Chap - A SIMD Graphics Processor

Adam Levinthal
Thomas Porter

Computer Graphics Project
Lucasfilm Ltd.

ABSTRACT

Special purpose processing systems designed for specific applications can provide extremely high performance at moderate cost. One such processor is presented for executing graphics and image processing algorithms as the basis of a digital film printer. Pixels in the system contain four parallel components: RGB for full color and an alpha channel for retaining transparency information. The data path of the processor contains four arithmetic elements connected through a crossbar network to a tessellated scratchpad memory. The single instruction, multiple data stream (SIMD) processor executes instructions on four pixel components in parallel. The instruction control unit (ICU) maintains an activity stack for tracking block-structured code, using data-dependent activity flags for conditional disabling subsets of the ALUs. Nested loops and if-then-else constructs can be programmed directly, with the ICU disabling and reenabling ALUs on the basis of their individual status bits.

CR Categories and Subject Descriptors: B.2.1 [**Arithmetic and logic structures**]: design styles — parallel; B.3.2 [**memory structures**]: design styles — interleaved memories; C.1.2 [**processor architectures**]: Multiple data stream architectures — SIMD. I.3.1 [**computer graphics**]: Hardware architectures — Raster display devices. I.4.0 [**Image Processing**]: General — Image displays.

General Terms: Design

Additional Key Words and Phrases: digital film printers, compositing, computer graphics, parallel processing, SIMD architecture, tesselation.

1. Introduction

The Lucasfilm Pixar project is producing high-performance machines for film-quality image creation. The first machine to be completed is a digital film printer that provides digital processing capabilities for special-effects film production. The system, called the Lucasfilm *Compositor,* is a digital realization of a conventional optical film printer under computer control [2].

A laser printing/scanning system replaces the projectors and process camera of an optical printer. A high-speed digital processor brings digital signal processing techniques to bear on each frame, extending the range of capabilities of a conventional optical film printer. These include:

- Merging multiple images to form a single image, handling partially transparent objects and edges;
- Creation of mattes for live-action blue screen shots;
- Hand touchups and simple creation of *garbage mattes;*
- Filtering to provide for diffusion, tinting, highlighting, defocusing, and edge enhancement;
- Color correction to account for non-linearities and crosstalk between the dye layers of color film;
- Rotation and perspective transformation of frames to correct for original camera misalignment or to simulate complex camera moves.

The Compositor is designed to execute a number of key algorithms at an average rate of one microsecond per pixel. This performance allows for interactive use on limited resolution images, as well as acceptable performance for production work on high resolution, movie-quality images.

The heart of the system is the Channel Processor (Chap), a programmable pixel processor for performing all the computation and controlling the flow of pixels in the Compositor. The Chap is based on a four operand vector data pipeline operating with a single instruction sequencer. This design was adopted to take advantage of the four component data structure used to represent digital images in the Compositor.

2. 4-Channel Pictures

Pixels contain four components. Frame buffers and disk files contain red, green, and blue color channels as well as an *alpha* or *matte* channel. The matte channel is used to specify transparency so that elements which do not cover an entire frame can be stored separately for later compositing. An alpha of 0 is interpreted to mean full transparency; an alpha of 1 indicates full coverage.

As presented in [7], the color channels are stored *premultiplied* by the alpha channel in the sense that half coverage of a pixel by a yellow object is stored as (.5,.5,0,.5), not (1,1,0,.5). This choice puts each channel on equal footing: so that most algorithms that process RGBA pictures can treat the alpha channel with precisely the same instructions as the color channels.

The SIMD architecture was originally considered based on the fact that many algorithms execute identical operations on all four components of each pixel. This is true of many key algorithms in the digital film printing process, including color correction of images, scaling and translation to align images, and merging of images during the compositing operation. Some algorithms however do not perform identical operations on each component. In particular, the matte algorithm [6] computes a final alpha value based on the initial values of the RGB component at each pixel.

The crossbar connection between memory and processors in the Chap architecture was introduced to support different possible approaches to pixel processing. Using the crossbar mechanism, Chap programs can be structured to access the *four components* of a *single pixel*, or the *same component* of *four consecutive pixels*.

2.1. 12-bit Channels

For film applications, 8-bit linear intensity values are insufficient. Consequently, in the prototype system, the frame buffer memory banks are configured with 12 bits per channel. The Chap internal scratchpad memory and registers are 16 bits wide to maintain extra precision for intermediate products. The frame buffer memory will support up to 16 bits per channel, and it is anticipated that this precision will be required in future applications.

The 12-bit values stored in frame buffer memory are skewed slightly upon access as shown in table 1.

12 Bit Value	Sign extended Value	Range
10xxxxxxxxxx	000010xxxxxxxxxx	(1.5, 1.0]
01xxxxxxxxxx	000001xxxxxxxxxx	(1.0, 0.5]
00xxxxxxxxxx	000000xxxxxxxxxx	(0.5, 0.0]
11xxxxxxxxxx	111111xxxxxxxxxx	(0.0, -0.5]

Table 1

This skewing means that sign extension is performed based on the uppermost two bits of the 12 bit components, instead of simply copying the most significant bit directly. This format was chosen in order to represent values in the range (1.5,-.5], providing 11 bits of fraction and sufficient range for underflow and overflow. As importantly, this representation assures an accurate representation of unity.

All of the algorithms critical to compositor operation can be handled with integer arithmetic; in fact, 16-bit integers are sufficient for most general image processing work, though 32 bits are needed for the accumulation of intermediate products in digital filtering.

3. Compositor Overview

Figure 1 shows the communication channels inside the Compositor.

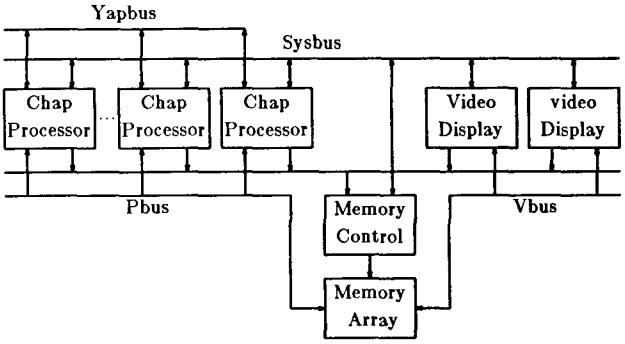

Figure 1: Compositor Communications Channels

The Chap communicates through three separate data paths:

- The *Pbus*, or Processor Access Bus, is the common data bus between the processors and framebuffer memory. The extremely high data rates on the Pbus (240 Mbytes/sec) allow multiple processors to operate in parallel on common framebuffer data. The Chap hardware DMA controller allows data to be transferred between processor memory and framebuffer memory with minimal processor overhead.
- The *Yapbus* (Yet Another Pixar Bus) is a high bandwidth (80 Mbytes/sec) data channel operating as a local area network between compositor system components. In particular, the Yapbus provides the data channel to the laser printer/scanner system.
- The *Sysbus*, or System bus, is a low bandwidth (2 Mbytes/sec) control interface between the host computer and compositor subsystems. Operating parameters, microcode instructions, and diagnostic commands are transferred to the Chap through the system bus interface.

The high bandwidth I/O channels of the compositor system contribute to the Chap processor's ability to operate as a high performance, general purpose, image processor.

4. Processor Architecture

The Chap is a microcoded four operand parallel vector processor. It operates as a single instruction, multiple data stream (SIMD) processor, executing each instruction on four operands at the same time. The processor performs arithmetic operations, operand addressing, I/O operations, and program sequencing in parallel with a highly horizontal instruction word. This parallelism, along with extensive pipelining, allows the Chap to achieve performance approaching 64 MIPS.†

The Chap processor block diagram is shown in Figure 2.

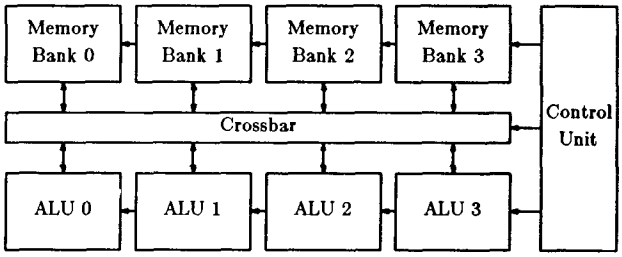

Figure 2: Chap Block Diagram

Four arithmetic elements (processors) are connected through a crossbar network to four Scratchpad memories. The processor data path is optimized for 16 bit integer arithmetic. All processor and memory operations are controlled by a single microcoded control unit. During each micro-cycle, all four processors receive the same instruction. Special condition code and memory control logic allows individual processor and memory operations to be conditionally inhibited during certain program segments.

The instruction control unit is designed to support common program control constructs such as *while(loopvariable)*, *if(condition)*, *if(condition)/else*, etc. Scalar (uniprocessor) support is provided in the data path to allow any of the processors to access non-vector parts of the machine.

4.1. Processor Elements

Each processor element is of the form shown in Figure 3, with a 16-bit arithmetic unit and 16-bit multiplier.

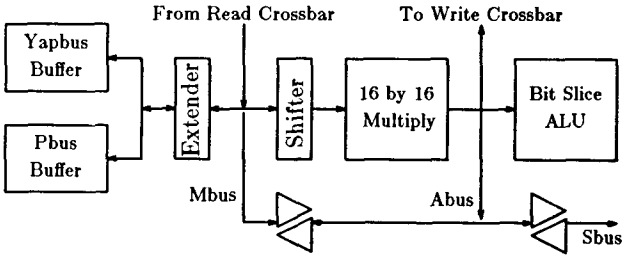

Figure 3: Chap Processing Element

The multiplier and ALU can operate in parallel for certain operations such as multiply/accumulate. Data is loaded into the multiplier over the Mbus, and data is loaded into the ALU over the Abus. Each arithmetic element can read and write the scalar (control) parts of the machine over the Sbus.

The arithmetic-logic units are bipolar 'bit slice' devices with 32 working registers, an accumulator, and a status register. Although a single multiplexed I/O port is used to load and store from the ALUs, internal latches allow reading external operands and writing results to external destinations (memory to memory, for example) in the same ALU instruction.

The multipliers are designed to multiply component values by alpha values and components by coefficients, producing component values as the product. In order to produce properly aligned and rounded component values in the top 16 bits of the product, multiplicands must be shifted up to three places. This is accomplished by shifter circuitry external to the multiplier inputs. The multiplier product is a full 32 bit number; However, the bus structure allows access to only 16 bits of the product in any instruction.

4.2. Scratchpad Memory

Four scratchpad memories provide 64K 16 bit words of general purpose memory for program data storage. The address calculation unit uses a register file containing pointer values to specify data elements in Scratchpad memory. Programs use these pointer to reference four operands (pixel data) or one operand (e.g., filter coefficients) during each access to Scratchpad. An arbitrary offset can be added to a pointer value at each instruction to allow sequential access of memory.

A single pointer value can be used to access four operands by ordering data in memory using a special processor-memory connection network. This network, called the memory *crossbar* allows *tessellated* access to the Scratchpad memory [8]. A number of tessellated access formats appropriate for graphics processing support are built into the crossbar network. The access format is specified in the Chap instruction word as in Table 2.

n	format
00	pixel
01	component
10	broadcast
11	indexed

Table 2

Pixel access is the normal access mode to scratchpad pixel data. Each access references a full pixel in parallel, with the red component going to the red processor, green to the green processor, etc.

Component access, where each processor works on a separate pixel, references a single component from four consecutive pixels in parallel. This format is provided to support vector processing in algorithms that do not operate on all components of a single pixel in parallel.

† This figure represents the peak rate for data moves in the Chap, where four operands are transfered every 62ns. The chap can perform multipies or ALU operations, as well as multiply accumulates, at a sustained 32 MIPS.

Broadcast access allows each of four processors to receive a single memory element. This is useful for operations where one scratchpad coefficient is sent to all four multipliers.

Indexed access uses a computed value from each processor as an index into a scratchpad table. indexed access is useful for color mapping applications, where each component is mapped from a different table.

Table 3 shows the tessellated nature of the scratchpad memory. The memory is actually four memory banks (S_0,S_1,S_2,S_3) partitioned in such a fashion as to optimize either Pixel (R_p,G_p,B_p,A_p) or Component (C_p,C_{p+1},C_{p+2},C_{p+3}) access, where p is the effective pixel address and C is some particular component. Notice that neither case causes contention by attempting multiple accesses to a single bank of memory.

	S_0	S_1	S_2	S_3
0	R_0	G_0	B_0	A_0
1	A_1	R_1	G_1	B_1
2	B_2	A_2	R_2	G_2
3	G_3	B_3	A_3	R_3
4	R_4	G_4	B_4	A_4
...	...	...	...	...

Table 3

5. SIMD Control

The Chap performs single-instruction, four-component processing. This type of architecture has been used succesfully in previous arithmetic processors [1][5]. We have found that many image processing problems can be solved with identical code for each of the RGBA components of every pixel.

There are occasions, however, where the ability to suspend some subset of processors over a range of instructions is desired. For example, some programs might switch to single processor operation to find a coefficient in a table and then switch back to four processor operation to multiply that coefficient by each component. Clamping is a another example; when clamping, we wish to set each processor's accumulator to unity only for those accumulators which exceed unity. We make the comparison to unity, suspend if less, set to unity, and resume each processor.

To support this conditional processing, the Instruction Control Unit (ICU) includes not only the standard *sequencer* functions for finding the next instruction, but also *runner* functions for determining which processors execute each instruction. Let us review the capabilities of the ICU, to illustrate its operation.

The Chap executes instructions stored in writable instruction memory. The first job of the ICU is to decide where the next instruction lies. Along with the standard sequencer opcode and condition select information, Chap ICU instructions specify a *Source Processor* qualifier. Thus, instructions like "jump (*opcode*) if alpha (*processor*) is zero (*condition code*)" are possible. The jump address may be computed, or included as a literal in the instruction word.

The ICU must also direct the flow of execution for individual processors based on conditions involving their own execution. It becomes necessary to suspend some processors while others execute a particular program segment. Note that this is somewhat complex when we consider nested if-then-else constructions and procedures.

To provide a mechanism for controlling individual processor activity, The ICU maintains a *runflag* register and a *stack* that contains runflag vectors at each program level. The current runflag indicates which processors are running and which are suspended.

At conditional test instructions, a four-bit condition vector is created, one bit for each processor, corresponding to that processor's condition with regard to the ICU condition select field. The logic of the ICU determines the current runflag and maintains the runflag stack based on this condition vector and immediate runflag bits in the instruction word.

Standard if-then, if-then-else, and while-do programming constructs are translated into runner instructions as shown in table 4.

action	C construct	assembler
push	{	push
pop.	}	done, fi
push; R &= C	if () {	if *e* then
R = (!R) & S0	} else {	else
R &= C	while () {	while *e* do

Table 4

R refers to the current runflag register; S0 is at the top of the runflag stack; C is the new condition runflag.

The complete set of 16 conditional ICU instructions expands on these five runner instructions. The complete set controls three concurrent stacks: the runflags, return addresses, and loop counters.

Let us consider the C program fragment:

```
if (c1) {
    if (c2) {
        do0;
    } else {
        do1;
    }
} else {
    do2;
}
```

Assume that condition c1 generates a runflag of 1110, suspending processor 3, and condition c2 generates a runflag of 1100. Table 5 shows the state of the current runflag R and the top two runflag stack locations after each ICU instruction.

ICU instruction	R	S0	S1
calculate c1	1111	....	....
push; R &= c1	1110	1111	....
calculate c2	1110	1111	....
push; P &= c2	1100	1110	1111
do0	1100	1110	1111
R = (!R) & S0	0010	1110	1111
do1	0010	1110	1111
pop	1110	1111	....
R = (!R) & S0	0001	1111	....
do2	0001	1111	....
pop	1111	....	....

Sequencer conditions are specified with an *any/all* qualifier as well as the normal true/false qualifier of typical instruction sequencers.

This allows a while loop which runs until all four processors are satisfied, suspending individual processors as it goes, to be simply stated:

```
while any (c1) do
    statement;
done
```

and translates into:

```
    push
loop: calculate c1
    R &= c1; jump out if all disabled
    statement
    jump loop
out: pop
```

The previous example, that of clamping accumulator values that exceed unity, takes the form:

```
if any alu positive then
    acc = 1;
fi
```

and translates to:

```
temp = acc - 1 ;
push; r &= positive;
acc = 1;
pop
```

The appendix contains code fragments from actual Chap programs. Code fragment (1) illustrates the inner loop for a scanline linear interpolate of the form:

$$Target_0 = Source_0 + (1 - \alpha_0) Source_1$$

6. Chap Programming

The wide Chap instruction word can be split into six parts, offering control over the ALUs, multipliers, data paths, scratchpad address calculation unit, crossbar tessellation, and instruction control unit. The assembler [4] provides a powerful syntax for maintaining control. The runtime monitor [5] provides linking and loading facilities to promote modular program development. Pipeline delays built into the hardware modules complicate programming, but become a distinct advantage (over unpipelined designs) when writing optimized standard modules. Features of the machine and the assembler allow the programmer to stretch out the instruction timing and overlook the pipeline delays when developing code for the first time.

6.1. Chap/Host Interface

A number of features allow the Chap to support host interaction during program execution.

- 16 words of shared memory are used for parameter passing.
- 4096 'virtual register' locations are decoded in the Sysbus interface which allow the host to initiate Chap processes by reading and writing memory-mapped function registers.
- Interrupt logic allows the Chap to interrupt the host under program control.

7. Conclusion

We have described a digital processor specifically designed to support digital pixel processing, providing parallel vector arithmetic with convenient programming language support. The SIMD architecture appears well suited to the particular algorithms used in the digital film printing process.

8. Acknowledgments

Particular credit should go to Loren Carpenter, who originally suggested the basic four channel SIMD architecture, and contributed to elements of the design throughout the project.

Special thanks is also due to Mark Leather, who shared in the final design stages and debugged the prototype design, writing much diagnostic software in the process to verify the design.

Rodney Stock should be credited for his role as hardware manager, as well as logic designer, on the Pixar system, and should be identified as one of the principal architects of the Pixar system.

In addition, the processor working group of Bill Reeves, Tom Duff, and Sam Leffler, provided many important criticisms and suggestions during the design and review phases.

Sam Leffler should also be credited for his work in providing the assembler and monitor that we use to develop Chap code.

Thanks to Andy Moorer and Curtis Abbott, who stressed the importance of complete diagnostic support for program debugging, resulting in important improvements to the host interface section.

Thanks also to Tom Noggle for his contribution to the design of the Yapbus network.

APPENDIX

This section is included to provide a little detail about specific issues in generating code for the Chap.

(1) This code fragment is an example of chas assembly code to lerp (linearly interpolate) between two scan lines:

$$Target_0 = Source_0 + (1 - alpha_0) Source_1$$

$Source_0$, $Source_1$ and $Target_0$ are all stored in scratchpad memory. This is an example of assembly code where no instruction overlap is attempted. Comments are delimited by the C styles /* and */ constructs.

```
Lerp: loop Xcount do
      multx = unity - @s0ptr; round
      { multy = (comp)(s1ptr); round;
            s1ptr = s1ptr + p_inc }
      { acc = (s0ptr);
          s0ptr = s0ptr + p_inc }
      { (tptr) = acc + msp;
          tptr = tptr + p_inc }
      done
```

The inner loop can be translated as follows:

/* the "loop *e* do" construct pushes the current address + 1 onto the return stack, and loads the loop counter with the value *e*, in this case a literal value of Xcount from the immediate field */

```
loop Xcount do
```

/* Read from scratchpad in broadcast mode (specified by the @ sign) using pointer s0ptr, and subtract it from unity (a predefined ALU register) and load the results into the multiplier X inputs with the rounding bit set. */

```
multx = unity - @s0ptr; round  /* 3 ticks */
```

/* Read from scratchpad in pixel mode (specified by the () around s1ptr) using pointer s1ptr, and then load that pixel into the multiplier Y inputs shifted up by 2 places (specified by the qualifier (comp)) with the rounding bit set. Furthemore, increment s1ptr by the value defined by p_inc (in this case 4). */

```
{ multy = (comp)(s1ptr); round;
      s1ptr = s1ptr + p_inc }  /* 3 ticks */
```

/* Read from scratchpad in pixel mode using pointer s0ptr and load them into the ALU accumulators, and increment s0ptr by p_inc. */

```
{ acc = (s0ptr);
    s0ptr = s0ptr + p_inc }  /* 4 ticks */
```

/* Add the values in the multiplier most significant result register (top 16 bits) to the current accumulator, and store the results as a pixel in scratchpad at the location pointed to by tptr, and then increment tptr by p_inc. */

```
{ (tptr) = acc + msp;
    tptr = tptr + p_inc }     /* 2 ticks */
```

/* The next statement terminates the loop and causes the PC to be loaded from the top of the stack without popping it. The test for termination is done at the top, and if it fails, execution continues at the point after the done statement. */

```
done
```

(2) The code can be rewritten to take advantage of pipelined operation, with an increase in performance of 50% as shown below. The code takes advantage of the machine pipelining the details of which are beyond the scope of this paper. Suffice to say that the duration of each instruction can be specified in clock ticks, overriding the default assembler durations, and that the first scratchpad read is repeated at the bottom of the loop to maintain the pipelining around the loop. Also note that whereas the previous loop required 16 clock cycles to execute, the following loop executes in 8 clock cycles.

```
        xbar = @s0ptr; push
loop:   { unity = unity - xbar; special;
            spad = (comp)(s1ptr); 1tick }
        { multx = unity - latch;  round;
            spad = (s0ptr); 1tick }
      { multy = xbar; round;
          s1ptr = s1ptr + p_inc; 1 tick }
      { acc = xbar; special;
          s0ptr = s0ptr + p_inc; 1 tick }
      acc = xbar; 1 tick
        acc = acc + msp; special; 1 tick
        { dowhile !lc zero;
            spad = @s0ptr; tptr = tptr + p_inc;
            acc = acc + msp; special; 1tick }
      { (tptr) = acc + latch;
          tptr = tptr + p_inc; 1 tick }
```

References

[1] Barnes, G., et all, The ILLIAC IV Computer. *IEEE Transactions on Computers* Vol C-17, No 8 (August 1968), pp 746-757.

[2] Fielding, R., *The Technique of Special Effects Cinematography.* Hastings House, New York, 1977.

[3] Kubo, M., Taguchi, Y., Agusa, K., Ohno, Y., A multi-microprocessor system for three dimensional color graphics. *Proc of IFIP 80, 1980.*

[4] Leffler, S., Chap Assembler Reference Manual. Technical Memo 98, Computer Division, Lucasfilm Ltd, December, 1983.

[5] Leffler, S., Chap Runtime Monitor Reference Manual. Technical Memo 102, Computer Division, Lucasfilm Ltd, December, 1983.

[6] Porter, T., Matte Box Design. Technical Memo 63, Computer Division, Lucasfilm Ltd, August 1983.

[7] Porter, T., Duff, T., Compositing Digital Images. *Computer Graphics* Vol 18, No 3, 1984, To be published

[8] Shapiro, H. D. Theoretical Limitations on the Efficient Use of Parallel Memories. *IEEE Transactions on Computers,* Vol C-27, No. 5 (May 1978), .