel_{i,j} \cdot hstr + elsize$.

Proof. Eq. (7) is obvious. Let $C_{i,j}$ be the last block in the ith row, i.e., let $j = k_i$. Then the number $rel_{i,j}$ gives the number of stream elements in $C_{i,j}$. Equation (8) is based on the observation that $rel_{i,j}$ can be less than $els_{i,j}$. We notice that for the mask $mask'_{i,j}$ obtained using only the first part of (8), it might be necessary to zero the last several nonzero mask values, because they correspond to bytes which lie after the row end. This is exactly what is done by the bitwise AND operation on the right-hand side of (8). Indeed, the number of the leftmost bytes in the block which might belong to the stream is equal to $z = rel_{i,j} \cdot hstr + elsize + ofs_{i,j}$. The remaining *bsize*-*z* bytes definitely do not belong to the stream. The mask $msb\_mask(bsize, (y + ofs_{i,j}) \cdot w)$ used in (8) corresponds to this situation, having each of the z leftmost w-bit values set to $11\ldots1$ and each of the remaining (*bsize*-*z*) values set to $00\ldots0$. This concludes the proof. □

5.2. A pipelined implementation

The formulas presented in Proposition 1, Theorem 1, and Proposition 2 are sufficient to generate an AG record for each cache block that contains stream elements. It is important to determine whether these calculations can be implemented in hardware at acceptable cost, what their latency is, and whether they can be pipelined. The possibility of pipelining is critical (if the latency is more than one cycle), because otherwise the rate at which AG records are produced will be less than the rate at which they are consumed by the L1 cache ports (under assumption that the cache hit latency is one cycle), and the performance of the CSI unit might become limited by the address generators. In this section we present a pipelined implementation of the CSI AGs with a latency of three machine

cycles, where the machine cycle time is assumed to be approximately twice as long as the delay of a 32-bit adder.

An address generator is organized as a 3-stage pipeline as depicted in Fig. 9. In this picture, rectangles denote the internal registers and roun-

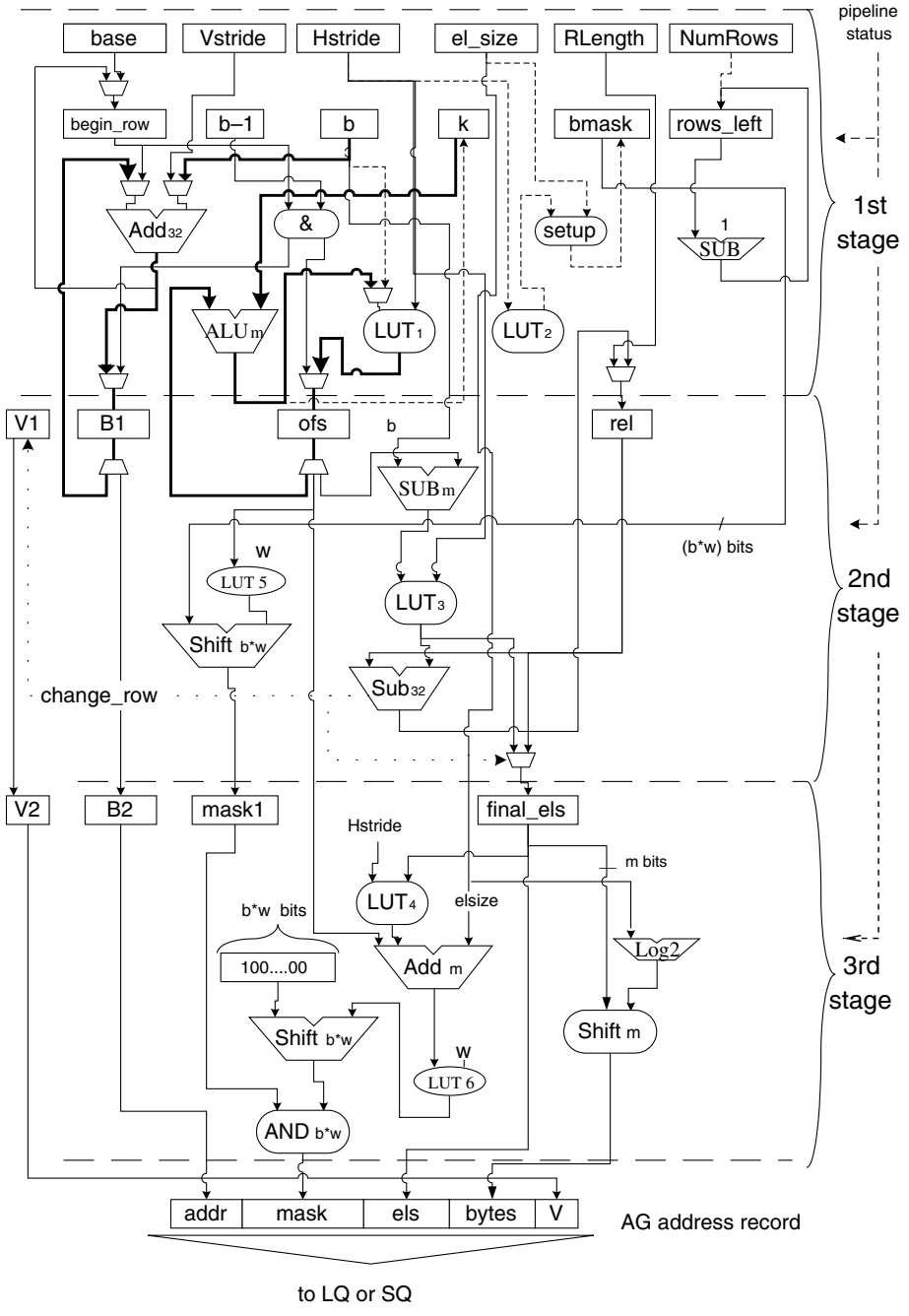

Fig. 9. Organization of a CSI address generator.

ded and trapezium shapes are used for processing hardware. Below we describe the pipeline stages and the operations they perform in detail. To make the figure clearer, the number b is sometimes used instead of *bsize*. The symbol w denotes $\log_2(bsize)$, and m denotes $w+2$.

First stage. Let *str* be the stream assigned to the AG. The stream is described by a certain SCR-set S. During the first stage the block addresses $B_{i,j}$, the offset of the first stream element in the block $ofs_{i,j}$, and the number of the elements remaining in the same row $rel_{i,j}$ are computed for the current cache block $C_{i,j}$. These values are stored in the pipeline latches B1, ofs, and rel, respectively.

The registers in the top row hold copies of the corresponding stream control registers from the SCR-set S. The registers in the second row contain the following values: begin_row holds the address $A_{i,1}$ of the first stream element in the current row $A_{i,1}$, b−1 holds the value of *bsize*-1, and b holds the value of *bsize*. The register k contains the number $hstr - bsize\%hstr$. The bmask register contains the base mask $bmask(bsize, hstr, elsize)$. Finally, rows_left is a counter which contains the number of stream rows (excluding the current one) for which address information has yet to be generated.

Depending on whether $C_{i,j}$ is the first block of the first row, the first block of some other row or not the first block of any row, data can flow along different paths through the first stage. There are three possible situations.

- $C_{i,j}$ is the first block in a row that is not the first one, i.e., $i > 1$ and $j = 1$. The paths used in this case are shown as thin lines in Fig. 9. The following actions take place simultaneously:
  - The values of $A_{i,1}$ contained in the begin_row register and of (*bsize*-1) contained in the b−1 register are fed into the functional unit labeled "&". This unit simultaneously computes the values of $A_{i,1}\&(bsize\text{-}1)$ and of $A_{i,1}\&\tilde{}(bsize\text{-}1)$. These values are written into the B1 and ofs registers, respectively.
  - begin_row is incremented by the contents of VStride using the 32-bit adder Add_{32}.
  - The value contained in HLength is transferred to the rel latch.
  - The rows_left counter is decremented by one.

  Each of these actions can be done in one cycle.
- $C_{i,j}$ is not the first block of the ith row. The paths used in this case are shown as thick lines in Fig. 9. The following actions take place in parallel:
  - Register B1 is incremented by *bsize* using the 32-bit adder Add_{32}. After that, B1 contains the *bsize*-aligned address of $C_{i,j}$.
  - The contents of the ofs latch (which is equal to $ofs_{i,j-1}$ at the beginning of the cycle) is incremented by the value contained in the register k using the m-bit arithmetic unit ALU_m. The result is routed to the LUT_1 unit together with the contents of HStride. This unit computes a mod b, where a is its first input and b its second. The result is written to ofs. These two operations implement Eq. (3). Thus, on the edge of the cycle *ofs* will contain the value of $ofs_{i,j}$.

  Incrementing B1 fits in one cycle. The m-bit addition using the ALU_m unit will take a fraction of a cycle. The subsequent m-bit modulo division will most likely not fit into one cycle if LUT_1 is implemented as an m-bit divider. However, we observe that the width of the first input of LUT_1 is limited by $2 \cdot bsize$:

$$ofs_{i,j-1} + hstr - bsize\%hstr \leqslant bsize + hstr \leqslant 2 \cdot bsize.$$

Furthermore, the second input is constant for a given stream. Therefore, if the remainders (x mod *hstr*) are precalculated for all x ($0 \leqslant x \leqslant 2 \cdot bsize$), the LUT_1 unit can be implemented as a look-up table with $2 \cdot bsize$ entries. For a typical block size of $bsize = 32$, it will have 64 entries. The critical path in this case goes through ALU_m and LUT_1. It is assumed that the critical path fits in one cycle.

According to our experience, most multimedia streams have horizontal strides of 1, 2, 3, or 4. Therefore, if the hardware budget allows, the look-up tables for these strides can be hardwired in the LUT_1 unit. One more table is needed for the case that the stride is different. In this case the table should be loaded with the precalculated remainders before the AG can start generating addresses. This setup operation might take a few cycles. The

parts of the AG datapath used for this setup operation are not shown in Fig. 9. Note that the total number of different tables that should be precalculated is limited by *bsize*, because $hstr \leqslant bsize$. A similar approach can be applied to the LUT_2 look-up table described below.

- Finally, if $C_{i,j}$ is the first block of the first row (i.e., $i = j = 1$), the same actions will take place as for any other first block in a row but, prior to this, some setup actions are performed. The parts of the datapath of the first stage which are utilized during these actions are shown as dashed lines in Fig. 9. These setup operation may require several cycles.
  - If the stride is non-standard, the look-up tables LUT_1 and LUT_2 are loaded.
  - The content of the Base register is transferred to begin_row and the content of the NumRows register to rows_left.
  - The contents of the b and HStride registers are routed to LUT_1, and the resulting value ($bsize\%hstr$) is written to ofs. From there it is passed to the ALU_m unit which computes $hstr - bsize\%hstr$. The result is written to the register k.
  - Simultaneously, the number *hstr* contained in HStride is sent to LUT_2 which returns the value of $bmask(bsize, hstr, 1)$. This value is sent to the setup unit which generates the mask $bmask(bsize, hstr, elsize)$ by means of shift and bitwise AND operations and writes it to the bmask latch.

Which parts of the datapath of the first stage are used and which of the three operation sequences described above are performed is determined by the *pipeline_status* control signal generated by the pipeline stage controller. This signal depends on the *CSI_start* signal provided by the host CPU, the *change_row* signal generated by the second stage of the pipeline, and the stream horizontal stride *hstr*.

Second stage. During this stage the total number of stream elements in the block is computed and some initial mask calculations are performed. This stage has two possible modes of operation: *setup* and *process*, which are activated by the *pipeline_status* signal. When the stage is operating in the *setup* mode, the look-up table LUT_3 is loaded (if the horizontal stride is non-standard). The corresponding parts of the datapath are not shown in Fig. 9.

When the stage operates in the *process* mode, the following actions are performed in parallel.

- The V1 flag which indicates if the record generated for $C_{i,j}$ is valid (i.e., whether $j \leqslant k_i$) is copied to V2.
- The block address is copied from B1 to B2.
- The mask stored in bmask is shifted to the right by $ofs_{i,j} \cdot w$ bits and written to mask1. This action carries out the first computation on the right-hand side of Eq. (8). The multiplication of $ofs_{i,j}$ with w is implemented by the look-up table LUT_5.
- The SUB_m unit calculates $bsize\text{-}ofs_{i,j}$. The result is passed to the look-up table LUT_3, which implements integer division by *hstr*. The output of LUT_3 is equal to $els_{i,j}$ calculated according to Equation (4). It is then subtracted from $rel_{i,j}$ stored in rel. If the result is negative then $C_{i,j}$ is the last block in the row and the *change_row* signal is generated. If the signal is asserted, the V1 flag is set to zero, invalidating the record for $C_{i,j+1}$ for which the calculations are started by the first pipeline stage during the current cycle. The signal also controls the multiplexer in front of the final_els latch and is communicated to the pipeline stage controller.

The critical path of this stage goes through SUB_m, LUT_3, and SUB_{32}. Given that the m-bit subtracter SUB_m is small (typically $m = 7$ or 8) and that the lookup table LUT_3 usually has at most 32 or 64 entries, we estimate that the critical path fits in one machine cycle.

Third stage. During this stage the mask computations are completed. This stage also has two possible modes of operation: *setup* and *process*, which are activated by the *pipeline_status* signal. When the stage operates in the *setup* mode and the stream has a non-standard stride, the look-up table LUT_4 is loaded. When the stage operates in the *process* mode, the following actions are performed in parallel.

- The registers V2 and B2 are copied to the corresponding fields of the appropriate entry of the AG FIFO buffer.
- The look-up table LUT_4 computes the product of the m-bit number contained in the final_els latch and the horizontal stride. The product is routed to the 3-input m-bit adder Add_m which calculates $z = y + ofs_{i,j}$, where y is computed according to the formula presented in Proposition 2. The hardwired $(b \cdot w)$-bit constant $10\ldots0$ is shifted arithmetically to the right by $z \cdot w$ bits, resulting in the mask $msb\_mask(bisze, z \cdot w)$ (see the last term of Equation (8)). By bitwise ANDing this mask with the mask contained in mask1 the final mask is obtained which is written to the corresponding field of the appropriate entry of the AG's FIFO buffer. The multiplication of z by w is implemented by LUT_6. We note that the $(b \cdot w)$-bit shifter Shift_{b*w} is simpler than a common $(b \cdot w)$-bit shifter because shifting can be done only by amounts that are a multiple of the constant w.
- The number contained in the final_els register is shifted to the right by $\log_2(elsize)$ bits, producing the number $final\_els \cdot elsize$, which is written to the *bytes* field of the FIFO entry.

The critical path of this stage goes through the LUT_4, Add_m, LUT_6, Shift_{b*w}, and AND_{b*w} units. We remark that the m-bit adder Add_m is small (typically $m = 7$ or 8) and the lookup tables usually have at most 32 or 64 entries. Given the fact that the Shift_{b*w} shifter is simpler than a common $(b \cdot w)$-bit shifter and the AND_{b*w} unit can be implemented in just one level of gates, we estimate that the critical path fits in one machine cycle.

This concludes the description of the AG. The presented pipelined implementation is able to produce a new AG record every cycle. We observe that since the *change_row* signal is generated during the second stage, a one-cycle bubble will appear in the pipeline when a new row is started. However, the *change_row* signal is, in fact, generated by the carry bit produced by the Sub_{32} unit of the second stage. The carry can be produced before the subtraction itself has finished and, therefore, the signal can become available before the end of the cycle. Since the unit labeled "&" that calculates the block address of the first block in the next row $B_{i+1,1}$ consists of just one level of logic gates, the bubble can be removed if an extra 32-bit adder which calculates during every cycle $A_{i+1,1} = A_{i,1} + VStride$ is added to the first AG pipeline stage and connected to the unit labeled "&".

6. Conclusions

In this paper we described how a CSI execution unit can be implemented. We presented a general organization of such a unit as well as a detailed description of a unit that is attached to the first-level data cache. The decision to interface CSI unit to the L1 cache was motivated by particular characteristics of typical multimedia applications, such as MPEG-2 and JPEG coders/decoders, which exhibit high cache hit rates and, furthermore, require high data throughput. In the presented implementation, a whole cache block is transferred to or from the unit in a single access. When hit rates are high, such a design can provide a high data throughput without increasing the number of cache ports, thus satisfying the needs of streaming multimedia applications. We analyzed the computations needed for the generation of address information. Although these computations turn out to be complex, we have shown that they can be performed in a pipelined fashion. We described a detailed design of a pipelined address generation unit that can compute the address information for a new cache block every machine cycle with the latency of three cycles.

In our previous work [7–10] we used a near cycle-accurate simulator where it was assumed that the latency of an AG is one cycle. We have performed experiments assuming a three-cycle latency and observed that the latency of a CSI instruction usually increases with just two cycles. This is to be expected since AG calculations are overlapped with useful computations and the increased AG latency is incurred only for the first access. Taking into account that CSI instructions typically take tens to hundreds of cycles, an increase of two cycles is insignificant.

There are several directions for future research on the CSI architecture. First, we intend to develop a VHDL model of the CSI execution unit in order to obtain more precise estimates of its area and delay. Second, the issue of compilation from a high-level language to CSI instructions should be addressed. We expect that this problem can be solved. CSI instructions have two main features: operands are vectors, and individual vector elements can have short integer data types (8- and 16-bit) and require packing and unpacking. Vectorizing compilers have shown the possibility of automatic vectorization. Furthermore, recently, Intel provided a compiler for MMX/SSE, showing that compilation for an instruction set which supports short integer data types is feasible [14].

Acknowledgements

We thank the anonymous referees for their comments which helped to improve the presentation of this paper.

References

[1] A. Peleg, S. Wilkie, U. Weiser, Intel MMX for multimedia PCs, Communications of the ACM 40 (1) (1997) 24–38.
[2] S. Thakkar, T. Huff, The Internet streaming SIMD extensions, Intel Technology Journal (May) (1999).
[3] P. Ranganathan, S. Adve, N. Jouppi, Performance of image and video processing with general-purpose processors and media ISA extensions, in: Proc. Int. Symp. on Computer Architecture (ISCA), 1999.
[4] N. Slingerland, A. Smith, Measuring the performance of multimedia instruction sets, IEEE Transactions on Computers 51 (11) (2002) 1317–1332.
[5] J. Hennessy, D. Patterson, Computer Architecture—A Quantitative Approach, second ed., Morgan Kaufmann, 1996.
[6] S. Palacharla, N. Jouppi, J. Smith, Complexity-effective superscalar processors, in: Proc. Int. Symp. on Computer Architecure (ISCA), 1997.
[7] B. Juurlink, D. Tcheressiz, S. Vassiliadis, H. Wijshoff, Implementation and evaluation of the complex streamed instruction set, in: Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 2001.
[8] D. Tcheressiz, B. Juurlink, S. Vassiliadis, H. Wijshoff, Performance of the complex streamed instruction set on image processing kernels, in: Proc. Euro-Par'01, 2001.
[9] D. Cheresiz, B. Juurlink, S. Vassiliadis, H. Wijshoff, Architectural support for 3D graphics in the complex streamed instruction set, in: Proc. Int. Conf. on Parallel and Distributed Computing and Systems (PDCS 14), 2002.
[10] D. Cheresiz, B. Juurlink, S. Vassiliadis, H. Wijshoff, Performance scalability of the multimedia instruction set extensions, in: Proc. Euro-Par'02, 2002.
[11] E. Hakkennes, S. Vassiliadis, Multimedia execution hardware accelerator, Journal of VLSI Signal Processing 28 (3) (2001) 221–234.
[12] N. Slingerland, A. Smith, Cache performance for multimedia applications, in: Proc. 15th Int. Conf. on Supercomupting (ICS), 2001.
[13] S. Vassiliadis, B. Juurlink, E. Hakkenes, Complex streamed instructions: introduction and initial evaluation, in: Proc. EUROMICRO 26, 2000.
[14] A. Bik, M. Girkar, P. Grey, X. Tian, Automatic intra-register vectorization for the intel architecture, International Journal of Parallel Programming 30 (2) (2002) 65–98.

Dmitry Cheresiz was born in 1974 in Novosibirsk, Russia. In 1991 he started his studies at the Faculty of Mathematics and Mechanics of Novosibirsk State University and graduated in 1995 (cum laude). From 1997 till 2002 he was a Ph.D. student at the Computer Science Department of Leiden University. Currently, he is a postdoc at the Computer Engineering Lab of TU Delft. His research interests include computer architecture, vector and multimedia processing, of application-specific instruction sets, and processor design, modelling, and performance evaluation.

Ben Juurlink, born in 1968, received an M.S. degree from Utrecht University, The Netherlands, in 1992, and obtained a Ph.D. degree from Leiden University, The Netherlands, in 1997, both in computer science. From January 1997 until July 1998, he worked at the Heinz Nixdorf Institute in Paderborn, Germany. In September 1998 he joined the Computer Engineering Laboratory of the Department of Electrical Engineering of Delft University of Technology, The Netherlands, where he is currently an assistant professor. His research interests include parallel algorithms, architectures, programming and cost models, application-specific ISA extensions, multiple-issue (superscalar and VLIW) processors, low power techniques, and hierarchical memory systems.

Stamatis Vassiliadis was born in Manolates, Samos, Greece in 1951. He is currently a chair professor in the Electrical Engineering department of Delft University of Technology (TU Delft), The Netherlands. He had also served in the EE faculties of Cornell University, Ithaca, NY and the State University of New York (S.U.N.Y.), Binghamton, NY. He worked for a decade with IBM where he had been involved in a number of advanced research and development projects. For his work he received numerous awards including 24 publication awards, 15 invention awards and an outstanding innovation award for engeneering/scientific hardware design. His 70 USA patents rank him as the top all time IBM inventor. In 1992 he received an honorable mention best paper award at the ACM/IEEE MICRO25. He received the best paper awards in the IEEE CAS (1998 and 2002), IEEE ICCD (2001) and PDCS (2002). Dr. Vassiliadis is an IEEE fellow.

Harry A.G. Wijshoff was born in 1960 in Grevenbicht, the Netherlands. He received the M.S. degree (cum laude) in Mathematics and Computer Science in 1983, and the Ph.D. degree in 1987, from Utrecht University, the Netherlands. From 1987 untill 1990 he was a visiting senior computer scientist at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He was an associate professor at the Department of Computer Science at Utrecht University untill 1992. Currently, he is a professor of computer science at the Leiden Institute of Advanced Computer Science, Leiden University, the Netherlands. His current research interest include performance evaluation, sparse matrix algorithms, programming environments for parallel processing and optimizing compiler technology.