5.3 ISA CRAM System Prototype

An ISA CRAM system, named CS64p1kISA, has been built. It comprises of one C64p1k CRAM chip, an ISA CRAM Controller FPGA, and the controller downloading CPLD and EPROM. It is made of two PCBs, the CRAM system PCB described in Section 5.2.3 (Appendix B) and a very small PCB that consists of an ISA male connector only. The ISA connector PCB is attached to the main PCB by screws, and its ISA signals are soldered to the main PCB External Bus Header using very thin wires. Figure 5.4 shows a photograph of the CS64p1kISA ISA PC card.

The system has been tested in a PC under both Linux and MSDOS. Because of problems with the software driver for the card, all routines tested on the prototype system were written in machine code (generated automatically using CRAM C++ simulator). Also, because of the small size of memory per PE and the small number of PEs, only small algorithms, especially basic arithmetic operations, were run on the system. The main objective of building the prototype was to demonstrate a working model of the CRAM concept rather than to highlight the performance advantages of CRAM. As mentioned earlier, for many applications, the performance advantage of CRAM becomes evident as the number of PEs increases. Therefore, the performance analysis of Chapter 7 is based on simulations of CRAM systems with a realistically large number of PEs.

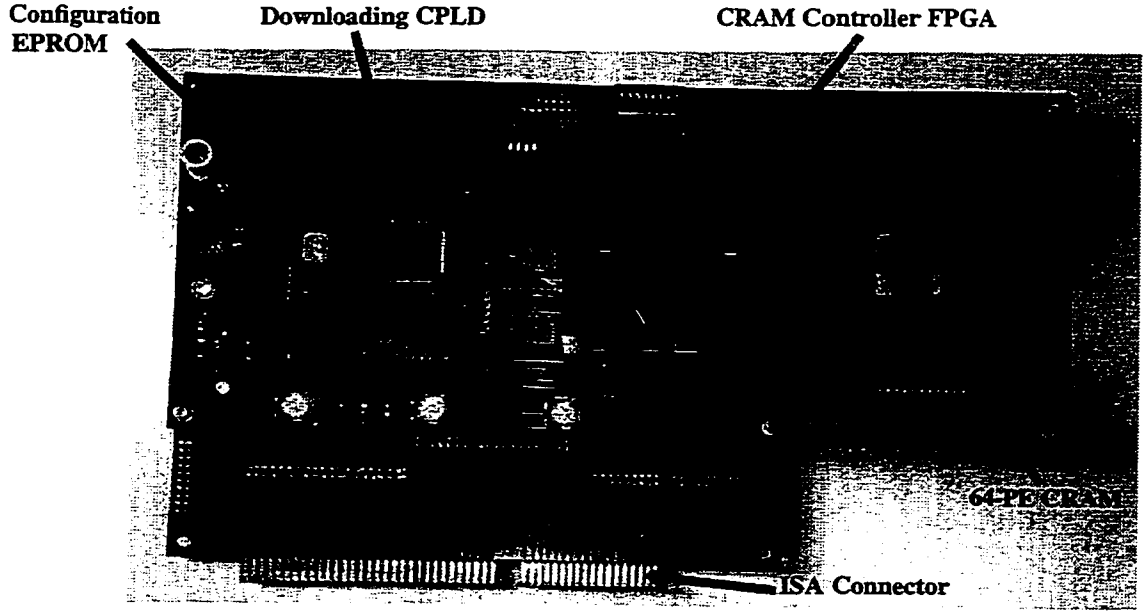

Figure 5.4 CRAM System ISA Card

5.4 Summary

A VHDL synthesis design flow was used in the design of the CRAM controller in order to reduce design time and provide a generic design that can easily be targeted for implementation in different technologies. A CRAM system VHDL simulation model has been designed to allow the running of actual system assembly or machine code when testing the functionality of the controller and CRAM chips. This reduces the time of generating input test vectors and collecting test output. The simulation model consists of VHDL models of the controller, the CRAM chips, a generic host processor, and host system buses, as well as a couple of C++ tools for processing text-file models of CRAM microinstructions and CRAM/host code.

Two controller prototypes have been implemented in a Xilinx XC4013EPQ240-2 FPGA. The ISA CRAM controller, with a 192-word control store, uses 564 CLBs and runs at 14 MHz. Because of area constraints, the PCI CRAM controller was implemented with a control store of 128 words. It uses all the 576 CLBs, and runs at 10 MHz. In TSMC 0.35 μm CMOS technology, the CRAM controller has an area of just over 12000 gates (2-input NAND gate equivalents) and runs at 90 MHz (limited by the cycle time of SRAM cores). To demonstrate a working model of the whole CRAM concept, a 64-PE ISA CRAM system prototype has been built and tested in a 133 MHz Pentium PC under both Linux and MSDOS.

Chapter 6

CRAM System Software Tools

This chapter describes the high-level software tools for application programming, software development, and system simulation. Section 6.2 describes the CRAM C++ Compiler, which is a CRAM C++ library that allows the use of the standard C++ language and standard C++ compilers when writing CRAM programs. Section 6.4 describes the CRAM assembly code, and Section 6.5 briefly describes a high-level tool for developing CRAM microcode. The CRAM C++ Simulator, a tool that is used to simulate the behavior of a CRAM system, is described in Section 6.7. This is used to analyze applications, software tools, and CRAM architectural features. Apart from software tools, this chapter also discusses two other software issues: data transposition (Section 6.3) and grouping of microroutines (Section 6.6). Data transposition is used to convert the format of data between the bit-serial CRAM and the bit-parallel host computer. Microroutine grouping is used to reduce the number of microinstructions in the control store so that a smaller microprogram memory can be used.

6.1 Introduction

The lowest level CRAM instructions are the control bits issued to the CRAM chip by the CRAM controller. These include the PE opcodes (COP and TTOP), as well as the control, data and address signals. These are stored as microinstructions in the CRAM controller control store, and are collectively referred to as the CRAM Machine Language.

Writing application programs using the CRAM machine code requires detailed knowledge of the architecture of the CRAM chip and the controller. Also, like any other machine code, it is both tedious and slow to program using these instructions. Therefore, high-level software tools have been developed that programmers and designers can use when writing CRAM applications or other software development tools. This also includes a C++ CRAM simulator that can be used by programmers for testing and timing CRAM applications. The simulator is also a vital tool for CRAM hardware and system designers in that it allows different architectural features to be explored before committing to the actual hardware implementation. Figure 6.1 shows the relationship and use of the CRAM software tools.

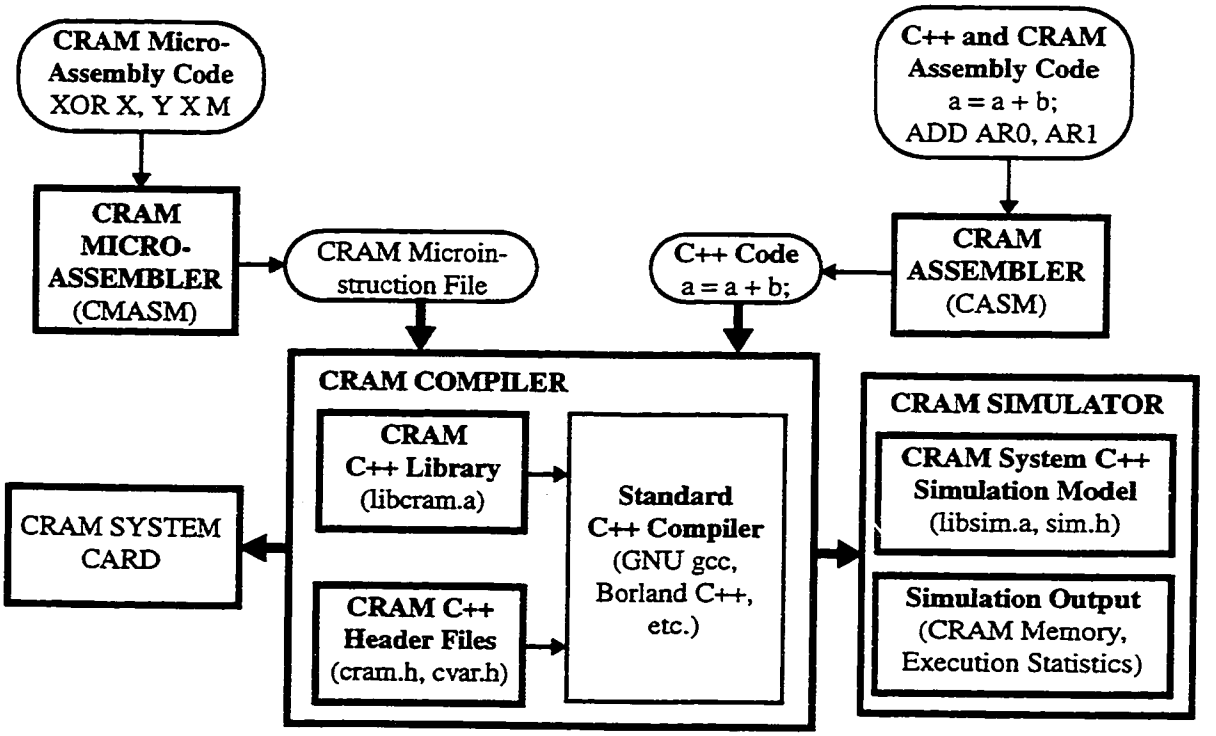

Figure 6.1 CRAM System Software Tools

CRAM applications can either be written in C++ code and compiled using a standard C++ compiler, or they can use both C++ and CRAM assembly code which is preprocessed through the CRAM Assembler before compilation. The CRAM Microassembler is a high-level tool for generating CRAM microcode. Note that the VHDL simulator described in Section 5.1.6 is used primarily for verification of the behavioral and synthesized VHDL controller design and is therefore not considered as a high-level tool for use by application programmers. It is therefore not included in Figure 6.1.

6.2 CRAM Compiler (CRAM C++ Library)

6.2.1 Using a C++ Compiler

One of the most negative attributes that SIMD machines are tagged with is that they are difficult to program. Because of this, designers of most well known SIMD machines have gone to great efforts to develop very elaborate software tools for their machines. There are three major approaches that can be followed. The first is to design the programming language for the machine from scratch. An example is the programming language for STARAN [43]. The second approach is to design a dialect of a common standard programming language such as C, Pascal or Fortran. This is a popular approach in most very high performance SIMD machines. Examples of such languages include the MasPar Fortran (MPF) and MasPar C (MPC) [44], the Connection Machine C*™ language [33], and the Terasys PIM data parallel bit C (dbc) [13]. The third approach is to use C++ as the programming language for the SIMD machine, with the addition of specialized libraries [4], [14].

The C++ programming language [45] allows creation of new classes of objects. These are actually new data types. The class constructor allows specialized initialization or operation when an object of a class is created. Such initialization might include allocating memory for the object. The class destructor is invoked when the object goes out of scope and is to be destroyed. This can be used in operations such as memory deallocation. The other important characteristic of C++ is operator overloading, which allows the use of standard C++ operators such as +, -, /, for any C++ class. By using operator overloading,

class objects can be used in expressions in exactly the same way as built-in data types. The third characteristic of C++, called inheritance, allows classes to be built from existing classes, thereby inheriting all the characteristics of the base class.

We have chosen to use C++ as the programming language for CRAM because we don't have to build a compiler, and hence the development cycle is short. It is also easier to upgrade since this only requires the addition of new classes or extension (adding new class functions, members, etc.) of existing classes. More important, C++ is easy to learn and use since many application programmers will already have used it before.

6.2.2 CRAM Classes (Data Types)

The CRAM C++ library contain classes that represent CRAM parallel integer variables. Because of the complexity of manipulating floating-point numbers in bit-serial format, and the fact that our current prototypes have very small memory per PE to consider using CRAM for floating-point operations, the design of CRAM C++ classes for parallel floating-point variables has been left for future work. Figure 6.2 shows the physical definitions and characteristics of CRAM data types. The following sections describe the CRAM classes.

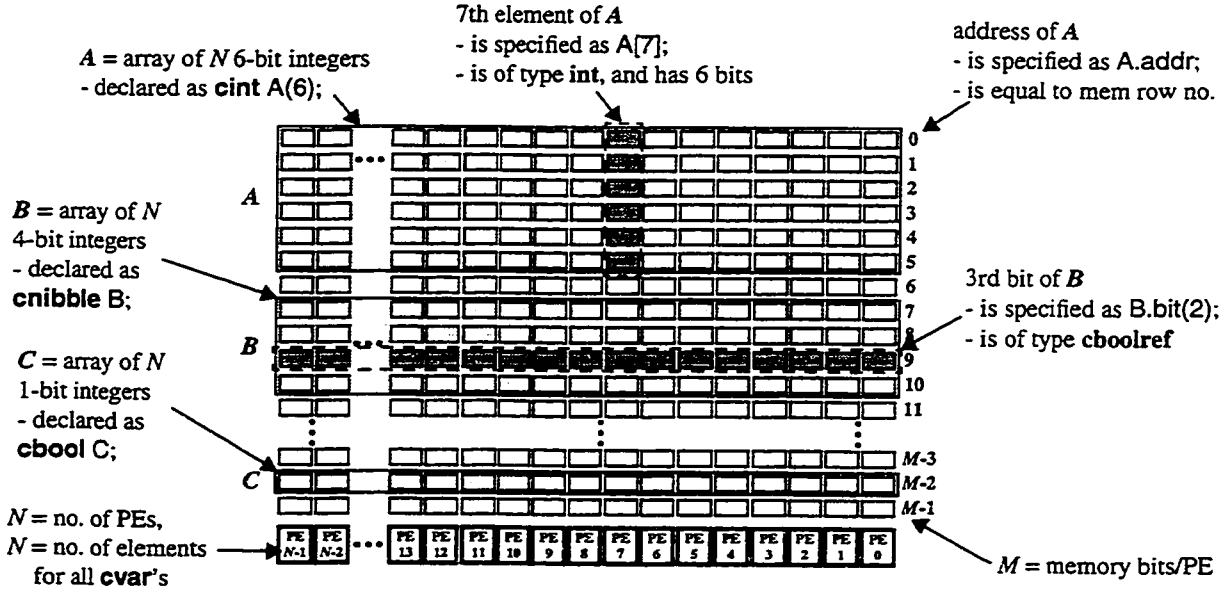

Figure 6.2 Definitions of CRAM Data Types

- **cvar:** The basic class from which all CRAM classes are derived is `cvar`. It represents an array of CRAM parallel integer variables. Its main member variables/functions include:
  - *bits* - The number of bits of the variable. This is limited by the size of memory/PE.
  - *addr* - The address of the variable in CRAM memory. This is assigned by the `cvar()` constructor when the variable is being created.
  - *type* - signed or unsigned integer.

  `cvar` also contains the functions for all the C++ overloaded operators and functions. These are described in Section 6.2.4 and Section 6.2.5.

- **cint** and **cuint:** These classes are descendants of `cvar` and represent signed and unsigned CRAM integer variables. `cint` and `cuint` only differ by *type* (`cint` is `CVAR_SIGNED` and `cuint` is `CVAR_UNSIGNED`). The size (*bits*) of the integer can be any positive number less than PE memory size. However, if *bits* is not specified, it defaults to `sizeof(int)` to mimic integer variables on the host computer system. This is usually 32 on most systems. The following shows the declaration syntax for `cint` and `cuint` variables.

Declaration Syntax

```
cint a;                 // a is a 2-bit signed CRAM integer
cuint sum1(5);          // 5-bit unsigned
cuint error(INIT_0);    // 32-bit unsigned, initialize all its elements to zero
cint b(8, INIT_1);      // 8-bit signed interger, initialize all bits to 1 (0xFF)
```

- **cbool** and **cboolref:** `cbool` is a CRAM data-parallel boolean variable. Unlike on standard computer systems where boolean variables are usually implemented as 8-bit (`char`) variables, on CRAM, `cbool` is, as it should be, a 1-bit variable. Like `cint` and `cuint`, `cbool` is a descendant of `cvar`. It however has extra functions to handle logical operators used to combine relational operators (`||`, `&&`, and `!`). `cboolref` is a pointer to a bit of a `cvar` variable. It is not a physical object in CRAM memory, but it contains all the information about the address of a bit of `cvar`. This is an important class in optimizing operations involving `cbool` or 1-bit `cvar`, especially where there is no need to create a `cbool` variable. Otherwise, `cboolref` behaves exactly like `cbool`. The declaration syntax for CRAM boolean variables is shown below.

Declaration Syntax

```
cbool a;                // a is a CRAM boolean variable
cbool a = 0;            // initialize a to 0 on declaration
cbool mask1(INIT_1);    // CRAM boolean, initialize to 1's
cuint d;                // 32-bit CRAM unsigned integer
cboolef c(d.addr);      // c points to bit 0 of d
cboolef e = d.bit(3);   // e points to bit 3 (4th bit) of d
```

- **cchar, cshort, clong, cnibble:** These are fixed bit-length CRAM integer variables. They are also direct descendants of `cvar`. Like `cint`, they all have corresponding unsigned integer classes (`cuchar`, `cushort`, `culong`, and `cunibble`). The sizes (*bits*) of these classes are derived from the sizes of their corresponding C++ built-in data types. For example, the size of `cchar` is calculated from `CHAR_BIT*sizeof (unsigned char)`. On most machines, `cchar` would be 8-bit, `cshort` 16-bit, `cint` 32-bit, and `clong` would also be 32-bit. `cnibble` is a 4-bit CRAM integer variable. `cchar`, `cshort`, `clong` and `cnibble` have only one constructor with no parameters. In other words, you can not specify the number of vectors, nor can you initialize them on creation (you can always initialize them after they have been declared, without any increase in execution time). The first reason for having fixed-length classes is to allow programmers to declare variables in exactly the same way as is done for built-in C++ data types (no parameters in the constructor). The second reason, and the more important of the two, is that C++ does not allow declaring arrays if the class declaration (constructor) requires parameters. For example, the declaration `cuint a(8)[10]` is a syntax error, but `cuint a[10]` is correct. Therefore, where there is need to declare arrays, the programmer can use data types that do not require parameters. Since these classes cover only bit-lengths of 4, 8, 16, and 32, programmers can create any fixed bit-length CRAM integer class by using the declarations and functions of these classes. Declaration syntax for these variables is shown below:

Declaration Syntax

```
cchar a;                // 8-bit signed cint
cushort diff = 4;       // 16-bit cuint, initialize all elements to 4
clong d[10];            // an array of 10 32-bit cint's
cnibble rec[8][12];     // a 2-dimensional array of 96 4-bit cint's
```

6.2.3 Memory Allocation

Any call to the `cvar` constructor invokes a CRAM memory allocation function. Since all classes of CRAM variables are descendants of `cvar`, the `cvar()` constructor is always called when objects of such classes are being created. In CRAM, variables are not required to be aligned to any particular address boundaries. There is one exception to this rule. A variable cannot be allocated memory that crosses over the 256*n* address boundary (*n* = 1, 2, 3,...). For example, a 6-bit `cint` variable cannot be allocated CRAM memory at CRAM row 254 because this will require its end address to be 259 (crossing over 256). This is so because the only part of the CRAM address that can be incremented or decremented by the controller to move through the bits of a CRAM variable is 8-bit. This is the part of the address that is stored in the AR0-AR1 registers (AXn registers are fixed).

When a CRAM variable goes out of scope, or when the `cvar` destructor (`~cvar()`) is explicitly called, the variable is destroyed and its memory is deallocated. Deallocated memory is returned to the CRAM free memory heap if the variable was at the end of the allocated memory. Otherwise it is returned to the fragmented garbage memory. A new variable is allocated memory in the fragmented garbage memory if it can fit in any memory fragment, else it is allocated in the continuous memory heap. If the memory is badly fragmented, i.e. so many small free unallocatable memory holes, the compiler will defragment the memory automatically. For the compiler to be able to do this, it keeps a list of pointers to all currently allocated CRAM variables. All memory allocation and deallocation issues are handled fully by the compiler.

6.2.4 Overloaded Operators

Most of the standard C++ operators have been overloaded for CRAM classes. These operators can be used with CRAM variables in exactly the same way as for standard C++ data types. Table 6.1 lists all operators that can be used with CRAM variables.

Operator (@)	Description	Examples (`cint a, b; cuint c; cbool h; int i`)
=	Assignment	`a = b;`

+
-
*
/
%
&
|
^
	Addition
	Subtraction
	Multiplication
	Division
	Modulus
	Bitwise AND
	Bitwise OR
	Bitwise XOR
		`a = a + b;`
		`c = c % 6;`
		`b = 7 & a;`

+= -= *= /= %= &= |= ^=	Add and assign Subtract and assign Multiply and assign Divide and assign Modulus and assign Bitwise AND and assign Bitwise OR and assign Bitwise XOR and assign	`a += b;` `c &= 0xFF;`

++
--
-
~
	Increment
	Decrement
	Negate
	Complement
		`a ++;`
		`b = --a;`
		`a = -b;`
		`c = ~a;`

<< >>	Shift left Shift right	`c = a << 5;` `b = a >> i;`
<<= >>=	Shift left and assign Shift right and assign	`c <<= 5;` `b >>= 2;`

<
>
==
!=
<=
>=
	Greater than
	Less than
	Equal to
	Not equal to
	Less than or equal to
	Greater than or equal to
		`cif (a >= 6)`
		`  a = b;`
		`cend`

||
&&
!
	Logical OR
	Logical AND
	Logical NOT
		`cif ((b<c) && (!b))`
		`  a = ++b;`
		`cend`

[]	Subscripting (*i*th element of `cvar`)	`i = a[24];` `b[3] = 6;`

Table 6.1 CRAM C++ Operators

6.2.5 Library Functions

Table 6.2 lists functions provided in the CRAM C++ library for CRAM data types. These include functions for bit-slicing, bit-setting, minimum and maximum searches, and PE shifts. Note that PE shift functions (`PE.shiftl()` and `PE.shiftr()`) are used to shift the elements of a CRAM variable, whereas standard shift operators (<< and >>) operate on CRAM variables in exactly the same way as they do on standard integer variables, i.e. the bits of the elements are shifted left or right (equivalent to multiplying or dividing by 2^n).

Function(s)	Description	Examples (`cuchar c; cint a, b; cbool h; int y`)
`set()`, `ones()`	Sets all bits of `cvar` to '1'	`c.set();` // c[PE] = 0xFF
`reset()`, `clear()`, `zero()`	Sets all bits of `cvar` to '0'	`c.clear();` // c[PE] = 0x00; equivalent to c = 0
`bit()`	Returns *i*th bit (`cbool`) of `cvar`	`a.bit(5) = h;` // set bit 5 of a to the value of h
`from()`	Returns bit-slice (`cvar`) of `cvar`	`c = a.from(4,11);` // c is set to bits 11:4 of a
`search_bitset()`	Returns the PE number with bit of `cbool` set to '1'	`y = search_bitset(h);` // y is set to a value such that h[y] = '1'
`ismax()` `ismin()`	PE has max(min)imum element of `cvar`	`h = ismax(a);` // h[PE]=1 if a[PE] is the maximum element of a
`max()` `min()`	Returns max(min)imum of two `cvar`'s	`a = min(a, b);` // a[PE] = (a[PE] < b[PE]) ? a[PE] : b[PE]
`maxindex()` `minindex()`	Returns index of PE with max(min)imum element	`y = maxindex(a);` // yth PE contains maximum element of a
`maxele()` `minele()`	Returns the max(min)imum element of `cvar`	`y = minele(a);` // does int x=minindex(a), then reads a[x]
`abs()`	Returns absolute value of `cvar`	`a = abs(a);` // a = (a<0) ? -a : a
`shiftr()` `shiftl()` `rotater()` `rotatel()`	Shift PE elements left/right (`rotatex` connects edge PEs (PE_0 and PE_{N-1}] when shifting)	`PE.shiftr(a, b, 4, 1);` // a[PE] = b[PE+4], fill edge PEs with '1' `PE.rotatel(a,b);` // a[PE] = b[PE-1], a[0] = a[*N*-1]

Table 6.2 CRAM Library Functions

6.2.6 Data-Parallel Conditional Statements

- Data-parallel conditional execution if supported by the `cif` statement proposed by Elliott [4]. This is similar to the `ifarray` construct used in STARAN [43]. The syntax of the cif statement is shown below:

Syntax

```
cif (cbool expression)
{
  ctstatements ...
[] celse {
  cfstatements ...
}]
cend
```

Example

```
cif (a > b)
  c += a;
celse
  c += b;
cend
```

cbool expression is any expression that returns a `cbool` object. The resulting `cbool` value has a 1 where the PE evaluated the expression to be true, and a 0 where the expression evaluates to false. This `cbool` value is then written to the PEs write enable (WE) registers to enable or disable the PEs from writing to their own memory. One key difference between `cif` and the C++ `if` constructs is that in the `cif`, all the PEs will execute the statements in both the `cif` (*ctstatements*) and `celse` (*cfstatements*) branches regardless of whether the *cbool expression* evaluates to false or true for a particular PE. The conditionality of the `cif` construct lies in the fact that the PE will only write the results of the executed statements to its memory if the condition evaluated to be true. Therefore, a useful statement inside a `cif/celse` will be one that assigns its result to a `cvar` (variable in CRAM memory), otherwise the statement will be executed unconditionally. Also, unlike the C++ `if`, `cif` cannot be used for say code execution speedup since all statements inside a `cif` or `celse` are always executed.

In order to support nested `cif`'s, a stack of `cbool` variables is maintained, with the top of the stack containing the current value of the write enable registers. This stack is automatically managed by the constructor and destructor of the `write_en` CRAM class. When a `cif` is executed, the result of the `cbool` expression is ANDed with the top of the stack (WE values), and the result written back to the WE registers and the top of the stack. The statements inside the `cif` are then executed. After this, if `celse` is encountered, and the `cbool` value next to the top of the stack is true, then the contents of the WE registers

and the top of the stack are inverted. After this, the statements inside the `celse` are executed. Notice that `celse` is optional. Every `cif`, or `cif/celse` pair, must be terminated with a `cend`. When `cend` is executed, the top of the stack is popped out and the write enable registers are restored to their values before the conditional execution.

`cif`, `celse` and `cend` are all defined as preprocessor macros. The curly brackets surrounding the statements inside a `cif` or `celse` are optional, but it is advisable to include them if there is more than one statement inside a conditional branch.

6.2.7 Operations with Scalar Constants

Operations that have a `cvar` and an integer value as operands (except << and >>) use the CRAM controller constant broadcast unit (Section 4.5.2). Instead of loading the integer into a temporary `cvar` in CRAM memory and doing the operation on two `cvar` variables, the integer is loaded into the controller write buffer and the operate-immediate instruction (such as `ADDI`, `SUBI`, etc.) is executed. Thus, the load-constant instruction (`LDK`), which has 2*bits microinstructions, is avoided, and no extra CRAM memory is required to store the integer constant. Also, the constants can be pre-loaded into the write buffer and operations performed directly with the constants in the buffer. The advantages of this are outlined in Section 4.5.2.

The CRAM C++ compiler performs some optimizations for operations with certain integer constants. This optimization is done by substituting the operation with an operation which results in fewer microinstructions. Table 6.3 shows all optimized operations. The compiler also optimizes multiplication and division by 2^n (n = 0, 1, 2,...) by replacing them with shift operations (<< and >>, respectively). However, to reduce the overhead of testing against all the possible constants, this optimization is only done for the constants 2, 4, 8, 16, 32 and 64. The programmer can perform the optimizations for the rest of the constants either at compile time (if they are known in advance) or at runtime (by testing against all such constants of interest).

Intended Operation	Optimized Operation	Microinstructions Saved(n=bits)
a = b + 0 ⇒ ADDI a, b, #0) a = b - 0 ⇒ SUBI a, b, #0	a = b ⇒ MOV a, b	$3n+1$
a = b | 0 ⇒ ORI a, b, #0 a = b ^ 0 ⇒ XORI a, b, #0 a = b & 1 ⇒ ANDI a, b, #1	a = b ⇒ MOV a, b	$2n$
a = b & 0 ⇒ ANDI a, b, #0	a = 0 ⇒ CLR a	$4n-1$
a = b | -1 ⇒ ORI a, b, #-1	a = 1 ⇒ SET a	$4n-1$
a = b ^ -1 ⇒ XORI a, b, #-1	a = $\overline{b}$ ⇒ NOT b	$2n$
a = b | 1 ⇒ ORI a, b, #1	a(0) = 1 ⇒ SET a(0) a(n-1:1) = b(n-1:1) ⇒ MOV a(n-1:1), b(n-1:1)	$2n+1$
a = b & 1 ⇒ ANDI a, b, #1	a(0) = b(0) ⇒ MOV a(0), b(0) a(n-1:1) = 0 ⇒ CLR a(n-1:1)	$4n-3$
a = b ^ 1 ⇒ ORI a, b, #1	a(0) = $\overline{b(0)}$ ⇒ NOT a(0), b(0) a(n-1:1) = b(n-1:1) ⇒ MOV a(n-1:1), b(n-1:1)	$2n$
a = b + 1 ⇒ ADDI a, b, #1 a = b - (-1) ⇒ SUBI a, b, #-1	a = b + 1 ⇒ INC a, b	$2n-1$
a = b - 1 ⇒ SUBI a, b, #1 a = b + (-1) ⇒ ADDI a, b, #-1	a = b -1 ⇒ DEC a, b	$2n-1$
a = 0 - b ⇒ SUBI a, #0, b	a = -b ⇒ NEG a, b	$2n-1$
a = -1 - b ⇒ SUBI a, #-1, b	a = -1 - b ⇒ †MINUS a, b	$2n-1$
a = 0 ⇒ MVI a, #0	a = 0 ⇒ CLR a	‡$n-1$
a = -1 ⇒ MVI a, #-1	a = -1 ⇒ SET a	‡$n-1$
a = 1 ⇒ MVI a, #1	a(0) = 1 ⇒ SET a(0) a(n-1:1) = 0 ⇒ CLR a(n-1:1)	‡$n-1$
a = b * 0 ⇒ MLTI a, b, #0	a = 0 ⇒ CLR a	‡$6n^2+3n-1$
a = b * 1 ⇒ MLTI a, b, #1	a = b ⇒ MOV a, b	‡$6n^2+n$
#a = b * 2^k ⇒ MLTI a, b, #2^k	a = b << k ⇒ SL a, b, #k	‡$6n^2-1$
a = b / 1 ⇒ DIVI a, b, #1	a = b ⇒ MOV a, b	‡$16n^2+36n+2$
#a = b / 2^k ⇒ DIVI a, b, #2^k	a = >> k ⇒ SR a, b, #k	‡$16n^2+35n+1$

†Special type of NEG with Y initially set to 1.
‡Plus the time for loading the constant from the host into the controller write buffer.
#k = 1, 2, 3, 4, 5, 6.

Table 6.3 Optimizations for Operations with Constants

6.2.8 Operand Extension

Operand extension is required if source operands have different number of bits, or if the number of bits of one operand is less than that of the destination operand. In CRAM, rather than physically extending the operands by creating temporary variables, each overloaded C++ operator has a CRAM instruction that is executed if a specific operand extension is required. This increases the speed of execution and also removes the need for using extra CRAM memory. For example, consider the execution sequence of an expression `c = a + b`, where `a` is 8-bit, `b` is 4-bit, and `c` is 16-bit. Firstly, the `ADD` instruction is executed for the first 4 bits. Then `ADD1` is executed for the next 4 bits to extend `b` to the size of `a`, and finally `ADD0` is executed to extend `a` and `b` to the size of `c`. In other words, `ADD1` only has to ripple carries, and `ADD0` is just sign extension. These instructions give a saving of $2n$ and $4n$ cycles, respectively, when executed for n bits instead of using the full `ADD` instruction. CRAM operand extension instructions have very few microinstructions and hence have been implemented for all the main operators.

6.2.9 Support Classes

Apart from the CRAM variable classes described in the previous sections, the CRAM C++ library also has classes for objects of the CRAM controller, CRAM PEs, and the host processor. These are briefly described below.

- **Controller Objects:** The CRAM controller class contains all information about the controller that is necessary to the CRAM C++ compiler. This includes the memory map of the units, the size (in bytes) of the instruction FIFO, the read/write buffers, and the control store, and information about controller registers and other units. This information is provided in form of controller sub-classes and instantiation of their objects in the controller class object. This approach is advantageous because it mimics the hardware architectural composition of the controller. Therefore specific functions can be implemented for the class of an individual controller unit or register, in order to model its behavior and keep track of its status during instruction execution. This makes it easier for the compiler to perform code execution optimization that is dependent on the architecture of the controller. Controller sub-classes include classes for the

instruction FIFO, read/write buffers, control store, command/status registers and their individual bits, and all user-accessible parameter registers. Typically, a class of a controller register includes members to store the register address and value. Member functions include those to implement register assignment (e.g. `WLEN = 7`), and register increment/decrement (e.g. `AR0++`). Common member functions for classes of the other units (FIFO, buffers, control store) are functions for reading and writing data. As mentioned earlier, these sub-classes and their functions allow the use of standard C++ constructs when assigning, incrementing, decrementing, reading and writing controller registers and units. Also, by having controller hardware units as objects in the C++ library enables the compiler to have an intelligent and updated information about the status of the relevant units. This is used in code optimization, for example by avoiding unnecessary updating of a register when the register already contains the value that it is supposed to be updated with.

- **Processing Element (PE) Objects:** The CRAM PE class contains information about, and functions to manipulate PE components. This includes the instantiations of CRAM registers (X, Y, M, W) and their groupings (such as XY, XWY), and functions for register assignment and PE shift operations. Like the controller classes, PE classes simplify the use of PE objects in the compiler or application source code. For example, to set all the PE X registers to zero, instead of issuing a native CRAM instruction, one may code it in normal C++ syntax using `PE.X = 0`. A number of PE assignment operators have been included to implement common PE register operations such as initialization (`PE.W = 1`), register-to-register copy (`PE.X = PE.Y`), and cvar-to-PE-register copy (`PE.W = a.bit(8)`).

- **Host Computer Objects:** Host computer classes include the software transposing buffers (Section 6.3) and CRAM instruction types. The instruction types are used for specific instruction characterization during execution. Examples include a one `cvar` source operand instruction (`one_addr_instr`), two `cvar` source operands instruction (`two_addr_instr`), and one `cvar` and one integer constant instruction (`addr_cont_instr`).

6.2.10 CRAM System Objects and Their Initialization

When an application compiled with the CRAM C++ library is run, a number of CRAM system objects are created and initialized. The compiler initializes these objects to the initial values of the real hardware units they represent.These objects include:

- **PE** - This represents the components of CRAM processing elements described in Section 6.2.9. Therefore, in the application code, all references to CRAM PE registers should be qualified with PE, e.g. `PE.X` to denote all PE X registers.

- **controller** - This object denotes the CRAM controller, and hence must qualify all reference to controller resources, e.g. `controller.WLEN`.

- **host** - Represents the host processor. The software transposing buffers (Section 6.3) are defined in the host object (`host.read_buffer` and `host.write_buffer`).

- **ALL_ZEROES_MASK** - This is of type `cbool` and is initialized to a value of all zeroes. It is located at CRAM memory row 0.

- **ALL_ONES_MASK** - This is of type `cbool` and is initialized to a value of all ones. It is located at CRAM memory row 1.

- **MASK_01** - This is of type `cbool` and is initialized to the value 0x01. It is located at CRAM memory row 2.

6.3 Corner-Turning: Host Access of CRAM Variables

Since CRAM operates on data in bit-serial format while the host computer is a bit-parallel system, data must be converted from bit-serial to bit-parallel and vice-versa when it is transferred between the two systems. This is referred to as either corner-turning, data transposing, or format conversion. Typically, data is transposed using multidimensional access memory (MDA) and a flip or shuffle network like the ones used in STARAN [46], [47], or the MIT image processor [14]. These methods require a lot of hardware, especially if the system is general-purpose and its operands are not fixed to a particular size (bits). For example, even for such an application-specific processor as the MIT pixel-parallel image-processing system [14], whose data path is designed for 8-bit gray scale pixels, sixteen 64Kb x 4 SRAMs, plus shuffler, address and other glue logic, are used to transpose the data between the host and the PE array [41]

Because of the high hardware cost and complexity of hardware corner-turning, data on a CRAM system is transposed in software. Traditionally, software data transposition is done on the host computer. This is slower than a hardware transposer, and may severely degrade performance especially if the number of elements to be transposed in a parallel variable is large. But with current increasing processor speeds, host-based data transposition offers a good alternative for low-cost system implementation. An example of a logic-in-memory system that uses software data transposing is the Terasys PIM workstation, which transposes data using its Sparc-2 host processor [13].

To reduce the overhead of transposing a large number of elements of a CRAM variable, a parallel array-based corner-turning approach that exploits the large degree of parallelism of the PE array and the 1-D inter-PE communication network has been developed. A 6-bit Buffer Address Increment register has been included in the CRAM controller to allow faster corner-turning of multi-byte data. The following sections describe these two corner-turning approaches.

6.3.1 Host-Based Data Transposition

Host-based data transposition is used if the number of elements to be accessed by the host is small. This is supported in the CRAM C++ library through the subscripting operator (`[]`). This operator has been overloaded for `cvar` in order to support constructs similar to those used by C++ in accessing elements of arrays for built-in data types. Any element of `cvar` can be addressed using the `[]` operator. For example, to initialize the 5th element of a `cint` variable `x` with an integer value of 6, the expression `x[5] = 6` is used. Similarly, the *i*th element of `x` can be assigned to an integer variable `y` using the expression `y = x[i]`. Figure 6.3 shows how integer values are written to `cvar` elements using this method. The data is first corner-turned through the host software write buffer (using shifting and masking), then written to the CRAM controller write buffer, and finally transferred to the CRAM memory using a `WRITE` instruction. If CRAM is a byte-wide memory, for the values to be transferred using the `WRITE` instruction, eight consecutive `cvar` elements must be loaded at the same time. If cvar elements are assigned in a non-consecutive manner (e.g. the expression `a[0] = 2` is followed by `a[3] = 7`), or if fewer than eight consecutive elements are assigned to, or if neither the first nor the last of the eight elements is on the $8n^{th}$ PE (n=0, 1, 2, ...), the values are written to the PEs memory as scalar constants. This is much slower because for an element to be initialized

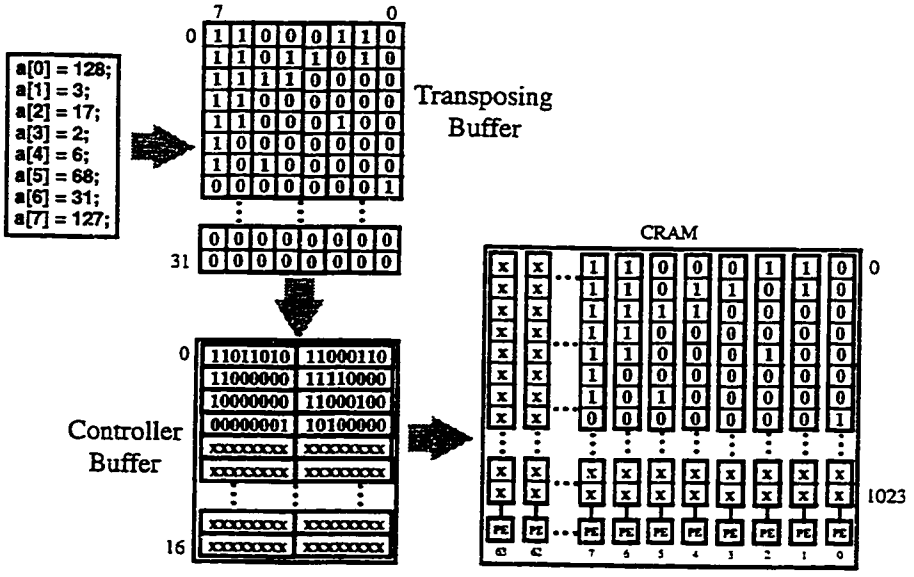

Figure 6.3 Host Corner-Turning

as a constant, the other PEs are first masked off by writing 0's to their write enable registers. After that, the constant is loaded into the controller write buffer, and finally loaded to all `cvar` elements using a `LDK` (load-constant) instruction. The WE registers are then restored to their previous values. If more than one element is to be loaded, this process is repeated for each element. Therefore to speed up the initialization of `cvar` elements, an effort must be made to initialize them consecutively. This can be achieved simply by initializing the elements using a `for` loop, with the initial value of the index starting at the $8n^{th}$ PE. To complement this, a `cvar load_elements()` function is provided which takes as arguments a pointer to the integer array, the size of the elements in the integer array, the `cvar` variable, and the number of `cvar` elements to be initialized. In both of these methods, the CRAM compiler automatically uses the transpose-WRITE method unless the number of `cvar` elements to be initialized is less than eight.

All elements of a `cvar` can be loaded with a uniform value by making the assignment without the `[]` operator. For example, for a `cint` variable `y`, all its elements can be loaded with 7 by using the expression `y = 7`. This uses a load-constant instruction with all WE registers enabled (by default, WE registers are always enabled). This is faster than the transpose-WRITE method since it requires only one load (to load the value into the controller write buffer) and one CRAM instruction (`LDK`).

Reading the value of a `cvar` element is almost the reverse of the write process described above. If CRAM is a byte-wide memory, eight `cvar` elements, including the one to be read, are read from CRAM to the controller read buffer using the `READ` instruction. These values are then transferred to the host read buffer, where the required value is transposed and returned to the assignment statement. Since eight elements were read, any subsequent assignment to the elements whose values are in the host read buffer will use these values without the need to read them from the CRAM chip. This is done until the elements in the read buffer are marked 'dirty', which happens when the `cvar` variable is used as a destination operand in an instruction.

6.3.2 Parallel Array-Based Data Transposition

To transpose the elements of an n-bit CRAM variable on the PE array itself, the data is loaded onto the CRAM array in the normal host (parallel) format, with n elements spread across n PEs. The data is then transposed in groups of n PEs by rotating the n x n bits. Typically, the minimum value of n is 8 since the byte is the smallest storage unit on most standard computers. Figure 6.4 illustrates array-based data transposition. For clarity, only

```
-----------------------------------------
-- TXUP: Transpose from top             --
--   AR0 - Addr of LSB of source data --
--   AR1 - Addr of LSB of tx'd data   --
-- TXDN: Transpose from down            --
--   AR0 - Addr of MSB of source data --
--   AR1 - Addr of MSB of tx'd data   --
-----------------------------------------
TXUP:
CREAD AR1;
LOAD RY, M;
CWRITE AR0, INCA, SETLOOP;
LOAD RY, Y;
CWRITE AR0, INCA, LOOPN, IF LPCNZ;
LOAD LX, X;
LOAD W, X, END;
--
TXDN:
CREAD AR1;
LOAD LX, M;
CWRITE AR0, DECA, SETLOOP;
LOAD LX, X;
CWRITE AR0, DECA, LOOPN, IF LPCNZ;
LOAD RY, Y;
LOAD W, Y, END;

/*-------------------------------------*/
/* cram_transpose()                    */
/*  CRAM-based parallel transposition  */
/*  of 8-bit integers;                 */
/*-------------------------------------*/
void cram_transpose (address& src,
                     address& dst)
{
  int i;

  // load the mask into regs X and W
  execute_instr (LDWX, MASK_01.addr, 0);

  // transpose from up: both addresses
  // increase after each iteration
  for (i=7; i>0; i--)
  {
    execute_instr (TXUP, dst, src, i);
    dst ++; src ++;
  }

  // load the mask from X regsiter to Y
  execute_instr (TXY, 0);

  // transpose from down
  for (i=7; i>0; i--)
  {
    execute_instr (TXDN, dst, src, i);
    dst --; src --;
  }
}
```

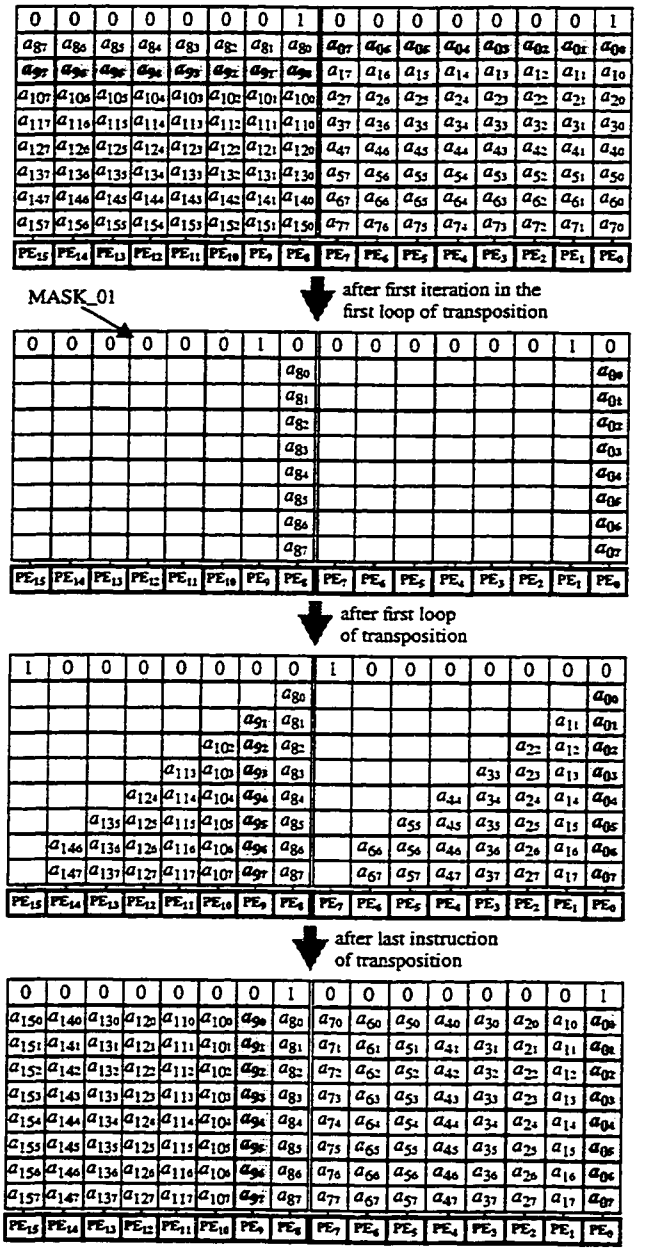

Figure 6.4 Parallel Array-Based Data Transposition

the first 16 elements of an 8-bit CRAM variable are shown. The exact steps are described in the actual transpose microroutines (`TXUP` and `TXDN`) and the C++ function (`cram_transpose()`).

If i is the number of the PE in the n-PE group, and j is the memory row of the PE, then data transposition is performed by moving the bits in the n x n window such that

$$a'_{ij} = a_{ji},\ 0 \le i, j < n \tag{6.1}$$

where a' is the transposed version of a. This is a symmetrical transformation and hence to transform data from CRAM format to host format, exactly the same operation is performed.

The `TXUP` and `TXDN` microroutines each executes $(5 + 2i)$ microinstructions, where i is the loop count variable in the transpose loops of the C++ `cram_transpose()` function. Each of these loops is executed $(n\text{-}1)$ times (n is the number of bits of the CRAM variable). Also note that before the transpose microroutine is executed, the WLEN register is first set to i. This makes the number of microinstructions executed for each transpose microroutine call equal to $(6 + 2i)$. Therefore, the total number of microinstructions executed in each of the transpose loops in `cram_transpose()` is equal to

$$M_{tx} = 6(n-1) + 2\sum_{i=1}^{n-1} i \tag{6.2}$$

For a CRAM system of cycle time T_c, the total time to transpose data, T_x, is equal to the time to execute the $2M_{tx}$ transposition microinstructions and six initialization microinstructions. The initialization instructions are used to update address extensions for `MASK_01`, `src` and `dst` operands, plus two microinstructions for `LDWX`, and one for `TXY`. Therefore, T_x is given by

$$T_x = \left\{2\left[6(n-1) + 2\sum_{i=1}^{n-1} i\right] + 6\right\}T_c \tag{6.3}$$

Simplifying Equation 6.3 gives

$$T_x = 2(n^2 + 5n - 3)T_c \tag{6.4}$$

Like all array-based PE operations, this time is constant for all numbers of elements equal

to or less than the number of PEs. Therefore, as the number of PE increases, the equivalent time to transpose an element decreases because of the parallelism in the PE array. For an 8-bit CRAM variable, the number of microinstructions executed in Equation 6.4 is equal to 202, and the transpose time for a 50 ns CRAM system is 10.1 μs. Figure 6.5 shows the performance of transposing a varying number of elements of an 8-bit CRAM variable using either the PE array of a 50 ns 64K-PE PCI CRAM system or a 133 MHz Pentium host PC. It is evident from the figure that host-corner turning is beneficial only if a very small number of elements (less than 8) is to be accessed.

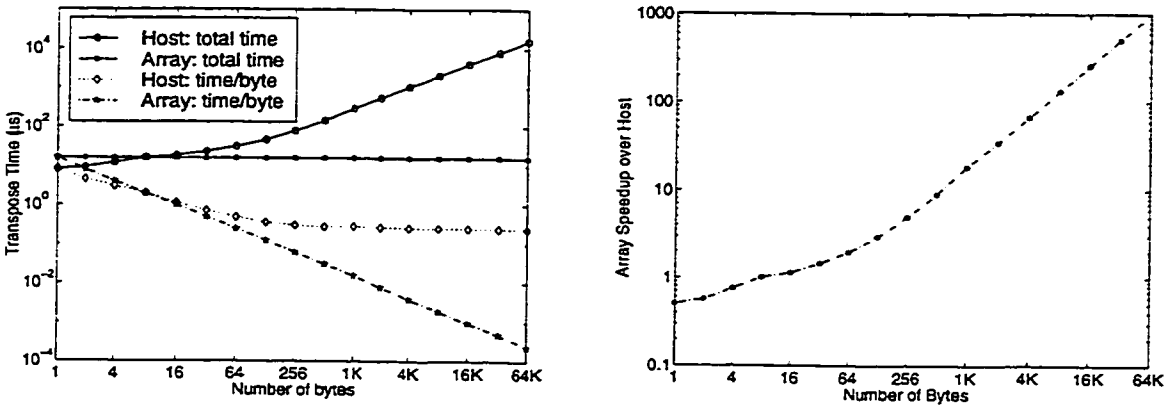

Figure 6.5 Array-Based vs. Host-Based Transposition

Transposing N-byte Variables

The transpose time for an n-bit variable has a complexity of $O(n^2)$. Therefore the transpose time for 16-bit and 32-bit variables would be almost 4 times and 16 times, respectively, the transpose time of 8-bit variables. More precisely, using Equation 6.4, the transpose time for a 16-bit and 32-bit variable is 3.3 times and 11.7 times, respectively, the transpose time of an 8-bit CRAM variable.

To reduce the time of transposing multi-byte variables, a procedure that has a complexity of $O(n)$ has been developed. This involves loading corresponding bytes of 8 consecutive elements in 8 consecutive memory rows, and then transposing them as 8-bit variables. Figure 6.6 illustrates this for a 16-bit variable. There is no extra overhead for this data arrangement because it is supported in the CRAM controller by simply loading the Buffer Address Increment register with N prior to issuing the `WRITE` instruction.

Using this approach, the transpose time for an N-byte CRAM variable is only N times that of a 1-byte (8-bit) CRAM variable.

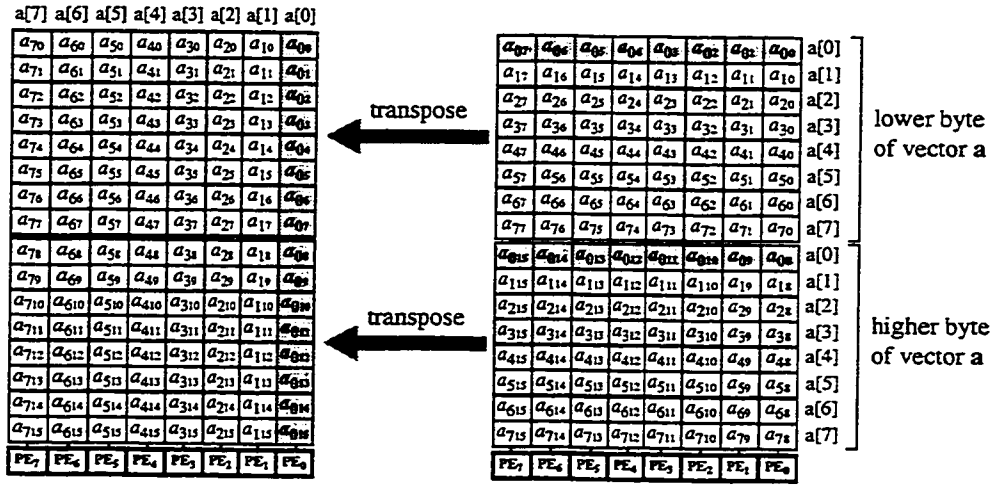

Figure 6.6 Transposing N-Byte Variables

6.3.3 Effect on Performance

The performance of data transposition is usually measured relative to the I/O overhead of transferring data between the bit-serial system and its host. This is so because in most cases the need to transpose data arises because data on the host has to be used by the bit-serial system and vice versa. An ideal data transposer adds nothing to the I/O overhead.

Figure 6.7 shows the data transpose time as a percentage of the total I/O overhead (data transposing + data transfer). Again, this is for a 50 ns CRAM system interfaced through a PCI bus to a 133 MHz Pentium PC. If the number of bytes to be transferred and transposed is small, both corner-turning approaches constitutes a very high percentage (more than 80%) of the total I/O overhead. But as the number of bytes increases, the percentage contribution of array-based transposition decreases, while that of host transposition remains almost constant. In other words, data transfer time increases in both cases, data transposition time when using the parallelism of the PE array remains constant, while the time of transposing data on the host also increases with the number of bytes. Typically, a CRAM system would have more than 4K PEs. This means that parallel CRAM variables would have 4K elements or more. For this case, data transposition using the PE array contributes less than 10% to the total I/O overhead. This number is as low as

0.5% for a 64K-PE PCI CRAM system. Note that the percentage contribution of data transposition is even smaller for slower buses because of the higher data transfer time. Also, when calculated as a percentage of the total execution time of an application (I/O overhead + code execution time), the effect of data transposition drops even further.

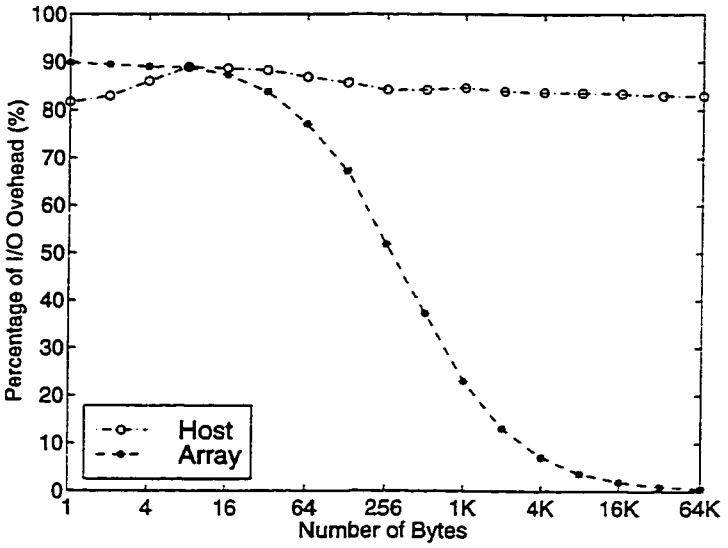

Figure 6.7 Data Transpose Time as Percentage of Total I/O Overhead

6.4 CRAM Assembler

6.4.1 Introduction

For applications that require explicit control of the raw hardware, applications can be written directly using the CRAM Controller Assembly Language (also known simply as the CRAM Assembly Language). This assembly code can either be mixed with the CRAM C++ code and tagged with the keywords `CASM` and `END CASM`, or it can written as separate code. If mixed with C++ code, the CRAM assembler (CASM) is used as a pre-processor before compiling the mixed code with a C++ compiler. In this case, CASM simply converts the assembly code into a C++ function to transfer the instruction to the controller instruction FIFO.

As with most assembly languages, it is very unlikely that a typical application programmer will ever use CRAM assembly code, except in situations where assembly

code offers considerable speed-up when compared to C++ code. Otherwise, CASM is a vital software development tool and has been used in, among other things, developing the CRAM C++ library and the CRAM C++ simulator, and generating input files for the controller VHDL simulations. The following section briefly describes the CRAM assembly language.

6.4.2 CRAM Assembly Language

A complete listing of the CRAM Assembly Language is found in Appendix C.1. This includes the syntax and usage examples of all the instructions in the language. The following sections give a very brief description of the instructions:

- **CRAM Memory Variable Instructions:** These are instructions that operate on CRAM memory variables (`cvar`). They include instructions to perform arithmetic, logical, relational, and search operations, as well as to shift, load, copy, set, and clear CRAM variables. Some of these instructions can have an immediate value as one of its source operands. A few of the logical operations are provided with what are called 'boolean instructions' in order to avoid the overhead of setting up the word-length (WLEN) register for operations involving boolean CRAM variables (see Appendix C.1 for details). A few examples of `cvar` instructions are shown below.

Examples

```
ADD #24, #16, #8;      // mem[24] = mem[16] + mem[8]
INC #24, #16;          // mem[24] = mem[16] + 1
MAX #27, #16;          // mem[17] = (is maximum element of mem[16])
MCLR #18;              // mem[18] = 0
MOV AR0, AR1;          // mem[AR0] = mem[AR1]
```

- **CRAM PE Register Instructions:** These are instructions that operate on PE registers. They include instructions to set or clear registers, and instructions to load PE registers from either other PE registers, or from `cbool` or `cboolref` variables.

Examples

```
CLR X Y;               // all X and Y PE registers are set to 0
LDPE X, !#16;          // load all X registers with inverse of mem[16]
TXPE X, Y;             // load X registers with contents of Y registers
```

- **CRAM Controller Instructions:** CRAM controller instructions include instructions to load, increment and decrement controller registers, as well as instructions for reading and writing data between CRAM and the controller. There is also a conditional read-bank instruction (RDBNK) used to scan a `cbool` or `cboolref` variable to check if there is a 1 at any PE position. This is done by first reading the byte of the variable corresponding to the first eight PEs. If this byte contains a 1 on any of its eight bits, the byte is saved in the DTR register and the RDBNK instruction terminates. Otherwise the bank address register (CBA) is incremented to point to the byte corresponding to the next eight PEs, and the cycle is repeated. This is repeated until either a 1 is found or all the bits of the variable have been scanned.The results of the scan (CBA and DTR) can be used by the host processor to compute the index of a PE that yielded a true value to a search or comparison.

Examples

```
LDAX0 #5;          // AX0 = 5
INCA AR1;          // AR1 = AR1 + 1
WRITE AR0;         // mem[AR0] = write_buffer[WIBA]
RDBNK AR0;         // loop (DTR = mem[AR0]) until mem[AR0] != 0
```

6.4.3 Using `cvar` Addresses and C++ Integer Variables

One special feature of the CRAM assembly language is that when it is mixed in CRAM C++ code, one can use the address of a C++ defined `cvar` variable where ever it is legal to use `#AR0`, `#AR1`, and `#AR2`. The address of a `cvar` is specified as *`var`*`.addr`, where *`var`* is the `cvar` variable. Any valid expression of `addr` can be used in the assembly code, e.g. *`var`*`.addr+2` for the address pointing to bit 2 (the third bit) of *`var`*. Similarly, any integer defined in the C++ code can be used where ever an immediate value or a `cvar` address is allowed in the assembly code. This allows easy optimization of part of the CRAM C++ code using CRAM assembly code.

Example

```
cint c; cuint b(8), a(8);                  // C++ defined cvar's
int x = 3;                                 // C++ defined integer variable
CASM
  LDWLEN #x                                // WLEN = x (i.e. WLEN = 3 or 4-bit word-length)
  AND c.addr+8, a.addr+4, b.addr+4;        // c[11:8] = a[7:4] & b[7:4]
END CASM
```

6.5 CRAM Microcode Assembler

The CRAM Microcode Assembler (CMASM), simply known as the CRAM Microassembler, is used as a high-level development tool when generating microinstruction routines. Instead of writing microinstructions in their native binary format, CRAM Microassembly Language is used. Typically, this is high-level language for low-level PE/RAM operations (e.g. `AND X, Y M X;` i.e. let each PE do an AND operation of the contents of registers Y M, and X, and write the result into X), and CRAM controller microinstruction execution control (e.g. `SETLOOP...LOOPN IF LPCNZ;` i.e. execute these microinstructions until the loop counter is zero).

Microinstruction routines for all operators and functions in the CRAM C++ library and the CRAM Assembly Language have already been generated (using CMASM) and comes together with the CRAM software library. Still there might be need to extend the C++ library or CASM, especially if a certain application-specific function occurs so frequently in application source codes as to warrant being implemented as a microroutine in the controller control store. In this case, the programmer can use CMASM to generate the microinstructions and append them to the existing ones. Otherwise, like CASM, the microassembler is mainly used by the CRAM designers in system development and analysis work. Details of the Microassembly Language can be found in Appendix C.2.

6.6 Grouping of Microroutines

As described in Chapter 3, the ALU result of a CRAM PE is derived from the truth table of its three input registers. Therefore, source operand registers are specified in terms of an 8-bit truth-table value, and this value is unique for each operation. This, together with the specification of destination operands, makes the number of microinstructions required for an operation much bigger when compared to standard processors. For example, with four possible destination registers (W, X, Y) and six possible source registers (X, Y, M, !X, !Y, !M), an instruction to move the contents (or inverse) of a register to another requires 34 unique 14-bit values of COP-TTOP combinations (6-bit COP and 8-bit TTOP).

To reduce the number of microinstructions in the control store, microroutines of all

instructions that differ by TTOP and/or COP in only one corresponding microinstruction have been grouped together. For each group, only one representative microroutine is loaded into the control store. The different values of TTOP and/or COP for the specific instructions in the group are coded in the unused operand fields of the instruction issued from the host. This has three advantages when compared to using bits in the instruction to point to specific microroutines or microinstruction subroutines. First, the size of the control store is reduced since we don't have to store the other microroutines or microinstruction subroutines. Second, the level of address decoding and multiplexing for the microprogram sequencer is reduced. This reduces both the area and critical path of the sequencer. Third, this approach further justifies the use of a uniform RISC-like instruction format for the macroinstructions because the wastage of bandwidth due to unused fields of the instruction word is now smaller. As mentioned earlier, a uniform instruction format results in simpler and faster instruction decoding and flow. Figure 6.8 illustrates microroutine grouping for instructions that set, clear, or transfer data between PE registers.

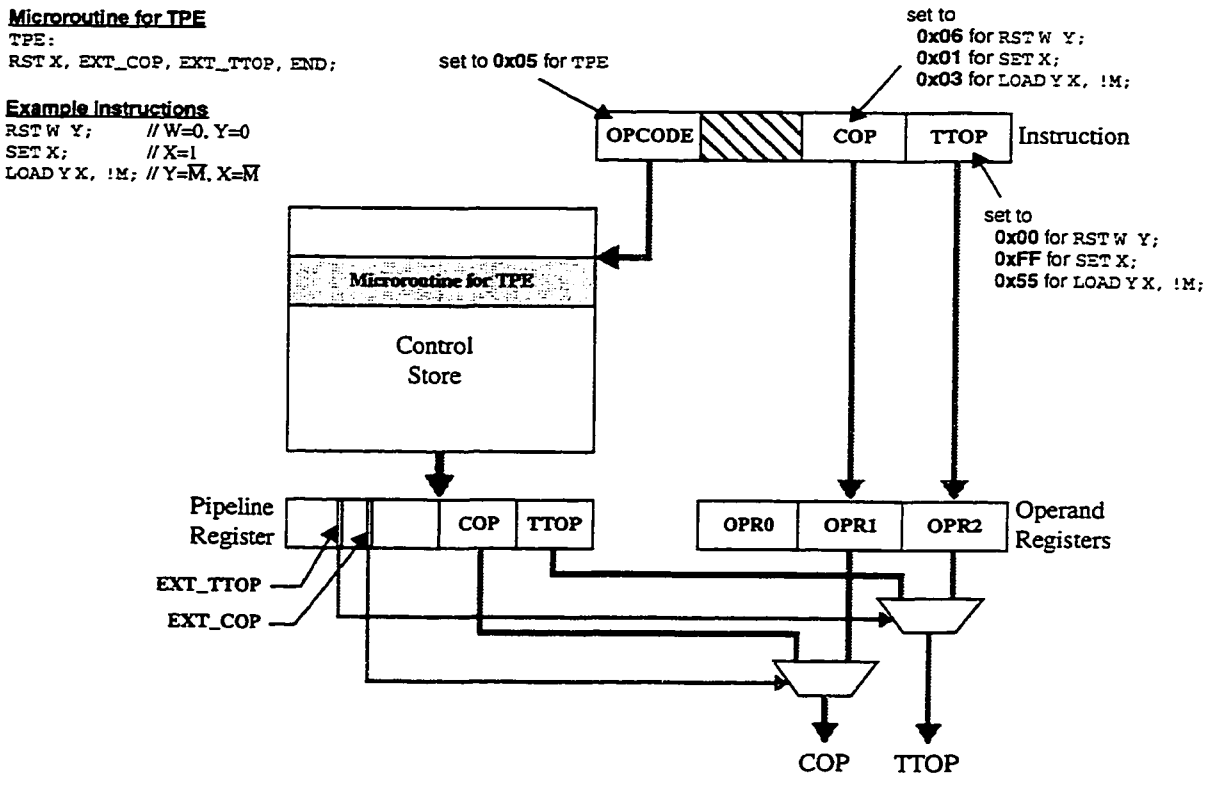

Figure 6.8 Grouping of Microroutines

Microroutines with more than one microinstruction are also grouped and executed in a similar way, with external TTOP and/or COP being selected only for the one specific microinstruction. Note that microroutine grouping is possible only if the instruction has less than three operands. Table 6.4 shows all instruction groups and those where microroutine grouping has been applied. Using this approach has reduced the number of microinstructions for basic operations from 462 to 197, i.e. a more than 57% reduction in the required size of the control store.

Instruction Group	No. of instructions	Total no. of microinstructions	†Grouped instructions	Microinstructions in control store
No operation	1	1	-	1
Controller-to-CRAM access	4	4	-	4
Controller instructions	9	9	1x9	1
PE reg-to-reg transfer, set, reset	48	48	1x48	1
Load PE-reg from memory	14	28	1x14	2
Load mem from PE-reg or constant, Extend result of logical operate	20	40	2x8, 1x10 1x9	6
Memory-to-memory move, Extend operand of logical operate	8	24	1x5 1x2	9
Add/sub, logical operate two `cvar`'s	7	39	-	39
Add/sub `cvar` & constant	3	18	-	18
Logical operate `cvar` & constant	3	12	1x3	4
Increment, decrement, negate, Extend operand of add/sub	16	81	2x3, 1x2 2x2	18
Extend constant of add/sub	6	27	3x2‡	15
Extend constant of logical operate	3	9	1x3	3
Extend result of arithmetic operations	12	47	2x6‡	10
Compare two `cvar`'s	2	8	1x2	4
Compare `cvar` and constant	3	9	1x3	3
Extend operand of compare	6	12	2x3	4
maximum/minimum search	2	18	1x2	9
PE Shifts	4	18	-	18
Data-parallel conditions	2	14	-	14
Data transpose	2	14	-	14
Total	**175**	**462**	-	**197**

†Example: 2x3 means 2 groups of 3 instructions each.
‡Partial or irregular grouping.

Table 6.4 Instruction Groups

6.7 CRAM C++ Simulator

6.7.1 Introduction

The CRAM C++ simulator is a tool that is used to simulate the behavior of a CRAM system. This can be used by both hardware and software designers. Hardware designers can use the simulator to explore different architectures of CRAM, its controller, and overall system design before committing to the actual hardware implementation. This would improve both the quality and performance of the design, while reducing the number of architectural errors that are only apparent after a design has been simulated or implemented. CRAM software tools designers can use the simulator to test and debug their tools, and applications developers can test their programs and extract more accurate timing information. The fact that the simulator offers a generic number of processing elements (PEs) makes it an ideal tool for performance analysis. While small prototype systems can easily be implemented at this stage of CRAM research work, the potential advantages of CRAM can only be demonstrated relevantly on systems with a large number of PEs. The simulator allows this to be easily realized.

For the simulator to be an accurate architecture exploration tool as well as yield more accurate program execution and timing behavior, the simulation model of the CRAM PEs and the CRAM controller is based on the hardware description of the actual implementations rather than on just their behavior. The disadvantage of this is that it makes the design of the simulator more complex and the simulator runs slower than that based purely on the behavior of the components. However, such a simulator gives out a behavior that is closer to that obtained by the design hardware description (VHDL/ Verilog) used in the actual design and implementation phases. This reduces the number of changes required when moving from an architecture obtained through simulation to that which is to be implemented using a hardware description language. It also yields more accurate timing information than that obtained in a behavioral simulator because in the latter a number of processes are usually bundled up in a single simplistic behavior.

The CRAM simulator is not an independent tool. Rather, it is designed to be a library that, when needed, can be linked with an application together with the CRAM C++

library. In this way, the execution of a program on the host, up to the point where the host transfers data onto the CRAM card or simulation model, is not a simulation but the real thing. This has some advantages. First, the complexity of the simulator is minimized since very few host features need be incorporated into the simulator. In particular, only the simulation of the CRAM card driver, and the timing information of the host system buses are required. All other aspects of program compilation and execution are already taken care of by the CRAM compiler. The second advantage is that even when you are using the simulator instead of an actual CRAM card, the timing information and program execution as related to the host system (except for transfers over the system bus) is exactly the same. This is especially important during the design and fine-tuning of the CRAM C++ library and other software tools.

6.7.2 Simulation Model

Figure 6.9 shows the CRAM C++ simulation model. When code executes on the host system, the simulator is invoked only when the host transfers data to the CRAM system. A single line in the CRAM compiler checks if the execution is on a simulator or on an actual CRAM card. If on a simulator, the host transfers data to the CRAM system by calling the simulator CRAM driver, otherwise the real driver is used. The simulation models of the CRAM system and the host environment are described in Section 6.7.3 to Section 6.7.5, and the different outputs of the simulator are described in Section 6.7.6 to Section 6.7.8.

6.7.3 Host Objects

The CRAM driver is the only component of the host system that is incorporated into the simulator. It is used as the link between the actual host system and the simulation model of the CRAM system. It simulates both the driver as well as different system buses (PCI, ISA, etc.). A link to the host system is through a function call to the driver (as would be done on a real system), while a link to the CRAM controller simulation model is through a function call that emulates the sequence of bus signals required to complete the data transfer on a specific system bus. Currently, only the ISA bus protocol is implemented in detail. The other buses are simulated by specifying the generic parameters

of the bus model such as bus width, bus speed, and whether the bus supports burst transfers.

Like all other models in the simulator, the host simulation model is implemented as a C++ class. Its class constructor is used, among other things, to model what happens when the CRAM driver is being initialized, such as the loading of microinstructions into the CRAM control store. Most timing parameters are also initialized at this time.

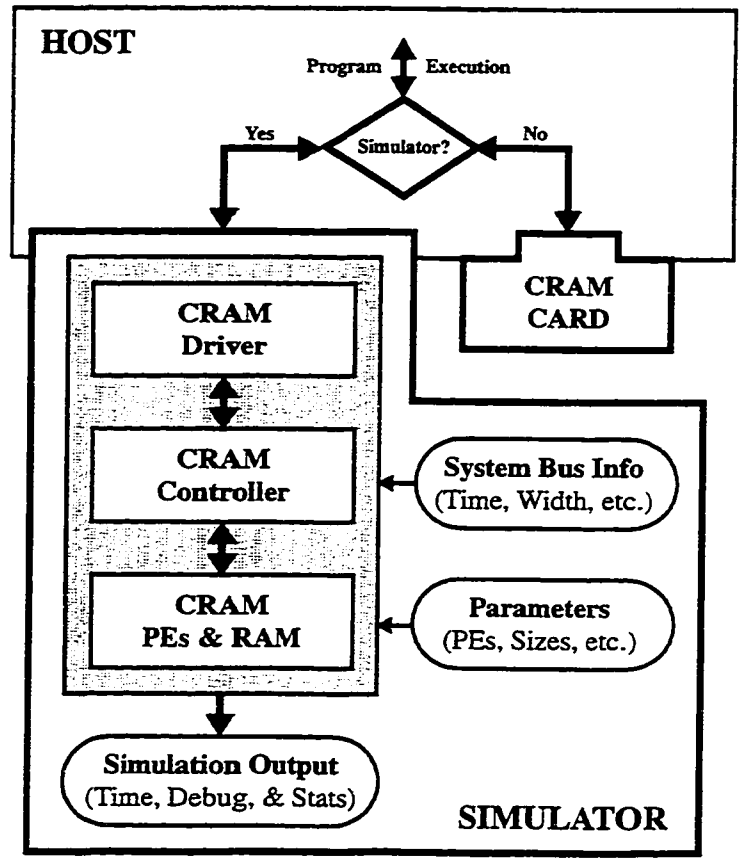

Figure 6.9 CRAM C++ Simulation Model

6.7.4 CRAM Controller

The simulation model of the CRAM controller is a hardware description (using C++) of the controller core components as well as the CRAM-host interface. Components are described as classes, with functions used to describe specific component features such as external and internal access and instruction execution. The flow of an instruction through the instruction execution unit (IXU) is described the same way as in the hardware, i.e.

instruction read from FIFO to IR, then executed in the microprogram sequencer using microinstructions, PE or RAM functionality decoded, address and data multiplexed with TTOP and COP and driven onto the CRAM bus, and finally the CRAM control signals (OPS, MCK, RW, etc.) asserted. All the other components are also modeled with such detailed hardware description. Such modeling is very easy to do if the real hardware was designed using a hardware description language such as VHDL or Verilog. This detailed modeling yields more accurate simulation and timing.

Like in the host simulation model, only the CRAM-ISA interface is described in detail. The other interfaces are simulated by specifying generic parameters of the bus model used in the host simulation model. Other generics in the controller model include the sizes (depths) of the instruction FIFO, the read/write buffers and the control store, as well as the size of the CRAM data and address buses.

6.7.5 CRAM PEs and RAM

The simulation model of the CRAM chips include the low-level hardware description of the PE architecture and the CRAM external interface. Other features of CRAM that relate to its computational nature (such as the bus-tie and shift operation) are also modeled based on how they connect and function in hardware. On the other hand, the memory component of CRAM is simply modeled as arrays of PE local memory since the hardware design of the memory is not a principal focus of the CRAM project. A generic number of PEs and memory bits/PE is used to simulate CRAMs of different sizes.

6.7.6 Timing Information

This measures the duration of time during specific stages of program execution from the host processor, via the system bus, to the CRAM controller, through the controller instruction execution unit, to the CRAM chip. This information can be used by both hardware and software designers. For example, the hardware designer can decide to adjust the size of the instruction FIFO or change the instruction pipeline scheme by studying the time that is spent filling an empty instruction queue or the time that the host processor has to wait before another CRAM instruction can be transferred to the CRAM controller.

CRAM software tools designers can evaluate the times that the host processor spends executing a specific CRAM C++ library routine and then decide whether to optimize or remove the routine. Applications designers can assess the time that an application takes to run on a CRAM system. Currently, the major timing parameters reported by the simulator include:

- **Host Execution Time:** The time that the host processor is executing some instructions while there are no instructions in the CRAM instruction queue, i.e. no parallel host/ CRAM activity.

- **Instruction Load Time:** The time taken by the host processor to load instructions into an empty CRAM instruction queue. This can be used, among other things, to determine the effect that the system bus transfer rate or the size of the instruction FIFO has on the overall program execution time.

- **Microinstruction Load Time:** The time that CRAM microinstructions are being loaded into the control store while the CRAM instruction queue is empty. This time is used to study the effect of the control store size as well as the make-up, size and selection of CRAM microroutines. Microinstruction load time does not include the time to load the initial microinstructions since this happens at boot-up during CRAM driver initialization. Therefore for a system with a control store big enough to require no microinstruction refilling, this parameter is always zero.

- **Data Access Time:** The time that the CRAM buffers are being accessed by the host processor while the CRAM instruction queue is empty. This time is used to study the effect of the size and configuration of the CRAM data buffers, as well as how an application data transfer loads and styles affect how fast it runs on a CRAM system.

- **Register Access Time:** The time that the CRAM memory-mapped registers are being accessed by the host processor while the CRAM instruction queue is empty.

- **Pipeline Fill Time:** The time to fill an empty CRAM instruction queue (FIFO⇒IR⇒Pipeline Reg) when an instruction begins executing from the FIFO. This is useful in studying, among other things, the effects of the FIFO size, the instruction

pipeline scheme and depth, and the inability of the host processor and the CRAM library to keep the CRAM instruction queue always full.

- **Microinstruction Execution Time:** The time taken to execute all CRAM microinstructions of routines invoked in an application. This is the major component of the total execution time of a program, and is calculated by multiplying the total number of microinstructions executed with the controller cycle time.

- **Total Execution Time:** This is the total time taken to run a program on a CRAM system. It is simply the sum of all the timing parameters described above.

All the timing parameters above accumulate as the program executes, but each can be individually reset at any point in the program or library routine in order to allow timing of specific features of a program or CRAM library routine. Note that most timing parameters are evaluated only when there is no concurrent execution of instructions in the CRAM controller because this is the only time that they contribute to the total execution time of a program. It must also be mentioned that there may be other uses of these parameters other than the ones mentioned here.

6.7.7 Debugging Information

The simulator outputs various information that can be used to debug an application, CRAM library routines, microinstruction routines, or even the CRAM software tools themselves. The main debug information includes:

- **CRAM Memory Image:** This gives the state, in binary format, of the local memory of each PE. The image is displayed with each PE and its local memory forming a single column of the memory array. The memory image can be displayed at any point in the application code or CRAM library routine, and the number of memory rows to be displayed can be specified. This, together with the fact that memory rows allocated to CRAM variables are grouped and separated from others, allows that a malfunctioning microroutine, library routine or part of an application code be debugged by simply looking at how they actually update the CRAM variables they are supposed to affect.

- **Microinstruction Execution:** As each microinstruction is being executed, its 32 bits, plus the resulting status of the CRAM address, data, and control buses are displayed. This may be used to debug microroutines or the simulation model of the controller.

- **Assembly Code Listing:** This lists the CRAM assembly code as the program executes. This feature is actually incorporated into the CRAM compiler and is activated using a #define statement. It may be used to debug CRAM C++ library routines.

- **Bus Transfers:** This displays the address and data of transfers between the host and the CRAM system.

6.7.8 Program Execution Statistics

Apart from timing and debugging information, a few statistics are reported by the simulator during program execution. These include:

- **Number of Specific Instructions:** Specific assembly code instructions may be counted. Typically, these are instructions used to update CRAM controller registers, such as LDWLEN and LDAXn. Such information may be used to optimize microroutines, the architecture of the controller, and the formats of both macroinstructions and microinstructions.

- **FIFO-Full Transfer Failure:** This is the number of times that the host data transfer to the CRAM instruction FIFO fails because the FIFO is full. In this case the host has to retry the transfer at a later time. This can affect the performance of the system especially if the host processor is working in a multitasking environment. The number of such failures depends on so many things, including the size of the FIFO, the depth of the instruction pipeline, and the cycle time of the controller in relation to the speed of both the host processor and the system bus.

- **Number of Microinstructions:** This is the number of CRAM microinstructions executed in a program. It is used, among other things, to calculate the main component of a program total execution time.

6.8 Summary

Writing application programs using low-level CRAM machine code requires detailed knowledge of the architecture of the CRAM chip and its controller. Therefore, to assist application programmers, software tools developers, and hardware or system designers, four high-level software tools that can be used to write and analyze CRAM applications and software tools, as well as analyze the different architectural features of a CRAM system have been developed. The CRAM C++ Compiler is a library of classes for CRAM parallel variables and controller objects that can be used to write CRAM programs using the standard C++ language. Such programs are compiled using a standard C++ compiler such as GNU C++ (gcc) or Borland Turbo C++. The CRAM Assembler can be used to write applications in CRAM assembly code, especially in cases where the C++ code does not yield the required speed or code size. The CRAM Microcode Assembler is a high-level tool for generating CRAM microinstructions. The CRAM C++ Simulator is used to simulate the behavior of a CRAM system. This tool outputs several timing and debug information, as well as other program execution statistics that can be used by both hardware and software designers.

This chapter has also described two other software issues: data transposition and microroutine grouping. Because of the high hardware cost and complexity of hardware corner-turning, data on CRAM is transposed in software. In this regard, a parallel array-based corner-turning approach that is several hundreds times faster than host-based transposition, and contributes less than 10% of the total I/O overhead for CRAM systems with more than 4K PEs has been developed. This removes the need for a hardware data transposer, thus reducing the area and complexity of a CRAM system. It also minimizes the effect of the host on CRAM performance, once again making it easier to implement CRAM systems on different platforms.

Finally, microroutines of similar instructions have been grouped together using unused fields of the instruction word. This has reduced the required size of the microprogram memory by more than 50%. This small size (less than 256 32-bit words) makes an on-chip control store feasible, even in standard ASIC technologies and FPGAs, and hence reduces the number of components on a CRAM system PCB.

Chapter 7

Applications and Performance Analysis

This chapter looks at CRAM applications and performance. The main objective is to show the suitability and performance of CRAM for practical applications of different characteristics. First, the characteristics of applications suitable for CRAM are highlighted, and the methodology used in performance evaluation is described. Section 7.2 then analyzes the performance of basic operations (addition, multiplication, etc.). After this, a few selected practical applications are described in terms of their definition, algorithm, implementation, and performance on both CRAM and conventional uniprocessor systems. The fields from which applications are selected include low-level image processing (Section 7.3), database applications (Section 7.4), and image and video compression (Section 7.5). The performance impact of the CRAM controller and other CRAM architectural features is analyzed in Section 7.6. The chapter concludes by comparing CRAM with other SIMD and logic-in-memory systems in terms of performance, hardware, and complexity.

7.1 Analysis Methodology and Parameters

Applications most suited for CRAM, like most massively parallel SIMD machines, are those that have fine grain parallelism and regular communication patterns. Such applications can be found in numerous fields including image processing, database operations, video and image compression, digital signal processing, computer-aided design, graphics, and numerical analysis [6], [7], [4]. The CRAM-implementation of a few of such applications is described in this chapter. The main objective is to highlight the use of CRAM for practical applications, as well as analyze the performance of the CRAM controller and the whole CRAM system based on practical applications. Three criteria have been used to choose the applications described here. First, different applications have been selected from different fields to show the general-purpose nature of CRAM. Second, these applications are of varying complexity and computation models in order to analyze the performance impact of different features of the CRAM system, such as inter-processor communication, global-OR usage, loading of scalars from the host, control and host overhead, degree of parallelism, and data types. Lastly, it is not within the scope of this thesis to find, develop, and test algorithms for all (or as many) applications that are suitable for CRAM. As mentioned earlier, the goal here is mainly to use some practical applications to analyze the CRAM hardware and software developed in this thesis, as well as to show that CRAM can indeed be used for practical applications. For this reason, the author has developed CRAM algorithms for only a few applications, and these are the ones described in this chapter. Appendix D lists the CRAM and uniprocessor C++ code for these applications. Other applications not described here, but whose CRAM algorithms have been developed by other people, include fault simulation, data mining, satisfiabilty problem, and FIR filters [4], as well as discrete cosine transform, adaptable and scalable vector quantization, and other MPEG-2 algorithms [65], [66], [67], [68].

Because of the small sizes and limited number of the prototypes implemented so far, the performance analysis work is carried out using the CRAM C++ Simulator. This not only allows the use of bigger CRAMs but also makes it possible to study how the performance of applications change as the size and other parameters of the CRAM system are varied. As described in Section 6.7, the CRAM C++ Simulator is designed to be a very

close approximation of the real CRAM system. Therefore the accuracy of the results obtained in these simulations is very close to the actual numbers. This has been verified by running some of the smaller algorithms, especially arithmetic, logic, and search operations, on the 64-PE ISA CRAM system prototype. The CRAM simulation model used for performance analysis is a 50 ns, 32 MBytes, 64K PE, PCI-based CRAM card on a 133 MHz Pentium PC. This is compared to the performance of running the applications on two uniprocessor systems: a PC driven by a 133 MHz Intel Pentium processor, with 32 MBytes of RAM (the same system serving as the host for the CRAM system), and a SUN Sparc Ultra workstation driven by a 167 MHz Sparc processor, with 64 MBytes of RAM.

7.2 Basic Arithmetic, Logic and Memory Operations

A basic way of assessing the performance of CRAM is to look at the execution of basic computational and memory-access operations. Table 7.1 shows the number of CRAM microinstructions required to perform these operations. It also shows the number of 32-bit and 8-bit operations performed per second by a 20 MHz, 64K PE CRAM.

Operation	Notation	Microinstructions per n-bit operation	8-bit Giga-Operations per second	32-bit Giga-Operations per second
Addition	C = A + B	$6n + 1$	26.75	6.79
Multiplication	C = A * B	$8n^2 + 16n + 2$	2.04	0.151
Division	C = A / B	$18n^2 + 53n + 2$	0.83	0.065
Logical AND	C = A & B	$5n$	32.77	8.192
Memory Clear	C = 0	$n + 1$	145.64	39.72
Load Immediate	C = #k	$2n$	81.92	20.48
Copy	C = A	$3n$	54.61	13.65

Table 7.1 CRAM Basic Operations

Table 7.2 compares the execution times of these operations on the CRAM and the two uniprocessor systems. This is for 64K *long* (32-bit) and *char* (8-bit) integer arrays. On CRAM, the execution time is proportional to the number of microinstructions in the

operations. But as shown in Table 7.2, these differences between operations do not translate into proportional differences in their execution times on the uniprocessor systems. This is reason why instructions with a smaller number of CRAM microinstructions yield higher speedups.

Application	Workstation Execution time (ms)		PC Execution time (ms)		CRAM			
					Execution time (μs)		Speedup over PC	
	32-bit	8-bit	32-bit	8-bit	32-bit	8-bit	32-bit	8-bit
Addition	10.76	7.25	12.07	6.43	9.65	2.45	1251	2624
Multiplication	15.19	14.6	15.57	10.71	435.3	32.1	36	334
Division	27.28	27.21	33.01	17.03	1006.5	78.9	33	216
Logical AND	11.74	7.34	12.07	6.43	8.0	2.0	1509	3215
Memory Clear	3.89	2.99	4.62	3.86	1.65	0.45	2800	8578
Load Immediate	3.89	2.99	4.62	3.86	3.2	0.8	1444	4825
Copy	2.8	2.73	8.85	5.12	4.8	1.2	1844	4267

Table 7.2 Performance of Basic Operations

The performance of SIMD machines with bit-serial processors stems from the massive number of processors rather than from the computational power of the individual processors. It is therefore important to assess how the performance of CRAM varies with the number of PEs. Figure 7.1 shows this variation for the basic operations. There are two things to note from the figure. First, for larger numbers of PEs, the speedup is proportional to the number of PEs. But for smaller numbers of PEs, this variation is not linear because the execution time of the operation itself becomes much smaller when compared to the

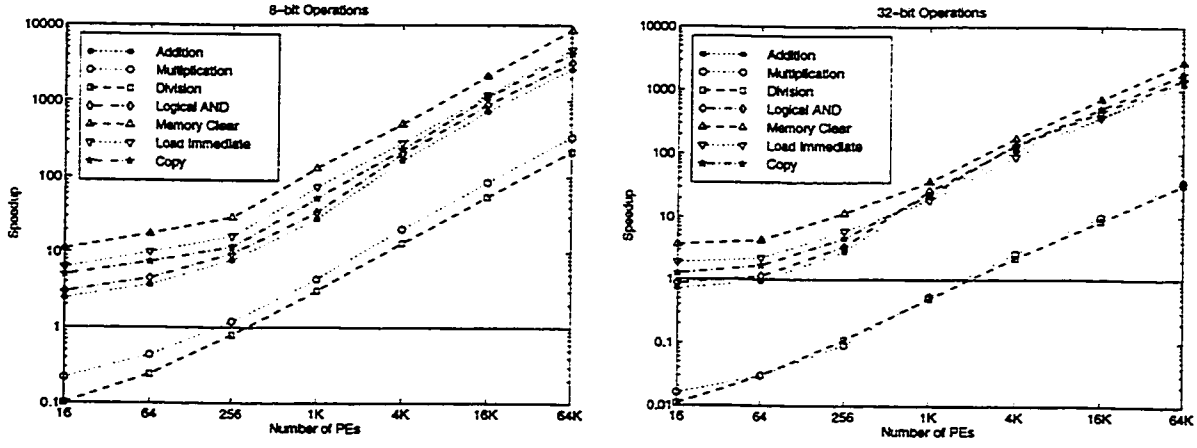

Figure 7.1 Effect of Degree of Parallelism on Basic Operations

other overheads of executing the operation. The second point is that for operations with bigger (bit-size) operands or higher degrees of complexity, not only does the speedup decrease, but the threshold at which CRAM performs better than the uniprocessor occurs at a larger number of PEs.

As shown earlier, the execution time of most CRAM operations is proportional to the bit size of their operands. On uniprocessor systems that are based on fixed-size (byte, word, double-word, and quad-word) operands, the execution time of most basic operations do not show substantial variation with operand size. For example, the execution time of 32-bit addition on the PC is less than twice the execution time of 8-bit addition, even though the operands are 4 times bigger. Because of these small differences in performance, it is common for programmers to use operand sizes that are bigger and more common (such as integers) than the required precision. But as shown in Table 7.1, the difference in execution times between CRAM operations with different operand sizes is very substantial. It is therefore more important to choose the right operand precision when writing programs for CRAM because there is a significant difference in the speedups over the PC if the operation is executed with different operand sizes. This is illustrated in Figure 7.2 for the basic operations on 64K arrays. Note that on the PC, 2-bit, 4-bit, and 8-bit operations all have the same execution times, thus yielding even higher CRAM speedups for operations on bit sizes less than 8.

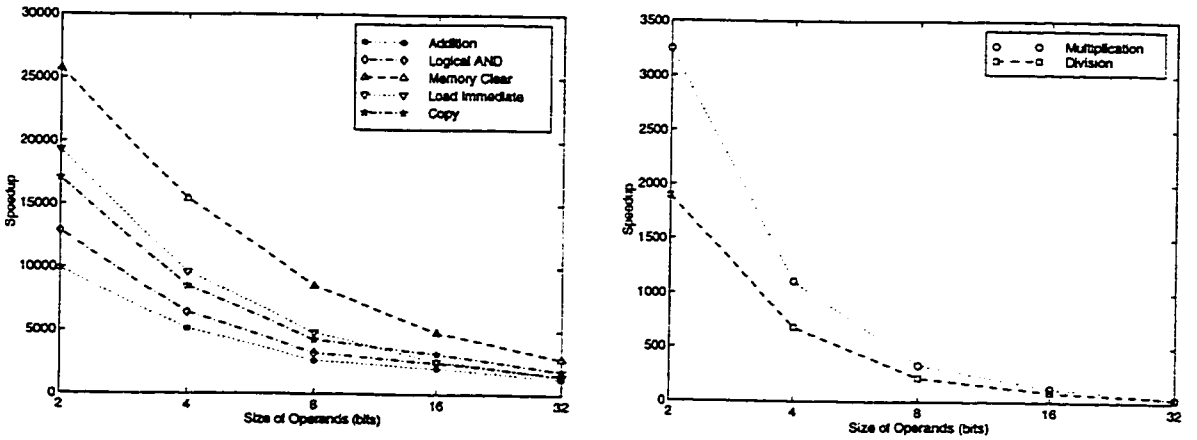

Figure 7.2 Effect of Operand Size on Basic Operations

7.3 Low-Level Image Processing

Low-level image processing involves sensing and pre-processing of an image before it is subjected to medium-level and high-level image processing operations such as segmentation, description, recognition and interpretation. Typical low-level operations include image enhancement, filtering to remove noise or improve image fidelity, edge detection, and threshold-based segmentation.

Low-level image processing is particularly suited for CRAM for two main reasons. First, most of the operations are massively parallel. The same computational operation is applied to numerous (usually more than 1000) pixel data. Therefore, each CRAM PE can be assigned to one or a few pixels and the computation done in parallel. Second, most low-level image processing operations are computed over small pixel neighborhoods [69]. These operations are either pointwise, i.e. the result at a pixel is a function of the initial data at that pixel alone, or they are local to a 3 x 3 or smaller pixel neighborhood. This means that data can be transferred from nearby PEs. This is important because of the limited inter-processor connectivity on CRAM.

The following sections describe the parallel algorithms and CRAM implementation of some low-level image processing operations. These are based on sequential algorithms adapted from those described in the literature such as [69], [70], and [71]. The simulations are based on a 256 x 256 grey-scale (8-bit) image and a 64K-PE CRAM, i.e. one pixel per PE. The timing results are for the case in which CRAM is used as the main memory. Otherwise, if the image is to be loaded onto CRAM from the host computer, processed, and then read back to the host, data I/O overhead has to be added to the total timing. This includes the time to write and read the data, and the time to transpose it. For a 256 x 256 8-bit image (64Kbytes), these times are 1.64 ms (2 x 0.82ms), and 31.58 μs (2 x 15.79 μs) respectively. It must be noted that usually several low-level image processing operations are applied to the image before it is used (displayed or further processed) elsewhere [11], [69]. For example, image enhancement operations such as contrast enhancement are usually followed by filtering. In this case, the I/O overhead is shared by several operations.

7.3.1 Brightness Adjustment

Brightness adjustment is used to brighten up dark images or darken images that are too bright. This is done by adding (or subtracting) a value to (from) each pixel in order to shift the pixel intensity by the specified number of grey levels. Algorithm 7.1 shows the brightness adjustment algorithm for both a uniprocessor and CRAM.

```
/* on uniprocessor */
for pixel=0 to N-1 pixels
   add/subtract a value to/from the pixel;
   if pixel is greater than grey-level limit
      saturate the pixel;
end for
```

```
/* on CRAM */
in parallel
   add/subtract a value to/from the pixel;
   if pixel is greater than grey-level limit
      saturate the pixel;
end in parallel
```

Algorithm 7.1 Brightness Adjustment

To adjust the brightness of a 256 x 256 grey-scale image took an average time of 9.12 μs on CRAM, 2.72 ms on the workstation, and 7.44 ms on the PC. This represents a CRAM speedup of 298 and 816 over the workstation and the PC, respectively.

7.3.2 Spatial Average Low-Pass Filtering

Apart from poor contrast, images may contain random pixels that have values higher or lower than what they should be. One way to reduce this type of noise is to replace the value of each pixel with the weighted average of its neighborhood pixels, i.e.

$$g'(x,y) = \sum_{i=-m}^{m} \sum_{j=-n}^{n} w(i,j) \cdot g(x-i,y-j) \tag{7.1}$$

where g is the noisy image, g' is the filtered image, and $w(i,j)$ are normalized filter weights in an m x n pixel neighborhood. A common and simplest class of spatial averaging filters has equal weights and is operated in a 3 x 3 pixel neighborhood, giving

$$g'(x,y) = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} g(x-i,y-j) \tag{7.2}$$

Its algorithms are described in Algorithm 7.2.

```
/* on uniprocessor */
for pixel=0 to N-1 pixels
  find average of 3 x 3 neighborhood pixels;
  replace pixel value with this average;
end for
```

```
/* on CRAM */
in parallel
  find average of 3 x 3 neighborhood pixels;
  replace pixel value with this average;
end in parallel
```

Algorithm 7.2 Weighted Average of Neighborhood Pixels

Most SIMD pixel processing machines have inter-processor connectivity that makes data access from the nearest neighbors implicit in the architecture. Examples of such hardware inter-PE communications include the 2-D (North, East, West, and South neighbors) connectivity used in MIT Pixel Processor [14], the square or "X" (nearest 8 neighbors) connectivity of BLITZEN [72], and other more complicated networks such as the hypercube and the global router used in the Connection Machine [73] and MasPar MP-1 [29]. For a 3 x 3 neighborhood, data can be accessed from a neighbor PE in at most two cycles for the 2-D inter-PE connectivity, and one cycle for the square connectivity. Other networks can transfer data between any two PEs in a few clock cycles (e.g. 16 clock cycles on the global router of MasPar MP-1). On CRAM, only a linear (1-D left/right) inter-PE communication is implemented in order to reduce the size of the PEs. This, however, results in large communication penalty when data has to be accessed from neighboring PEs. Algorithm 7.3 and Figure 7.3 illustrate the method that has been used to implement the access and computation of pixel data in 3 x 3 or smaller neighborhoods. For clarity, the figure shows the case for an 8 x 8 image.

```
do twice, first for shift = shift left, and then for shift = shift right
  shift image pixels 1 position into temp variable;
  use temp to access pixel p(x-1, y) or p(x+1, y);
  shift temp W-2 positions into temp;
  use temp to access pixel p(x+1, y-1) or p(x-1, y+1);
  shift temp 1 position into temp variable;
  use temp to access pixel p(x, y-1) or p(x, y+1);
  shift temp 1 position into temp;
  use temp to access pixel p(x-1, y-1) or p(x+1, y+1);
end do
```

Algorithm 7.3 Accessing Pixels in a 3 x 3 Neighborhood on CRAM

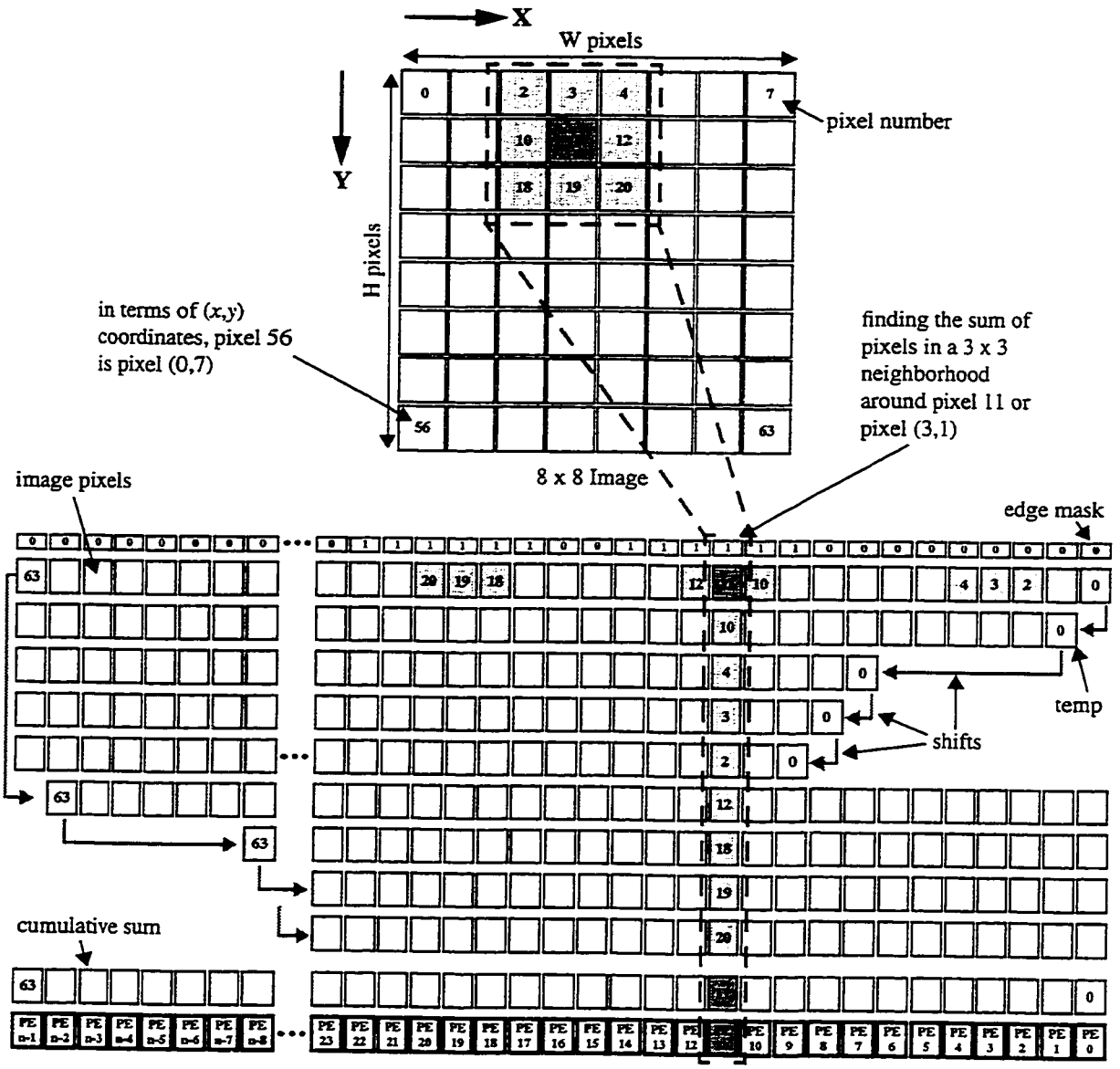

Figure 7.3 Computing in a 3 x 3 Pixel Neighborhood

The image is laid out across the PEs, one row at a time. For an image of width W pixels, the furthest pixel of the neighborhood will be W+1 PEs away. Instead of shifting the pixels independently, cumulative shifting is used. Therefore, to access a pixel to the left and the three pixels above, a total of W+1 shifts are required. The same is true for the other four pixels (one to the right, and three below). On the current CRAM system, shifting a b-bit variable by p PE positions takes $b(4 + p)$ microinstructions. The case where ($p = 1$) has been optimized and uses only $4b$ microinstructions. Therefore the total time for the shift operations in Algorithm 7.3 is given by

$$T_{shift} = 2bT_c(14 + W) \tag{7.3}$$

where T_c is the CRAM cycle time. The edge mask in Figure 7.3 is used to disable the processing of border pixels. This mask can either be pre-loaded like the other masks, or its loading time (about 0.1 ms for a 64K image) can be added to the total execution time.

Despite the inter-PE communication bottleneck, the weighted average filtering of a 256 x 256 8-bit image exhibits a CRAM speedup of 136 over the PC, and 82 over the workstation. The respective execution times are 0.39 ms, 53.13 ms, and 31.78 ms. Of particular interest is that for CRAM the total shift time (T_{shift}) for the average filtering is equal to 0.216 ms and represents about 55% of the total execution time and 56% of the total CRAM microinstructions executed. This means that a CRAM with at least 2-D inter-PE connectivity would have about twice the speedup reported here. Also note that because the shift overhead represents only about half of the total execution time, there is no advantage in re-ordering the image to reduce W since this would require that the averaging routine be executed more than once, and/or that the borders of the partitioned image be further processed. In fact, the arrangement of the image pixels shown in Figure 7.3 is advantageous because it retains the format that is used for the pointwise operations.

7.3.3 Edge Detection and Enhancement

Edge detection and enhancement are important operations in image segmentation and object recognition [69], [71]. The Prewitt and the Sobel are two of the most commonly used 3 x 3 neighborhood edge-detection operators. These compute both the magnitude and direction of the edge. However, if only the presence of the edge and not its direction is of interest, the direction-invariant Laplacian edge detector (Figure 7.4) is usually used.

0	1	0
1	-4	1
0	1	0

Figure 7.4 The Laplacian Edge Detection Operator

Algorithm 7.4 [70] enhances the edges in an image by subtracting the Laplacian of a

pixel from the pixel itself. This is a high-pass filter implemented by simply subtracting the low-pass filter output from its input [71]. Apart from extracting edges, this algorithm is also used for sharpening images that are slightly out of focus or taken with a jittery camera. On CRAM, the pixels in the 5-pixel neighborhood are accessed using the method described in Algorithm 7.3 and Figure 7.3, with the slight variation that the last two 1-position shifts are eliminated, and the top and bottom pixels are accessed with W-1 shifts. The total execution times for a 256 x 256 grey-scale image are 0.25 ms on CRAM, 31.4 ms on the PC, and 14.83 ms on the workstation. These translate into speedups of 126 and 59. The pixel-access shifts represents 85% of the total execution time, and 87% of the total CRAM microinstructions executed.

```
/* on uniprocessor */
for pixel=0 to N-1 pixels
  find the Laplacian of the pixel;
  find difference between pixel and its Laplacian;
  find absolute value of the difference;
  if absolute difference is greater than 255
    saturate it to 255;
  replace pixel value with this difference;
end for
```

```
/* on CRAM */
in parallel
  find the Laplacian of the pixel;
  find difference between pixel and its Laplacian;
  find absolute value of the difference;
  if absolute difference is greater than 255
    saturate it to 255;
  replace pixel value with this difference;
end in parallel
```

Algorithm 7.4 Edge Enhancement

7.3.4 Segmentation

Segmentation is used to partition an image into a useful set of objects and the background. Each image pixel is classified into a class of pixels based on some property of the pixel. Two examples of segmentation based on thresholding are conversion to binary image and multiple thresholding.

(a) Conversion to Binary Image

One of the simplest approach to segment a grey-scale image g is to threshold it at a value T to produce a binary image g'. This threshold operation is defined as

$$g' = \begin{cases} 1, g \geq T \\ 0, otherwise \end{cases} \tag{7.4}$$

A suitable value of T can be obtained from the histogram of the image. Binary images are used in object recognition or image compression operations such as motion estimation [68], [74]. To convert a 256 x 256 8-bit image to a binary image (Algorithm 7.5) took 9.25 μs on CRAM, 5.23 ms on the PC, and 2.31 ms on the workstation. This represents speedups of 565 and 250.

```
/* on uniprocessor */
for pixel=0 to N-1 pixels
  if pixel is greater than or equal to threshold value
    set its value to 1;
  else
    set its value to 0;
end for
```

```
/* on CRAM */
in parallel
  if pixel is greater than or equal to threshold value
    set its value to 1;
  else
    set its value to 0;
end in parallel
```

Algorithm 7.5 Conversion to Binary Image

(b) Multiple Thresholding

If there are M objects with distinct grey-levels, the image may be segmented by multiple thresholds, i.e.

$$g' = \begin{cases} M, & g \geq T_{M-1} \\ M-1, & T_{M-2} \leq g < T_{M-1} \\ \ldots, & \ldots \\ 1, & T_0 \leq g < T_1 \\ 0, & otherwise \end{cases} \quad (7.5)$$

where T_k is the threshold for object k. The desired objects can then be thresholded out of image g'. Multiple thresholding a 256 x 256 grey-scale image with 16 objects exhibits a CRAM speedup of 513 on the PC, and 209 on the workstation. The execution times are 0.27 ms on CRAM, 138.41 ms on the PC, and 56.54 ms on the workstation.

7.4 Database Applications

Many database operations are suited for parallel processing [6]. However, because of the limited inter-PE communication and the slow speed of the individual bit-serial PEs, the type of operations particularly suitable for CRAM are those that are pointwise and have an

SIMD-implementation complexity of $O(1)$. Examples of such operations are basic searches and multi-field record matching. In this section, algorithms are presented along with the simulation results of running these operations on a database of 64K (65536) records of 32-bit (long integer) data. The algorithm for the Least Means Squared Match used for searching multi-field database records is adapted from the one described by Elliott [4].

7.4.1 Basic Searches

Examples of basic searches include:

- *Equivalence search* - search for records equal to or not equal to the search key.
- *Extreme search* - search for maximum or minimum record.
- *Threshold search* - search for records greater than or less than the search key.
- *Between-limits search* - search for records between two search keys.

These four search operations were simulated on a randomly-generated database of 64K 32-bit records and a 64K-PE CRAM. In both the uniprocessor and CRAM algorithms (Algorithm 7.6), all records matching the search criterion are replaced by a new value. Table 7.3 shows the simulation results.

```
/* on uniprocessor */
for record=0 to N-1 records
   if record matches search criterion
      replace record with new value;
end for
```

```
/* on CRAM */
in parallel
   if record matches search criterion
      replace record with new value;
end in parallel
```

Algorithm 7.6 Basic Searches

Basic Search Operation	Execution Time (ms)			CRAM Speedup Over	
	PC	Workstation	CRAM	PC	Workstation
Equal-to	5.96	2.94	0.0138	432	213
Maximum	6.85	3.38	0.0150	457	225
Greater-than	7.06	3.54	0.0128	552	277
Between-limits	8.73	4.67	0.0196	445	238

Table 7.3 Performance of Basic Searches

7.4.2 Least Mean Squared (LMS) Match

LMS is used in searches where the data records (and the search key) contain multiple fields or criteria. While an index may be used for each field of the records, the nearest match for the combination of criteria may not be the best match for any single criteria. One way to combine all the criteria in a record, in order to find the best match with a search key is to find the sum of the squared differences (SSD) between the records fields and the corresponding fields in the search key. The best match is the record(s) for which SSD is a minimum. Given a database of N records, each with K fields, $R^0 = \{r_0^0, r_1^0, \ldots, r_{K-1}^0\}$, $R^1 = \{r_0^1, r_1^1, \ldots, r_{K-1}^1\}$,..., $R^{N-1} = \{r_0^{N-1}, r_1^{N-1}, \ldots, r_{K-1}^{N-1}\}$, and an K-field search key $S = \{s_0, s_1, \ldots, s_{K-1}\}$, the SSD is calculated as

$$SSD(S,R^i) = \sum_{j=0}^{K-1} (s_j - r_j^i)^2, \quad 0 \le i < N \tag{7.6}$$

Again, the LMS algorithm (Algorithm 7.7) replaces all matching records with a new value. Execution times for a database of 64K records, each with four 16-bit data, are 0.97 ms for CRAM, 81.7 ms for the PC, and 68.96 ms for the workstation. This represents a speedup of 84 over the PC, and 71 over the workstation.

```
/* on uniprocessor */
for record=0 to N-1 records
  reset difference to zero;
  for record field=0 to K-1 fields
    accumulate difference between record field
    and corresponding field of the search key;
  end for
  if difference is the minimum so far
    store the difference and the record number;
  end if
end for
for all records with minimum difference
  update record with new value;
end for
```

```
/* on CRAM */
// records (all fields) stored one per PE
in parallel
  reset difference to zero;
end in parallel
for record field=0 to K-1 fields
  in parallel
    accumulate difference between record field
    and corresponding field of the search key;
  end in parallel
end for
in parallel
  if difference is the minimum
    update record with new value
  end if
end in parallel
```

Algorithm 7.7 Least Means Squared (LMS) Match

7.5 Image and Video Compression

Image and video compression is a process of yielding a compact digital representation of images and video streams in order to minimize their storage and transmission requirements. This takes advantage of the high degree of redundancy present in these signals. Most image and video compression techniques involve a high degree of uniform computations on a large number of independent groups of pixels. This makes them highly suitable for implementation on CRAM. This section, presents algorithms and CRAM implementation of Vector Quantization and MPEG-2 Motion Estimation, two of the most commonly used image and video compression schemes [74].

7.5.1 Vector Quantization

Most multimedia applications are CD-ROM based and hence require only real-time decoding of the compressed video clips [74]. Unfortunately, most low-end machines cannot perform the decompression process in real time if the images are coded using transform-based coders such as JPEG and MPEG. Vector Quantization (VQ) is an alternative compression scheme that processes the image directly in the spatial domain. Its encoding processing is much more complex than transform-based coders, but it allows for very fast decoding using simple table lookups. Its simple decoder is the main reason why VQ is the method of choice for image and video coding on computing platforms with limited resources such as PCs and portable electronic notepads [74].

To compress an image using VQ, the input image is first divided into blocks of N x N pixels. These blocks are called M-dimensional vectors (where $M = N^2$). Each vector is then compared to each codeword of a previously generated K-word codebook to determine the codeword that best matches the vector (codebooks are generated using a collection of representative images). The index of the best matching codeword is transmitted or stored instead of the N^2 pixels, thus achieving a compression ratio of N^2:1. At the receiving (decoding) end, the image is reconstructed by using this index to point into a simple table lookup of an identical codebook. Common VQ distortion measures used in the matching process are mean square error (MSE) and mean absolute error (MAE). Given an N x N

input vector $V = \{v_0, v_1, \ldots, v_{M-1}\}$ and a set of K M-dimensional codebook vectors

$B^0 = \{b_0^0, b_1^0, \ldots, b_{M-1}^0\}$, $B^1 = \{b_0^1, b_1^1, \ldots, b_{M-1}^1\}$,...,

$B^{K-1} = \{b_0^{K-1}, b_1^{K-1}, \ldots, b_{M-1}^{K-1}\}$, MSE and MAE are calculated as

$$MSE(V,B^i) = \sum_{j=0}^{M-1} (v_j - b_j^i)^2, \quad 0 \le i < K \tag{7.7}$$

$$MAE(V,B^i) = \sum_{j=0}^{M-1} |v_j - b_j^i|, \quad 0 \le i < K \tag{7.8}$$

Image vectors are typically 2 x 2 or 4 x 4 pixels, and common codebook sizes range from 64 to 1024 words.

Algorithm 7.8 shows the VQ encoding algorithm for an image of I x J pixels. Its implementation on CRAM for a 512 x 512 8-bit image and a block size of 2 x 2 pixels is shown in Figure 7.5. Execution time for a 256-word codebook is 12.72 ms on CRAM, 74.95 s on the PC, and 38.22 s on the workstation. This represents speedups of 5892 and 3005.

```
/* on uniprocessor */
for image vector=0 to (I*J)/M - 1) vectors
  for codeword=0 to K-1 codewords
    reset distortion to zero;
    for vector pixel=0 to M-1 pixels
      accumulate distortion between vector pixel
      and corresponding pixel of the codeword;
    end for
    if distortion is the minimum so far
      store distortion and assign codeword index
      as the vector code in the coded image;
    end if
  end for
end for
```

```
/* on CRAM */
// image vectors are stored on PE's local memory
for codeword=0 to K-1 codewords
  in parallel
    reset distortion to zero;
  end in parallel
  for vector pixel=0 to M-1 pixels
    in parallel
      accumulate distortion between vector pixel
      and corresponding pixel of the codeword;
    end in parallel
  end for
  in parallel
    if distortion is the minimum so far
      store distortion and assign codeword index
      as the vector code in the coded image;
    end if
  end in parallel
end for
```

Algorithm 7.8 Vector Quantization

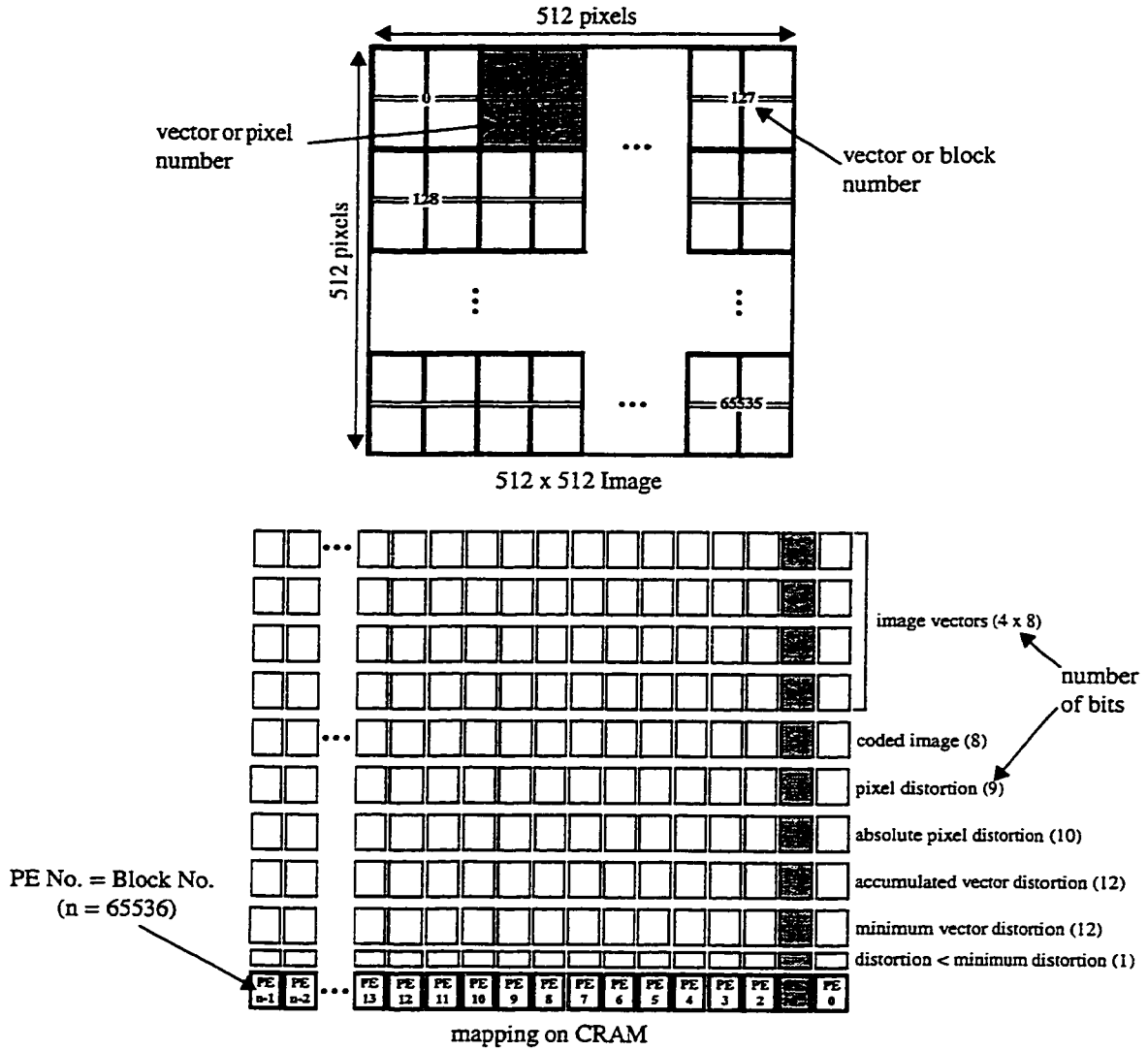

Figure 7.5 CRAM Implementation of Vector Quantization

7.5.2 MPEG-2 Motion Estimation

One of the most compute-intensive operations in the MPEG-2 video compression scheme is the motion estimation process [74]. If we define image frames at times t and t-n as the current and reference frames respectively, then the objective of motion estimation is to determine the motion vector of an N x M block between the two frames. These vectors are used in forming compressed frames (P-frames and B-frames) that are transmitted between the minimally-compressed reference I-frames.

The motion vector of a block C at location (x,y) in the current frame is found by searching the region $[-p,p]$ around the block in the reference frame (Figure 7.6). There are $(2p+1)^2$ search locations in this region. The best matching block R is defined as a block at location $(x+i, y+j)$ in the reference frame for which the mean absolute error (MAE) between its pixels and the pixels in C is minimum. If the pixels in C and R are denoted as $C(x+k, y+l)$ and $R(x+i+k, y+j+l)$ for $0 \le k < M$ and $0 \le l < N$, then the MAE is defined as

$$MAE(i,j) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} |C(x+k,y+l) - R(x+i+k, y+j+l)|, \quad -p \le i,j \le p \qquad (7.9)$$

The choice of N, M, and p is a compromise between accuracy and computation complexity. Small values (from 4 to 8) are preferable since the local smoothness constraint of the motion vector is easily met, while bigger values are more efficient for fast algorithms [74].

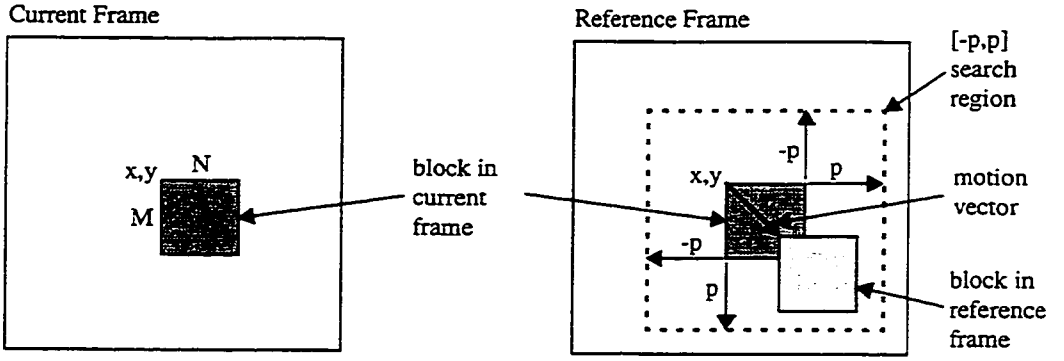

Figure 7.6 Motion Estimation

Algorithm 7.9 and Figure 7.7 illustrates the CRAM implementation of full-search motion estimation of 1024 x1024 8-bit picture frames using a block size of 4 pixels. The CRAM execution time of 19.39 ms represents a speedup of 1437 over the PC (27.87 s) and 1089 over the workstation (21.12 s). Note that computations in the 3 x 3 block neighborhood requires that data be shifted from the neighboring PEs. This uses a shift algorithm similar to that used for pixel neighborhoods (Algorithm 7.3).

```
/* on uniprocessor */
for each of the (image size/block size) blocks
  for each of the (2p+1)^2 search locations around
  the block
    reset MAE to zero;
    for each of the (p x p) pixels in the block
      accumulate MAE between block pixel and the
      corresponding pixel in the search area;
    end for
    if MAE is less than the minimum MAE so far
      store MAE as minimum MAE;
      save the search location as the motion vector
      of the block;
    end if
  end for
end for
```

```
/* on CRAM */
// reference and current frames are stored one block
// per PE
for each of the 9 search blocks around current block
  if not already in temp variables
    in parallel
      shift blocks required for pixel comparison into
      temp variables;
    end in parallel
  end if
  for each search location in the search block
    in parallel
      reset MAE to zero;
    end in parallel
    for each of the (p x p) pixels in the block
      in parallel
        accumulate MAE between block pixel and
        the corresponding pixel in the search area;
      end in parallel
    end for
    in parallel
      if MAE is less than the minimum MAE so far
        store MAE as minimum MAE;
        save the search location as the motion
        vector of the block;
      end if
    end in parallel
  end for
end for
```

Algorithm 7.9 Motion Estimation

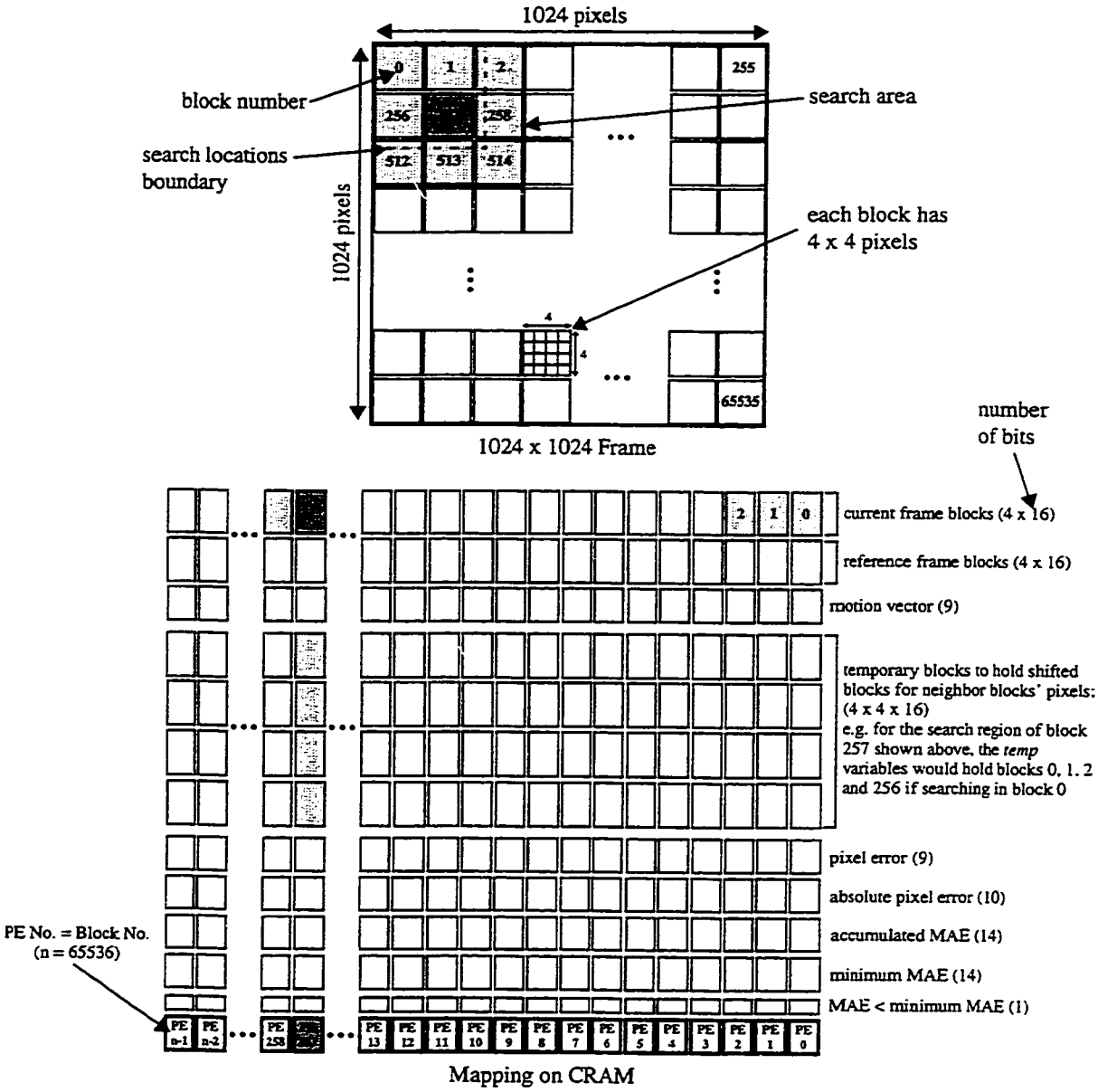

Figure 7.7 CRAM Implementation of Motion Estimation

7.6 Performance Analysis

7.6.1 Applications Performance Summary

Table 7.4 summarizes the performance of CRAM compared to the two uniprocessor systems for the applications describes in this chapter. Two cases of CRAM are considered. The first represents the case where CRAM is used as either part of the system main memory, or as the graphics or video RAM. Therefore the total execution time does not include the time to transfer application data between the host and CRAM. The second case is when CRAM is used just as an accelerator card for massively parallel applications. In this case, data is normally resident on the host and hence the total time to execute an application on CRAM has to account for the I/O overhead of transferring data between the host and CRAM. As noted before, even for the case that CRAM is not used as the system main memory or video RAM, it is not always the case that data has to be transferred between the host and CRAM for each operation. This is especially true for basic operations such as low-level image processing and basic searches because a number of such operations might need to be applied to the same data before the data is used elsewhere. For example, image enhancement operations such as brightness and contrast

Application	Work-station Execution time (ms)	PC Execution time (ms)	CRAM without I/O Overhead		CRAM with I/O Overhead	
			Execution time (ms)	Speedup over PC	Execution time (ms)	Speedup over PC
Brightness adjustment	2.72	7.44	0.0091	818	1.679	4.4
Spatial average filtering	31.78	53.13	0.3896	136	2.06	25.8
Edge enhancement	14.83	31.4	0.2474	127	1.92	16.4
Conversion to binary image	2.31	5.23	0.0093	562	1.679	3.1
Multiple-threshold segmentation	56.54	138.41	0.2686	515	1.94	71.4
Equal-to search	2.94	5.96	0.0138	431	6.694	0.89
Maximum search	3.84	6.85	0.015	457	6.695	1.02
Greater-than search	3.54	7.06	0.0128	551	6.693	1.06
Between-limits search	4.67	8.73	0.0196	445	6.7	1.3
Least means squared match	68.96	81.7	0.9686	84	14.33	5.7
Vector quantization	38222.4	74945.9	12.72	5892	19.4	3863
Motion estimation	21121.4	27869.4	19.39	1437	72.83	383

Table 7.4 Applications Performance Summary

enhancement are usually followed by filtering [11], [69]. Similarly, to find the spread of records from the maximum and minimum values in a database, both maximum and minimum searches, and some computations and record updating have to be performed on the same data. Therefore, even for the case where I/O overhead is to be included, operations might give higher speedups than the ones shown for this case in Table 7.4. Only the first operation on the data suffers the low speedup due to I/O overhead.

From Table 7.4, it is evident that CRAM yields substantial speedups over the uniprocessor for a variety of practical applications. There are higher speedups for applications that have a high degree of computations within each parallelized operation. These include Vector Quantization and Motion Estimation. Low speedups are evident in applications that either have multiplications and divisions, or use PE neighborhood computations, or have SIMD complexity of $O(n)$ or greater.

7.6.2 Controller and System Performance

Previous CRAM performance measurements [9], [4], [5] have been based on an ideal CRAM system with no control and host overhead. In this case, the reported CRAM execution times have merely been the time that the PEs are busy executing CRAM microinstructions. In a practical CRAM system, CRAM code is run from the host computer. Because of the smaller bandwidth at the host external bus, the challenge is to design a CRAM controller such that the practical CRAM system still results in significantly better performance when compared to a uniprocessor system. This section analyzes the performance of the controller and the whole system by comparing the execution times of a few practical applications on three CRAM systems: an ideal system (no control and host overhead), a practical system (described in this thesis), and an unoptimized system (without the performance-enhancement features of the controller, i.e. no instruction FIFO, no read/write buffers, and no buffer-based constant unit).

On a practical CRAM system, the total execution time for an application is made up of PE execution time, control overhead, and host overhead. PE execution time is the time that the PEs are busy executing CRAM microinstructions. This is also the total execution time on an ideal CRAM system. The other two components are defined only for the case when

no CRAM instructions are executing, i.e. when the functions described by these overheads are not executed in parallel with the execution of CRAM microinstructions. Host overheads include the time that the host executes the CRAM compiler code, sequential components of the application code, and other activities due to its interaction with CRAM. The later component depends heavily on the type of interface with the CRAM controller as will be shown later. Code execution delays on the host due to inherent activities on the system, are also bundled into host overheads. These delays are difficult to characterize. But the advantage of the simulation strategy adopted in this work (Section 6.7) is that you don't have to know or characterize these delays - they are automatically reflected into the execution time. Control overhead includes the time to load instructions into the CRAM FIFO, the overhead of filling the CRAM instruction queue, and the time to read/write data from/to the CRAM buffers and registers. Figure 7.8 shows these times for the applications described in this chapter.

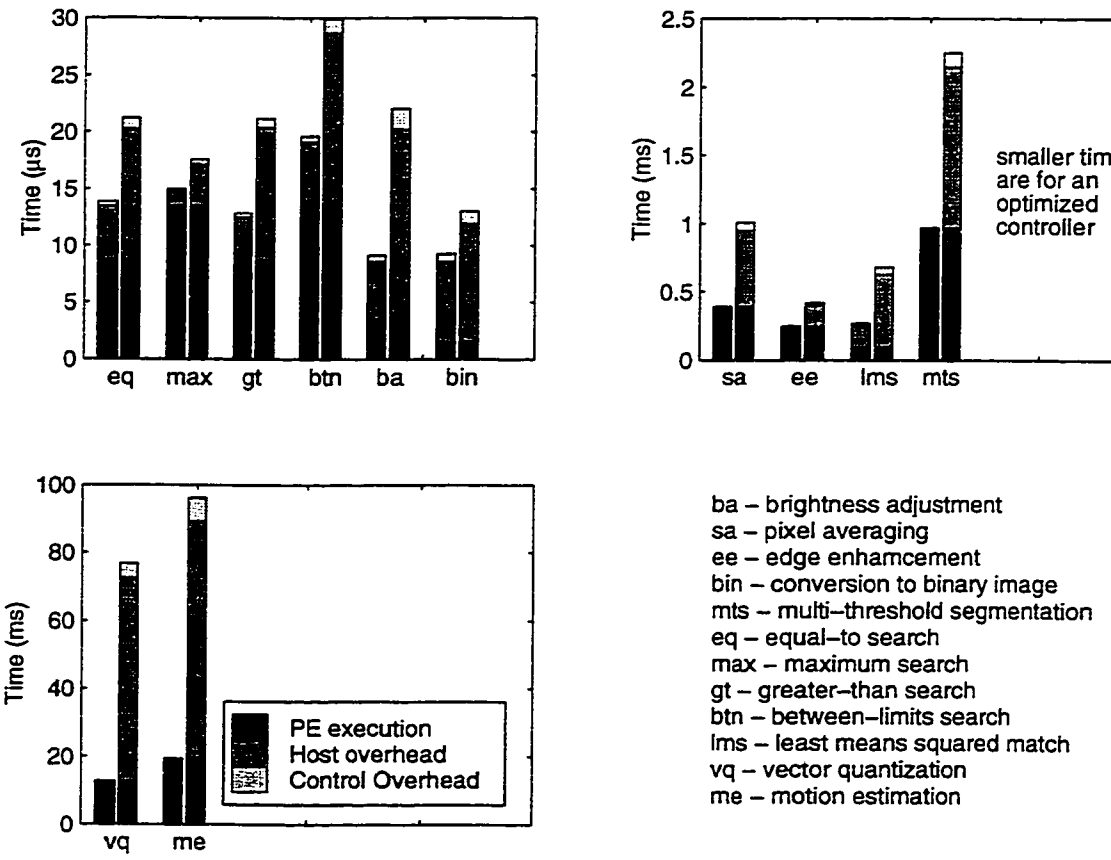

Figure 7.8 Effect of Controller on Applications Execution Times

From Figure 7.8, it can be seen that the performance-enhancement features of the controller help to reduce both the host and control overhead. Notice that the biggest component of the host overhead is the one due to the interaction of the host with CRAM. For an unoptimized controller, this component increases because the host does more data transfer retries, performs more conditional testing, executes more conditional code, sets up more data transfers, does more looping, etc. The control overhead also increases because the instruction queue is empty most of the time. This increases the overhead of filling the queue, and also means that most of the instruction-loading, transfer of data to/from CRAM registers and buffers, and other host activities are not done in parallel with the execution of CRAM instructions. Table 7.5 summarizes the impact of the controller performance-enhancement features on the performance of practical applications. For applications in which the PE execution time is very small, the difference between an ideal CRAM system and a practical CRAM system is very big because the total execution time of an application is dominated by host overheads. On the other hand, for applications with more parallel than sequential code, the CRAM system approaches an ideal system. This is good because naturally we would want to have the best CRAM performance possible when execution times are bigger. Otherwise the fact that the system efficiency is low for applications with very small PE execution time is not of much consequence because as

Application	**Ideal CRAM** Execution time (ms)	**CRAM with an Optimized Controller**		**CRAM with an Unoptimized Controller**	
		Execution time (ms)	PE Utilization (%)	Execution time (ms)	PE Utilization (%)
Brightness adjustment	0.0035	0.0091	38.46	0.0221	15.6
Spatial average filtering	0.3827	0.3896	98.22	2.1045	18.2
Edge enhancement	0.2408	0.2474	96.33	0.4152	58
Conversion to binary image	0.0015	0.0093	16.13	0.013	11.2
Multiple-threshold segmentation	0.0892	0.2686	33.21	0.6791	13.1
Equal-to search	0.0087	0.0138	63.04	0.0212	40.8
Maximum search	0.0134	0.015	89.3	0.0175	76.4
Greater-than search	0.0087	0.0128	68	0.0212	40.9
Between-limits search	0.0139	0.0196	70.92	0.0299	46.3
Least means squared match	0.9651	0.9686	99.64	2.2504	42.9
Vector quantization	12.4	12.72	97.5	76.69	16.2
Motion estimation	19.17	19.39	98.87	96.14	20

Table 7.5 Controller and System Performance

shown in Table 7.4, the actual CRAM speedup for such applications is even higher than for applications of equivalent complexity but higher system efficiency. The combination of high speedup and low system efficiency is just an indication that running the application on CRAM has reduced tremendously the execution time of the parallelized code such that the total execution time is now dominated by the sequential code and other host overheads.

7.6.3 Degree of Parallelism

As mentioned earlier, the performance of CRAM stems from the massive number of processors rather than the computational power of the individual processors. Therefore, it is important to assess how the number of PEs affect the performance of an application. Figure 7.9 shows how the CRAM speedup over the PC changes as the application is parallelized over a varying number of PEs. In this analysis, the vector size is matched to the number of PEs.

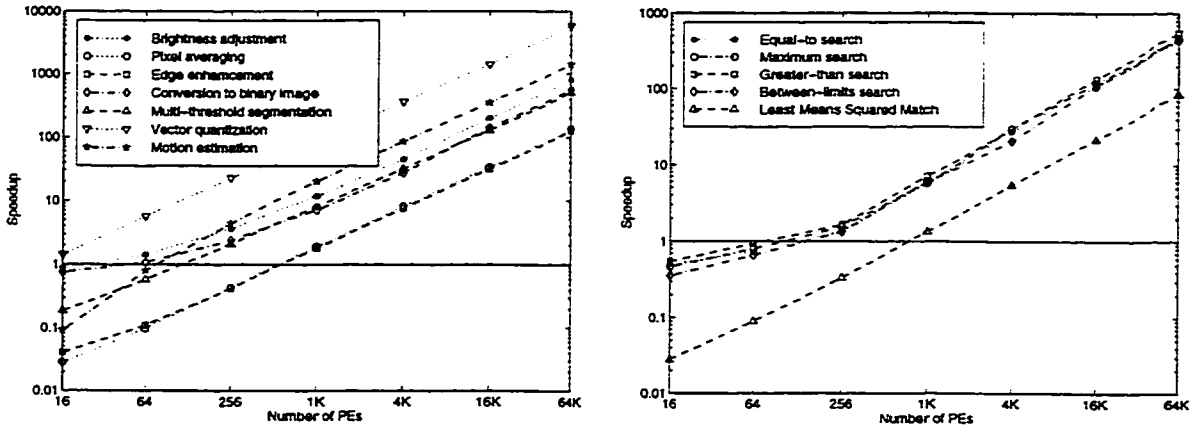

Figure 7.9 Effect of Degree of Parallelism on Applications Performance

There are three points to note from Figure 7.9. First, because of the high overhead of inter-processor communication, applications in which computations involve data from neighbor PEs exhibit lower speedups when compared to applications of similar complexity but with pointwise operations. Second, as expected, CRAM performs better than the uniprocessor PC if the number of PEs is large. However, what is important is to

determine the number of PEs at which this speedup is significant. It is evident from Figure 7.9 that this depends on the type and complexity of the application. Applications such as Vector Quantization and Motion Estimation, that have more operations within the parallelized code, generally attain reasonably significant speedup (greater than 10) at relatively smaller number of PEs (greater than 256). On the other hand, the more basic operations of low-level image processing and database searches attain such speedups at PE numbers greater than 4K. Lastly, while for large number of PEs the CRAM speedup of almost all applications increases linearly with the number of PEs, this is not the case when simple operations are parallelized over a small number (16 to 128) of PEs. This area of operation is magnified in Figure 7.10. In this case, increasing the number of PEs does not result into a proportional increase in speedup. The reason for this is that for such basic operations, the uniprocessor execution time for the main operations in the code (addition, assignment, etc.) is very small when compared to other host overheads. Therefore, the total execution time for the application becomes dominated by these overheads and hence increases slowly with the number of vectors. Also note that for this case, CRAM either performs worse than, or yields very small or no speedup, compared to the uniprocessor. In other words, the execution time of the parallelized code is so small such that the control and host overheads make the total execution time on CRAM almost equal to or bigger than that on the uniprocessor. Therefore, a CRAM system with less than 1K PEs would not be suitable for general-purpose parallel-processing, but could still be used for specific applications such as Vector Quantization and Motion Estimation.

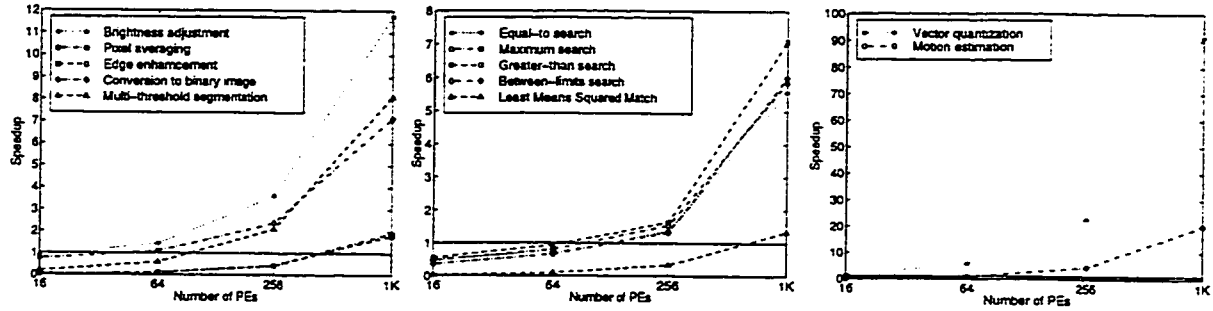

Figure 7.10 CRAM Speedup for Small Number of PEs

7.6.4 Comparison with other SIMD Systems

The main objective of CRAM is to allow conventional uniprocessor systems to attain high performance speedups when executing massively parallel applications, but with hardware far less than that of standard supercomputers. For this reason, the performance of CRAM has been compared consistently with uniprocessor systems, especially the PC. However, in order to justify the cost-effectiveness of CRAM, it is important to also know how CRAM compares with other SIMD and PIM systems. Table 7.6 shows the performance and other characteristics of two of the most popular SIMD supercomputers (the Connection Machine, CM-2, and the Massively Parallel Processor, MPP), four processor-in-memory (PIM) systems, and a 64K-PE CRAM. Performance measures used are those given in the literature for these systems. For many systems, these are in terms of Giga (10^9) Instructions per Second (GIPS) for 8-bit addition and multiplication. Where this is not available, the system application-specific performance measure is used.

From Table 7.6, we see that CRAM performance compares favorably with that of SIMD supercomputers, and CRAM has much less hardware than these systems. When compared to other PIM systems, CRAM has comparable or better performance, has the smallest hardware for all reported large-scale (greater than 4K PEs) PIM systems, and is the only general-purpose PIM system that has been implemented on the most widely used computer platform, namely the PC. It is also interesting to note that even when compared to the highly application-specific bit-parallel PIM systems such as the IMAP and FMPP-VQ64, CRAM performance is still comparable or even better, and it has less hardware when viewed on the basis of performance per PE area.

SYSTEM	No. of PEs	RAM bits/ PE	Inter-PE Communi-cation	PE Complexity	System Physical Attributes	PERFORMANCE				Types of Appli-cations
						8-bit add-ition (GIPS)	8-bit multip-lication (GIPS)	Others	Cycle Time (ns)	
MPP [6]	16K	1K	2-D grid	1000 transistors/ PE 1b PE, 6 1-bit regs, 1 full adder, shift reg & logic	large cabinet with 88 PCBs and 2100 PE chips plus memory chips	6.5	1.9	--	100	image processing
CM-2 [73]	64K	64K	2-D grid, hypercube network, router	1 PE chip has 16 PEs and is about 14,000 gates; 1b PE	large 1.5m x 1.5m cabinet, 152 PCBs, 4096 PE chips, 4096 floating-pt chips, 22528 RAM chips	4	3.3	--	--	general purpose
MIT-PP [14]	16K	128	2-D grid	1b PE with 8-to-1 mux, 5 latches	4 PCBs, 4 PE-RAM chips, 3 64Kbx32 RAM chips, 2 CPLDs, 10 reg chips, 16 64Kbx4 chips, 2 FPGAs	4.3	0.5	--	60	image processing
SRC-PIM [13]	32K	2K	parallel prefix & partitioned-	1b ALU, 3 regs, 3 muxes, & logic	external enclosure, 16 PE boards with 512 chips, 2 interface boards	--	--	$3.2x10^{11}$ bit operations/s (BOPS)	100	general purpose
IMAP [11]	512	32K	1-D (shift left/right)	7000 transistors, 8b ALU & shifter, 14 8b regs,	1 board with 8 PE chips, 1 control chip, 1 video PCB, 1 host PCB, 2 mem chips	--	--	3x3 average filter on 512 x 512 in 0.42 ms	25	image processing
FMPP-VQ64 [17]	64	128	--	12b ALU with 3 12b registers	1 PC card with 1 PE chip and 1 control chip	--	--	53,000 nearest neighbor search (NNS)/s	40	VQ
CRAM	64K	4K	1-D (shift left/right)	1b PE with 75 transistors (8-to-1 mux, 3 1b regs)	1 PC card with 8 or 16 PE-RAM chips, 1 control chip or 1 (FPGA + CPLD + EPROM)	27	2	$1.3x10^{12}$ BOPS $82x10^{6}$ NNS/s; 512x512 filter in 0.8 ms	50	general purpose

Table 7.6 System Comparisons

7.7 Summary

This chapter has looked at CRAM applications and analyzed the performance of the CRAM system. With regard to applications, the objective has not been to present an exhaustive review of all possible applications, but rather to use a few selected applications of varying execution models and complexity to study different CRAM performance issues. Using this approach, it has been shown that the type of applications most suited for CRAM are those that have fine grain parallelism and regular communication patterns. The implementation of different application structures on CRAM has also been demonstrated. For example, the use of the CRAM 1-D inter-PE communication network to implement 2-D communications during PE neighborhood computations has been described. This has also highlighted the inter-PE network as a major architectural bottleneck on CRAM. Even with this bottleneck, CRAM yields significant performance speedup over conventional uniprocessor systems when executing these massively parallel applications. The impact of the CRAM controller on the performance of a CRAM system has also been analyzed. In this regard, it has been shown that even with a bandwidth-limited host, a CRAM system still yields reasonably high performance because of the performance-enhancement features of the controller. Finally, it has been shown that even with less hardware and a simpler architecture, the performance of a CRAM system compares favorably with that of conventional supercomputers. Also, CRAM performs either better, slightly worse, or almost the same when compared to other (mostly application-specific) logic-in-memory systems. However, it has less hardware and is less complex than these systems.

Chapter 8

Conclusions and Future Work

This thesis describes the system design issues for a computational-RAM logic-in-memory parallel processing machine. This includes the architecture and implementation of the controller for the CRAM processing elements, the interface of CRAM to the host computer, and the development of CRAM system software tools. This chapter provides a summary of the thesis and presents some ideas for future work.

8.1 Summary

Integrating several 1-bit processing elements at the sense amplifiers of a standard RAM improves the performance of massively-parallel applications because of the inherent parallelism and high data bandwidth inside the memory chip. However, implementing such a system on a host computer such as the PC poses several challenges in controlling the processing elements, exchanging data between the host and CRAM, and writing CRAM application programs. This is so because of the small bandwidth at the host system buses, the differences in the bit-serial and bit-parallel data formats on CRAM and the host, respectively, and the constrained size of a PC expansion card. The CRAM controller and software tools developed in this thesis provides solutions to these system design challenges.

The area of the controller has been minimized by using a simplified microprogram

sequencer and grouping microinstructions such that the size of the control store is halved. This reduces system cost, allows easy implementation of a CRAM system on a single PC card, and facilitates future integration of the controller and the PE array on the same chip. Several architectural features are used to enhance the performance of PE control and interaction with the host computer. A FIFO-based instruction queue unit improves PE utilization, especially for instructions with a small number of microinstructions. Buffers reduce the time of transferring data between the host and CRAM, simplifies synchronization, and eases electrical and physical loading of the host bus. A new constant broadcast unit improves the performance of operations with constants, reduces the size of the control store, and simplifies the use of variable-size constants. A new parallel array-based data corner-turning scheme greatly reduces the time of software data transposition, thus removing the need for the high-cost hardware used to convert data between bit-serial and bit-parallel formats. These performance-enhancement features also minimize the effect of the host on the performance of CRAM. This allows the implementation of CRAM on a variety of platforms, including those with slow host buses and processors such as ISA computers and embedded systems with slow microcontrollers. In general, the high-performance, minimal-hardware and general-purpose architecture of the CRAM controller enhances the use of CRAM as a general-purpose parallel-processing system.

PCI and ISA CRAM controller prototypes have been implemented in a Xilinx XC4013EPQ240-2 FPGA. A 64-PE ISA CRAM system prototype has been built and tested as an expansion card in a 133 MHz Pentium PC under both Linux and MSDOS. These prototypes are used to demonstrate working models of the CRAM concept.

Four high-level software tools are used to hide low-level architectural details of the CRAM system, allowing software developers and system analysts to focus on implementation details. The CRAM C++ compiler, which is simply a library of CRAM parallel variables, is used to write CRAM programs using the standard C++ language and standard C++ compilers. The CRAM assembler provides optional low-level control of CRAM hardware. The CRAM microcode assembler is used for generating CRAM microinstructions. The CRAM C++ simulator simulates the behavior of a CRAM system and can be used by both hardware and software designers to analyze CRAM architectural features as well as the performance and implementation of applications. A CRAM system

VHDL simulation model allows logic designers to easily simulate the behavioral, gate-level, or post-place-and-route VHDL designs of the controller and CRAM chips by running the actual CRAM system assembly or machine code.

Comparing a 20 MHz 32 Mbytes 64K-PE PCI CRAM system with a 133 MHz 32 Mbytes Pentium PC and a 167 MHz 64 Mbytes SUN Sparc Ultra workstation show that CRAM yields significant performance speedup over conventional uniprocessor systems when executing massively-parallel applications. Comparisons with conventional supercomputers and other logic-in-memory systems show that CRAM has comparable speed but uses less hardware and has a less complex architecture.

8.2 Future Work

Two new ideas are suggested for the continuation of this work. These are an on-chip CRAM controller and a MIMD-SIMD CRAM system, as well as a pipelined constant-sensitive PE architecture. Other ideas for general future work are also suggested in this section.

8.2.1 An On-Chip Controller and a MIMD-SIMD CRAM System

With the general architectures of CRAM and its controller now established, another interesting step is to explore the possibility and analyze the pros and cons of putting the controller on the same chip as the PE/memory array. One obvious disadvantage is that CRAM now becomes less compatible with standard RAM, both in fabrication (more logic) and pinout (an extended pinout to support system bus signals). But there are many potential gains of this approach. The data access time between the controller and CRAM may be significantly reduced. Part of the CRAM memory, say the first 4 Kbytes, may be used as the control store. This would reduce the size of the controller logic, and may improve performance by dynamically changing the size of the microprogram memory (on a program-by-program basis) to allow more application-specific functions/algorithms to be implemented as microcode. An on-chip controller also reduces the number of chips on the CRAM system PCB. This may reduce system cost and also increase the CRAM size (number of CRAM chips) that can be implemented on a single card.

An on-chip controller also gives rise to a new possible architecture in which two or more CRAM chips are connected to form a MIMD-SIMD CRAM system. This is illustrated Figure 8.1. Each CRAM chip is connected to the host bus, possibly memory-mapped to a specific host address space, and therefore may operate on a different instruction/program (in normal SIMD stye) than the other chips. A 1-bit register on the controller may allow the chip(s) to be connected as one big SIMD array or as components of a MIMD system. Other issues to be studied include inter-controller communication and the number of PEs per controller (such as transistors per controller versus number of PEs x transistors per PE).

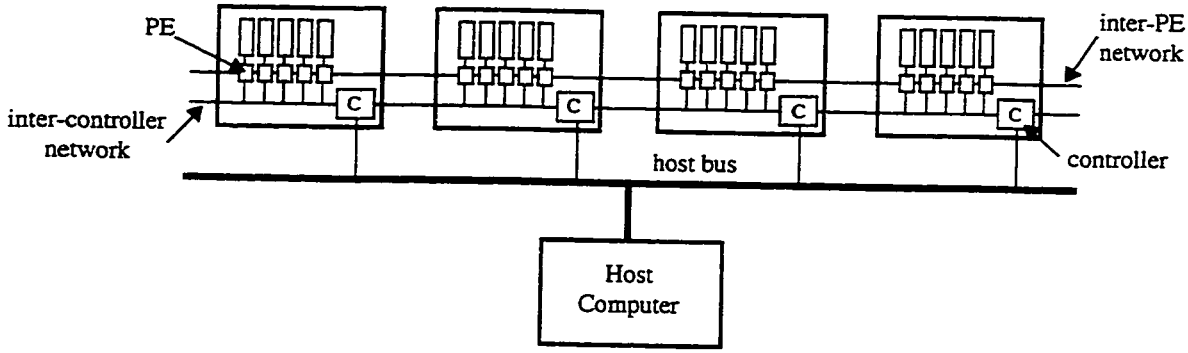

Figure 8.1 A MIMD-SIMD CRAM System

8.2.2 A Pipelined Constant-Sensitive PE

With an on-chip controller, the address and data buses of CRAM would no longer be multiplexed with PE control opcodes (TTOP/COP). This would allow an internal PE memory access to be executed in parallel with a PE operation. This can be accomplished by putting a latch at the output of the PE. This pipelining would reduce both the execution time and the size of the microprogram memory. The PE may also be enhanced to directly support operations with constants by multiplexing one or all three PE source registers with the bit coming from the constant broadcast unit. This would remove the need for an LDK instruction, thus reducing the execution of n-bit operations by n clock cycles. It may also slightly reduce the size of the control store.

Other improvements not related to an on-chip controller involves the inter-PE communication network. In the current architecture, the output of a PE shift operation is

hardwired to a specific register of the neighbor PE. This may be improved by making the output of a PE during shift operations equal to the output of its neighbor PE. This would allow the shifted value to be written to any of the three destination registers, thus reducing the number of microinstructions in such routines as data transposition. The inter-PE communication network may also be upgraded to a 2-D network. All these PE improvements will have to be carefully weighted against the increase in the PE area so that the total area of computational logic is still a small fraction (may be less than 15%) of the total area of the PE/RAM array.

8.2.3 Other Work

There are several other ideas for the continuation of this work. A bigger CRAM system prototype needs to be built so that more applications can be tested. This may require fabricating bigger CRAM chips, with the errors in the current chips corrected (A critical error is the bus-tie because it allows CRAM chips to be connected together to form bigger systems). An ASIC controller may also be fabricated for use in the prototype.

The software tools, especially the C++ library, also need to optimized so that the time that the host processor takes to execute this code is minimized. This might require changing the definition, functionality, hierarchy or code-ordering of some classes, or simply applying standard code optimizations, such as loop unrolling and code in-lining. The library may also be extended to support floating-point variables.

On the architecture and implementation of the controller, several things could be done. The control store could be enlarged so that more frequently used functions are implemented as microcode. The instruction queue unit could be extended to support looping and/or branching (with possible significant increase in the size of the controller). Variable-length instructions, or other instruction formats, could be explored. Finally, for embedded application-specific uses, an architecture using a hard-wired controller could be implemented with most of the general-purpose features stripped out. CRAM systems targeted for other buses, such as AGP and Sbus could also be designed.

References

[1] Harold S. Stone, "A Logic-in-Memory Computer", *IEEE Transactions on Computers*, pp 73-78, January, 1970.

[2] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design*, Second Edition, Morgan Kaufmann Publishers, San Francisco, 1997.

[3] D. G. Elliott, M. Snelgrove, and M. Stumm, "Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP", *Custom Integrated Circuits Conference*, May, 1992.

[4] D. G. Elliott, *Computational RAM: A Memory-SIMD Hybrid*, Ph.D. Thesis, University of Toronto, December, 1997.

[5] Thinh M. Le, *Visual Communications on a Memory-Embedded Array Processor: The Computational RAM*, Ph.D. Thesis Proposal, University of Ottawa, 1998.

[6] J. L. Potter, *The Massively Parallel Processor*, MIT Press, Cambridge, 1985.

[7] Harold S. Stone, *High-Performance Computer Architecture*, Addison-Wesley Publishing Co., 1990.

[8] D. G. Elliott, W. M. Snelgrove, C. Cojocaru, and M. Stumm, "A PetaOp/s is currently feasible by Computing in RAM", *PetaFLOPS Frontier Workshop*, February, 1995.

[9] R. McKenzie, *A DRAM-Compatible Parallel Processor for Real-Time Video*, Masters Thesis, Carleton University, December, 1997.

[10] Noboyuki Yamashita, et al, "A 3.84 GIPS Integrated Memory Array Processor with 64 Processing Elements and a 2-Mb SRAM", *IEEE Journal of Solid-State Circuits*, Vol. 29, No. 11, pp 1336-1343, November, 1994.

[11] Shin'ichiro Okazaki, et al, "A Compact Real-Time Vision System Using Integrated Memory Array Processor Architecture", *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 5, No. 5, pp 446-452, October 1995.

[12] Yoshiharu Aimoto, et al, "A 7.68 GIPS 3.84 GB/s 1W Parallel Image-Processing RAM Integrating a 16 Mb DRAM and 128 Processors", *IEEE International Solid-State Circuits Conference*, pp 372-373, February, 1996.

[13] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massivel Parallel PIM Array", *IEEE Computer*, pp 22-31, April, 1995.

[14] J. C. Gealow, F. P. Herrmann, L. T. Hsu, and C. G. Sodini, "System Design for Pixel-Parallel Image Processing", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp 32-41, March, 1996.

[15] David Patterson, et al, "A Case for Intelligent RAM", *IEEE Micro*, pp 34-44, April,1997.

[16] Kazutoshi Kobayashi, et al, "A Memory-Based Parallel Processor for Vector Quantization: FMPP-VQ", *IEICE Trans. Electron*, Vol. E80-C, No. 7, pp 970-975, July, 1997.

[17] Kazuhiko Terada, "An LSI for Low Bit-Rate Image Compression Using Vector Quantization", *IEICE Trans. Electron*, Vol. E81-C, No. 5, pp 1-7, May, 1998.

[18] R. Torrance, et al, "A 33 GB/s 13.4 Mb Integrated Graphics Accelerator and Frame Buffer", *IEEE International Solid-State Circuits Conference*, pp 340-341, February,

1998.
[19] D. G. Elliott and W. M. Snelgrove, "CRAM: Memory with a Fast SIMD Processor", *Proceedings of the Canadian Conference on VLSI*, pp 3.3.1-3.3.6, October, 1990.
[20] D. G. Elliott and W. M. Snelgrove, *Method and Apparatus for a Single Instruction Operating Multiple Processors on a Memory Chip*, U. S. Patent 5546343, Issued August 1996, Filed October, 1990.
[21] D. G. Elliott, M. Stumm, and W. M. Snelgrove, "Computational RAM: The Case of SIMD Computing in Memory", Workshop on Mixing Logic and DRAM, ISCA '97, pp 6.1-6.7, June, 1997.
[22] Christian Cojocaru, *Computational RAM: Implementation and Bit-Prallel Architecture*, Master's Thesis, Carleton University, January, 1995.
[23] H. L. Kalter, et al, "A 50-ns 16-Mb DRAM with a 10-ns Data Rate and On-Chip ECC", *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, October, 1990.
[24] Betty Prince, *Semiconductor Memories*, Second Edition, Chichester, England, John Wiley and Sons, 1996.
[25] Peter M. Nyasulu and W. Martin Snelgrove, "Architecture and Implementation of a Computational RAM Controller", *International Conference on Massively Parallel Computing Systems*, Colorado Springs, April, 1998.
[26] W. M. Loucks, M. Snelgrove, and S. G. Zaky, "A Vector Processor Based on One-Bit Microprocessors", *IEEE Micro*, pp 53-62, February, 1982.
[27] C. M. Habiger and R. M. Lea, "Hybrid-WSI: A Massively Parallel Computing technology?", *Computer*, Vol. 26, No. 4, pp 50-61, April, 1993.
[28] Kenneth E. Batcher, "Design of a Massively Parallel Processor", *IEEE Transactions on Computers*, Vol. C-29, No. 9, pp 836-840, September, 1980.
[29] Tom Blank, "The MasPar MP-1 Architecture", *Proceedings of the IEEE COMPCON Spring*, pp 20-24, February, 1990.
[30] J. Mick and J. Brick, *Bit-Slice Microprocessor Design*, McGraw-Hill Book Company, 1980.
[31] C. Fernstrom, I. Kruzela, and B. Svensson, *LUCAS Associative Array Processor*, Springer-Verlag, 1985.
[32] John R. Nickolls, "The Design of the MasPar MP-1: A Cost-Effective Massively Parallel Computer", *Proceedings of the IEEE COMPCON Spring*, pp 25-28, February, 1990.
[33] W. D. Hillis, *The Connection Machine*, Cambridge Massachussetts, MIT Press, 1985.
[34] Tom Shanley and Don Anderson, *PCI System Architecture*, Third Edition, Addison-Wesly Co, 1995.
[35] Warren Andrews, "PCI: The Universal Socket", *RTC Magazine*, Vol. VI, No. 2, pp 19-23, February, 1998.
[36] *PCI Local Bus Specification*, Rev. 2.1, June, 1995.
[37] *Accelerated Graphics Port Interface Specification*, Rev. 1.0, Intel Corporation, July, 1996.
[38] Edward Solari, *ISA & EISA Theory and Operation*, Annabooks, San Diego, CA, 1994.
[39] Wade D. Peterson, *The VME Handbook*, Third Edition, VITA, Scottdale, AZ, 1993.
[40] *Using Select-RAM Memory in XC4000 Series FPGAs*, Xilinx Application Note,

July, 1996.
[41] Jeffrey C. Gealow, *An Integrated Computing Structure for Pixel-Parallel Image Processing*, PhD Thesis, Massachusetts Institute of Technology, June 1997.
[42] Peter M. Nyasulu, Ralph Mason, W. Martin Snelgrove, and Duncan Elliott, "Minimizing the Effect of the Host Bus on the Performance of a Computational RAM Logic-in-Memory Parallel-Processing System", to be presented at *IEEE Custom Integrated Circuits Conference*, San Diego, May, 1999.
[43] R. G. Lange, "High-Level Language for Associative and Parallel Computation with STARAN" *Proceedings of the International Conference on Parallel Processing*, pp 170-176, August, 1976.
[44] Peter Christy, "Software to Support Massively Parallel Computing on the MasPar MP-1", *Proceedings of the IEEE COMPCON Spring*, pp 29-33, February, 1990.
[45] Bjarne Stroustrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, Reading, Massachussetts, 1997.
[46] Kenneth E. Batcher, "The Multidimensional Access Memory in STARAN", *IEEE Transactions on Computers*, February, 1977, pp174-177.
[47] Kenneth E. Batcher, "The Flip Network in STARAN", *IEEE Transactions on Computers*, February, 1977, pp 65-71.
[48] A. Abnous, et al, "Design and Implementation of the 'Tiny RISC' Microprocessor", *Microprocessors and Microsystems*, Vol. 16, 1992, pp 187-193.
[49] S. R. Wang, et al, "A 66 MHz PA-RISC Embedded Controller with 80486DX-Like Bus Interface", *IEEE COMPCON*, Spring, 1994, pp 53-57.
[50] J. Schack, et al, "KISS-16V1: A 16 Bit Signal Processor", *IEEE International Symposium on Circuits and Systems*, Vol. 4, 1991, pp1905-1908.
[51] Peter M. Nyasulu, *Microprocessor Design for Instrument Control*, Masters Thesis, Carleton University, August, 1995.
[52] D. Gajski, "Introduction to High-Level Synthesis", *IEEE design and Test of Computers*, Winter, 1994, pp 45-54.
[53] V. Nagasamy, et al, "Specification, Planning, and Synthesis in a VHDL Design Environment", *IEEE Design and Test of Computers*, 1992, pp 58-68.
[54] C. Jay, "VHDL and Synthesis Tools Provide a Generic Design Entry Platform into FPGAs, PLDs, and ASICs", *Microprocessors and Microsystems*, Vol. 17, Sept., 1993, pp 391-398.
[55] Bob Reese, "Use of VHDL Synthesis in an Advanced Digital Design Course", *IEEE SouthEast Con'92*, Vol. 2, 1992, pp 509-512.
[56] L. Clonts, et al, "Writing Area-Efficient Hardware Descriptions for Logic Synthesis", *IEEE SouthEast Con'92*, Vol. 2, 1992, pp 505-508.
[57] Darren Jones, et al, "Verification Techniques for a MIPS Compatible Embedded Control Processor", *IEEE International Conference on Computer Design*, Oct., 1991, pp 329-332.
[58] Tim Brodnax, "The PowerPC 601 Design Methodology", *IEEE International Conference on Computer Design*, October, 1993, pp 248-252.
[59] Kevin Skahill, *VHDL for Programmable Logic*, Addison-Wesley Publishing, CA, 1996.
[60] *Programmable Data Book*, Xilinx Inc., 1990.
[61] *PCI S5920 developer's Kit User Manual and Technical Reference Manual*, Applied

Micro Circuits Corporation, Revision 1.3, April, 1998.
[62] *S5920 PCI Target Interface Data Book*, Applied Micro Circuits Corporation, 1997.
[63] Tom Shanley and Don Anderson, *ISA System Architecture*, Third Edition, Addison-Wesly Co, 1995.
[64] Tom Shanley, *EISA System Architecture*, New Revised Edition, Mindshare Press, Richardson, TX, 1993.
[65] Thinh M. Le and S. Panchanathan, "Computational-RAM Implementation of an Adaptive Vector Quantization for Video Compression", *IEEE Transactions on Consumer Electronics*, Vol. 41, No. 3, pp 738-747, August, 1995.
[66] Thinh M. Le, S. Panchanathan, and W. M. Snelgrove, "Computational-RAM Implementation of Mean-Average Scalable Vector Quantization for Real-Time Progressive Image Transmission", *Proceedings of the 1996 Canadian Conference on Electrical and Computer Engineering*, Vol. 1, pp 442-445, May, 1996.
[67] T. M. Le, W. M. Snelgrove, and S. Panchanathan, "Computational RAM Implementation of MPEG-2 for Real-Time Encoding", *SPIE Proceedings of Multimedia Hardware Architectures '97*, pp 182-193, February, 1997.
[68] T. M. Le, W. M. Snelgrove, and S. Panchanathan, "Fast Motion Estimation Using Feature Extraction and XOR Operations", *SPIE Proceedings of Multimedia Hardware Architectures '98*, January, 1998.
[69] Zahid Hussain, *Digital Image Processing: Practical Applications of Parallel Processing Techniques*, Ellis Horwood Limited, Chichester, 1991.
[70] Loren Heiny, *Advanced Graphics Programming Using C/C++*, John Wiley & Sons Inc., 1993.
[71] Anil K. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[72] Robert Heaton, Donald Blevins, and Edward Davis, "A Bit-Serial VLSI Array Processing Chip for Image Processing", *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 2, April, 1990.
[73] Lewis W. Tucker and George G. Robertson, "Architecture and Applications of the Connection Machine", *Computer*, Vol. 21, No. 8, pp 26-38, August, 1988.
[74] V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures*, Second Edition, Kluwer Academic Publishers, Boston, 1997.
[75] K. Hwang and F. Briggs, *Computer Architecture and Paralell Processing*, McGraw-Hill Inc., New York, 1984.
[76] A. Bjorck, R. J. Plemmons, and H. Schneider, *Large Scale Matrix Problems*, Elsevier North Holland Inc., New York, 1981.

Appendix A

CRAM Controller Architectural Details

A.1 Microinstruction Word

This appendix describes bit settings for the CRAM microinstruction word shown in Figure 4.8.

A.1.1 Next Address Instruction (NAI)

This 4-bit field (Table A.1) selects the next address of the control store (i.e. the next microinstruction).

Next address Instruction (NAI)	Control Word Bits [31:28]	Description
NEXTI	0000	Go to the first microinstruction of the instruction in IR. This also implements the jump to address zero (JZ) or WAIT/NOP.
NEXTu	0001	Go to the next sequential microinstruction (at address uPC+1).
SETLP	0010	Set looping to begin at next microinstruction (LPR ← uPC+1).
LOOPN	0011	Loop back to the address in LPR, else next microinstruction.
LOOPE	0100	Loop back to the address in LPR, else end.
JSR	0101	Conditional jump to subroutine; save return address (SBR ← uPC+1).
JMP	0110	Conditional jump to address in microinstruction bits [7:0].
	0111-1111	Reserved.

Table A.1 Next Address Instructions

A.1.2 Condition Select (COND)

This field (Table A.2) selects condition to be used in a conditional jump or loop instruction.

Condition	Bits [27:25]	Description	Use
NOCND	000	Always true	unconditional execution
CNZ	001	Loop Counter is not zero	n-looping
CN1	010	Loop counter is not one	(n-1)-looping
GOR	011	CRAM chip Global-OR (Bus-Tie) status	general
SLO	100	CRAM chip shift left output of (n-1)th PE	general
SRO	101	CRAM chip shift right output of PE 0	general
DZ	110	Byte read from CRAM contains all zeroes	max/min searches, etc.
	111	Reserved	Reserved

Table A.2 Microsequencer Conditions

A.1.3 CRAM Function and Read/Write Bit

Microinstruction bits [23:22] indicates the CRAM function being executed in the current microinstruction. the settings are shown in Table A.3. Even though a NOP means that there is no operation on the CRAM chip, a controller instruction (CCI and ACCI, described in the next section) may be embedded in the microinstruction. Bit [21] indicates a read (1) or a write (0).

Bits [23:22]	CRAM Function
00	No Operation (NOP)
01	PE Operation
10	Internal CRAM read/write (data is read from CRAM memory to PE register M, or written from the PE ALU output to CRAM memory)
11	External CRAM read/write (transfer of data between CRAM and controller)

Table A.3 CRAM Functions

A.1.4 External TTOP (EXTTOP)

If EXTTOP is set, TTOP is selected to be the value of operand 2 (OPR2) of the instruction word. Otherwise TTOP is selected as bits [7:0] of the current microinstruction.

A.1.5 Address Select (ASEL)

Bits [20:19] select the value to be put on the CRAM chip address bus. They can be set to select either AR0 (ASEL="00"), AR1 ("01"), AR2 ("10") or bits [15:8] ("11"). Bits [15:8] represent either COP or a system mask/temporary address (TMPA).

A.1.6 CRAM Controller Instructions (CCI and ACCI)

CRAM Controller instructions (CCI) are coded in microinstruction bits [18:16]. These are mainly used for loading parameter registers or incrementing address registers. Table A.4 describes these instructions. To minimize the size of the microinstruction word, all controller instructions that do not execute simultaneously with a CRAM instruction are coded on bits [3:0]. These auxiliary controller instructions (ACCI) are decoded only if CCI is set to ACCI (i.e. bits [18:16] = "111"). These are described in Table A.5.

Instructio (CCI)	Bits [18:16]	Description
NOCCI	000	No controller instruction
INCA	001	After using it, increment the currently-selected address register
DECA	010	After using it, decrement the currently-selected address register
INCB	011	After using it, increment the bank address register
LDK	100	Load a constant's bit into a PE register
-	101	Reserved
XCOP	110	COP is from external (value of instruction operand 1)
ACCI	111	Execute an auxiliary controller instruction

Table A.4 CRAM Controller Instructions

Auxiliary Instruction (ACCI)	Bits [3:0]	Description
	0000	Reserved
LDCBA	0001	Load CRAM bank address (CBA) register
LDIBA	0010	Load the read/write buffer internal address (WIBA & RIBA) registers
LDWLEN	0011	Load the word length (WLEN) register
LDAX0	0100	Load address extension register 0 (AX0)
LDAX1	0101	Load address extension register 1 (AX1)
LDAX2	0110	Load address extension register 2 (AX2)
SETINT	0111	Load (set) the interrupt number
LDSIC	1000	Load the shift-input selection code
LDBAI	1001	Load the buffer address increment registers
	1010-1111	Reserved

Table A.5 Auxiliary Controller Instructions

A.2 Command, Status, and Parameter Registers

A.2.1 Command Register (CCR)

The command register is memory-mapped and can be written and read by the host processor on a byte basis. Writing a 1 to a bit of the command register will toggle that bit. The implemented CCR bits are shown in Table A.6. The IRQn bits are provided in this prototype to allow flexibility in the choice of the ISA interrupt line to use (The PC interrupts can sometimes be very populated and unshareable).

Name	CCR Bit	Function	Value on Reset
Control Store Ready (CSRDY)	14	1 - The control store is ready 0 - Control store not ready	0
Extended PE (XPE)	13	1 - CRAM has extended PEs 0 - CRAM has baseline PEs	0
Bit-Parallel CRAM (BPL)	12	1 - CRAM operates in bit-parallel mode 0 - CRAM operates in bit-serial mode	0
External Timing (XCTRL)	!1	1 - CRAM PEs use external timing 0 - CRAM PEs use internal timing	0
Interrupt Request Lines (IRQ15, IRQ11, IRQ10, IRQ9)	3-0	Indicates (if set) the interrupt line that the CRAM system will use. Only one may be set.	IRQ15 = 1 Others = 0

Table A.6 Command Register

A.2.2 Status Register (CSR)

The status register (CSR) is a 16-bit memory-mapped read-only register. Table A.7 shows the implemented bits of CSR. IRQP is automatically cleared when the host processor reads the byte that contains it (CSR[16:8]).

Name	CSR Bit	Function
Global-OR (GOR)	0	Value of the CRAM Bus-Tie (Global-OR)
Shift-Left Output (SLO)	1	Value shifted out of PE_{n-1} during CRAM shift-left operation.
Shift-Right Output (SRO)	2	Value shifted out of PE_0 during CRAM shift-right operation.
Instructions Executing (IXEC)	3	1 - Instructions are still executing or pending in the queue 0 - No instructions executing; the queue is empty
Interrupt Pending (IRQP)	4	1 - The CRAM system has issued an interrupt 0 - No interrupt has been issued by the CRAM system

Table A.7 Status Register

A.3 PCI Configuration Registers and Commands

This appendix describes how PCI configuration registers are implemented for CRAM. Since CRAM is not a PCI-to-PCI bridge, it implements a PCI device configuration header type zero shown in Figure A.1. The shaded area represents registers that are mandatory. **Header Type** identifies the format of the device configuration header (bits 6:0), and also indicates if the device is a single or multi-function (bit 7). The CRAM controller is a single-function PCI device (i.e. header type register bit 7 is hardwired to 0), and it implements a header type 0 (i.e. header type register bits 6:0 are hardwired to "0000000").

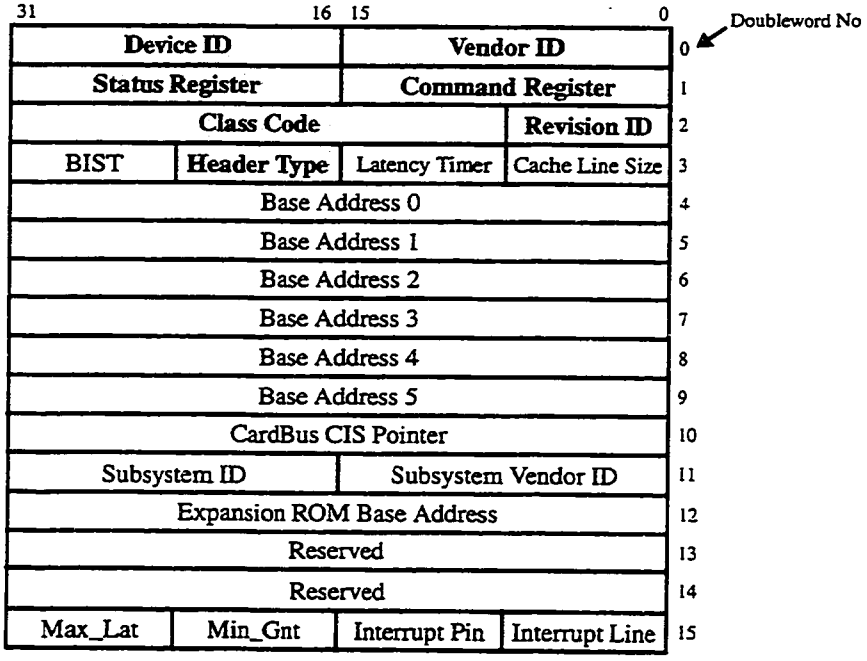

Figure A.1 PCI Device's Configuration Header

- **Vendor ID Register:** This16-bit register identifies the manufacturer of the device and is assigned by the PCI SIG (Special Interest Group). Since our design is just for prototyping purposes, an arbitrary value of 0x1008 is hardwired as the Vendor ID.
- **Device ID Register:** This is a 16-bit register. It is assigned by the device manufacturer to identify the type of the device. The CRAM device ID is hardwired to 0x0000.
- **Revision ID Register:** This 8-bit register is assigned by the device manufacturer to identify the revision number of the device. CRAM revision ID is hardwired to 0x00.
- **Command Register:** The 16-bit command register defines how a PCI device responds to PCI accesses. Table A.8 describes the nine defined bits of the command register and how they are implemented in the CRAM controller. Only bits 8, 6 and 1 are implemented as real registers. This means that for CRAM, these command bits can be set to either 1 or 0 using a PCI configuration cycle. The rest of the command bits have fixed values for CRAM implementation, and hence are just hardwired.

Bit	Function	CRAM Implementation
0	I/O Access Enable 1 - Device responds to PCI I/O accesses 0 - Disable I/O	Hardwired to 0. (All CRAM units/registers are memory-mapped)
1	Memory Access Enable 1 - Device responds to PCI memory accesses 0 - Disable	Register (RESET# to 0; set to 1 to allow memory accesses)
2	Master Enable 1 - Device acts as a bus master 0 - Device can't be master	Hardwired to 0 (CRAM only acts as a PCI target)
3	Special Cycle Recognition 1 - Device monitors special cycles 0 - Special cycles are ignored	Hardwired to 0
4	Memory Write and Invalidate Enable 1 - Master can generate memory write and invalidate 0 - Master only uses memory write commands	Hardwired to 0 (This is only for PCI bus masters)
5	VGA Palette Snoop Enable 1 - Device must perform palette snooping 0 - Disable	Hardwired to 0 (This is for VGA compatible devices only)
6	Parity Error Response 1 - Device can report parity errors (using PERR#) 0 - Device ignores parity errors	Register (RESET# to 0; set to 1 if CRAM is to report parity errors)
7	Wait Cycle Enable 1 - Device does address/data stepping 0 - No address/data stepping	Hardwired to 0
8	System Error Enable 1 - Device can report address parity errors (SERR#) 0 - Disables use of SERR#	Register (RESET# to 0; set to 1 if CRAM is to address report parity errors)
9	Fast Back-to-Back Enable 1 - Can perform fast back-to-back transactions 0 - Disable	Hardwired to 0 (This is for PCI bus masters only)
15:10	Reserved by PCI	Hardwired to "000000"

Table A.8 CRAM PCI Command Registers

- **Status Register:** This tracks the status of PCI transactions on a PCI device. Any bit of the status register can be cleared by writing a 1 to it using a PCI configuration write cycle. See Table A.9.

Bit	Function	CRAM Implementation
4:0	Reserved by PCI	Hardwired to "00000"
5	66 MHz Capable 1 - Device is capable of running at 66 MHz 0 - Device can only run at 33 MHz	Hardwired to 0
6	UDF Supported 1 - Device supports User Defined Features 0 - Doesn't support UDFs	Hardwired to 0
7	Fast Back-to-Back Capable 1 - Supports fast back-to-back transactions 0 - Isn't capable	Hardwired to 1 (CRAM can handle fast back-to-back transactions)
8	Data Parity Reported 1 - Master reported the parity error 0 - Didn't report any parity error	Hardwired to 0 (For bus masters only)
10:9	Device Select (DEVSEL#) Timing 00 - Fast DEVSEL# timing 01 - Medium DEVSEL# timing 10 - Slow DEVSEL# timing 11 - Reserved	Hardwired to "01" (CRAM can claim a PCI transaction 2 clock cycles after FRAME# was asserted)
11	Signaled Target Abort 1 - Target terminated cycle with Target Abort 0 - Didn't	Hardwired to 0 (CRAM never uses a Target Abort; only a Disconnect and a Retry are used to terminate transactions)
12	Received Target Abort 1 - Transaction was terminated by a Target Abort 0 - Didn't	Hardwired to 0 (Only indicated by a bus master whose transaction was terminated)
13	Received Master Abort 1 - Transaction was terminated by a Master Abort 0 - Didn't	Hardwired to 0 (Again, for bus masters only)
14	Signaled System Error (SERR#) 1 - Device signaled an address parity error 0 - Did not	Register (RESET# to 0; set to 1 if CRAM detects and issues a system error)
15	Detected Parity Error 1 - Device detected a data parity error 0 - Didn't	Register (RESET# to 0; set to 1 if CRAM detects a data parity error)

Table A.9 CRAM PCI Status Register

- **Class Code register:** The class code is a 24-bit read-only register that identifies the function of the PCI device. The CRAM controller is a Memory Controller (bits 23:16 hardwired to 0x05), and it is not a RAM or Flash Memory Controller (bits 15:8 hardwired to 0x80). Like all PCI Memory Controllers, the CRAM controller has no specific programming interface (bits 7:0 hardwired to 0x00).
- **Base Address Register 0:** Base address registers allow units on a PCI device to be automatically allocated a memory or I/O address space. Memory is prefetchable if reads have no side effects on the con-

tents of the locations being read, a read always returns all four bytes regardless of the byte enable settings, posted and merged writes do not cause errors, and the memory is not cached by the host processor. Table A.10 describes CRAM implementation of the base address register 0.

Bit	Function	CRAM Implementation
0	Memory or I/O Space 0 - Memory; 1- I/O	Hardwired to 0 (All CRAM units are memory-mapped)
2:1	Memory Location 00 - Anywhere in 32-bit address space 01 - Below 1 MB 10 - Anywhere in 64-bit address space 11 - Reserved	Hardwired to "00" (Current CRAM controller is on a 32-bit PCI bus)
3	Prefetchable Bit 0 - Non-prefetchable 1 - Prefetchable	Hardwired to 1 (CRAM memory is prefetchable)
31:4	Base Address	23:4 is hardwired to 0x00000 (CRAM address space is 16 MBytes); 31:24 are registers (RESET# to 0; initialized by the host processor to assign the base address of the CRAM card

Table A.10 CRAM PCI Base Address Register 0

A.3.1 PCI Commands

Table A.11 shows all PCI commands and the ones supported by the CRAM controller interface unit.

CBE#[3:0]	PCI Command	Supported in CRAM
0000	Interrupt Acknowledge	No (Only for Host/PCI bridges)
0001	Special Cycle	No (Broadcast messages are ignored by the CRAM controller)
0010 0011	I/O read I/O Write	No (All resources are memory-mapped) No
0100-0101	Reserved	No
0110 0111	Memory Read Memory Write	Yes Yes
1000-1001	Reserved	No
1010 1011	Configuration Read Configuration Write	Yes Yes
1100	Memory Read Multiple	Yes
1101	Dual-Access Cycle	No (No 64-bit addressing supported)
1110 1111	Memory Read Line Memory Write and Invalidate	Yes Yes

Table A.11 PCI Command Types

Appendix B

CRAM System PCBs and Pinouts

B.1 ISA CRAM System

B.1.1 PCB Layout

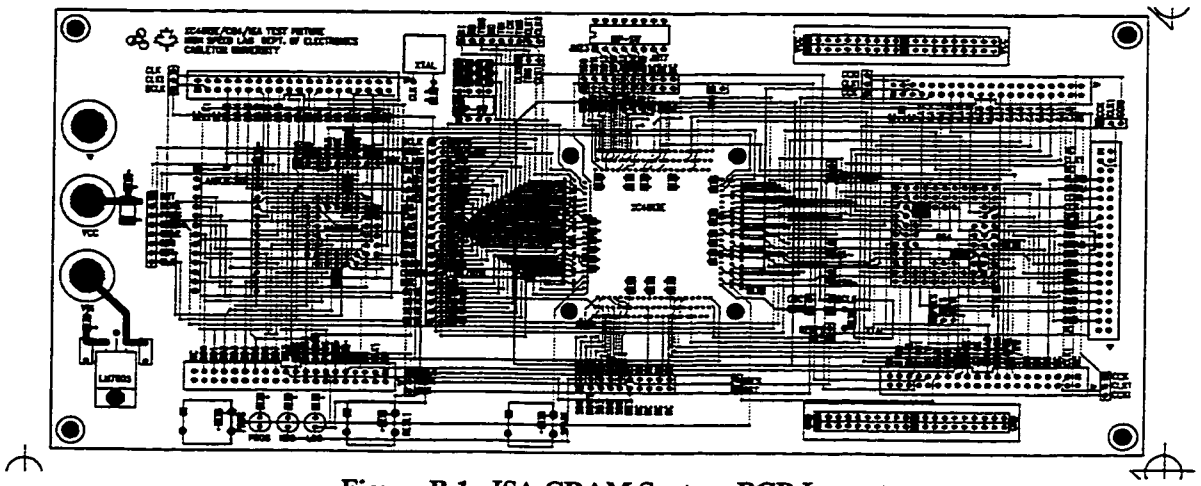

Figure B.1 ISA CRAM System PCB Layout

B.1.2 Pinout

CRAM Controller FPGA (Xilinx XC4013E-PQ240)

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
P1	GND	P27	BALE	P59	GND	P103	MCK	P151	GND	P205	IOB1
P2	BCLK	P28	SA<4>	P60	M0	P105	SLORI	P152	D<4>	P207	XCTRL
P4	MWTC	P29	GND	P61	VCC	P106	GND	P154	D<5>	P209	RSTB
P5	MRDC	P30	VCC	P62	M2	P108	SROLI	P156	D<6>	P211	GND
P6	LA<17>	P31	SA<5>	P63	SD<0>	P110	CCK	P159	D<7>	P212	VCC
P7	LA<18>	P32	SA<6>	P64	SD<1>	P117	A<10>	P161	VCC	P213	BTOUT
P8	LA<19>	P33	SA<7>	P65	NOWS	P119	GND	P162	A<0>	P222	VCC
P9	IRQ15	P34	SA<8>	P66	SD<2>	P120	DONE	P164	A<1>	P223	JMPR<17>
P10	LA<20>	P35	SA<9>	P67	SD<3>	P121	VCC	P166	GND	P224	JMPR<18>
P11	LA<21>	P36	SA<10>	P68	SD<4>	P122	PROGRAM	P167	A<2>	P225	JMPR<19>
P12	IRQ11	P38	SA<11>	P69	SD<5>	P123	CCKI	P169	A<3>	P226	JMPR<20>
P13	LA<22>	P39	SA<12>	P70	IRQ9	P126	A<11>	P171	A<4>	P227	GND
P14	GND	P40	VCC	P71	SD<6>	P128	A<12>	P173	A<5>	P228	JMPR<21>
P15	IRQ10	P41	SA<13>	P72	SD<7>	P130	A<13>	P175	A<6>	P229	JMPR<22>
P16	LA<23>	P42	SA<14>	P73	RESET	P132	A<14>	P176	A<7>	P230	JMPR<23>
P18	SBHE	P43	SA<15>	P75	GND	P134	A<15>	P177	DIN	P232	SD<15>
P19	VCC	P44	SA<16>	P80	VCC	P135	GND	P179	CCLK	P233	SD<14>
P20	M16	P45	GND	P89	INIT	P140	VCC	P180	VCC	P234	SD<13>
P21	OSC	P46	SA<17>	P90	VCC	P141	D<0>	P182	GND	P235	SD<12>
P23	SA<0>	P47	SA<18>	P91	GND	P144	D<1>	P192	A<8>	P236	SD<11>
P24	SA<1>	P48	SA<19>	P97	RAM	P146	D<2>	P194	A<9>	P237	SD<10>
P25	SA<2>	P51	CHRDY	P100	OPS	P148	D<3>	P196	GND	P238	SD<9>
P26	SA<3>	P58	M1	P101	VCC	P150	VCC	P201	VCC	P239	SD<8>
								P202	BTEN	P240	VCC

Download CPLD (MACH 210A-10JC)

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
P1	GND	P12	GND	P23	GND	P34	GND
P2	INIT	P13	CLK	P24	A<0>	P35	N/C
P3	DONE	P14	D<0>	P25	A<1>	P36	A<8>
P4	PROGRAM	P15	D<1>	P26	A<2>	P37	A<9>
P5	DIN	P16	D<2>	P27	A<3>	P38	A<10>
P6	CCLK	P17	D<3>	P28	A<4>	P39	A<11>
P7	FSM<0>	P18	D<4>	P29	A<5>	P40	A<12>
P8	FSM<1>	P19	D<5>	P30	A<6>	P41	A<13>
P9	N/C	P20	D<6>	P31	A<7>	P42	A<14>
P10	N/C	P21	D<7>	P32	N/C	P43	A<15>
P11	N/C	P22	VCC	P33	N/C	P44	VCC

Download EPROM (AM27C512)

PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL	PIN	SIGNAL
P1	A<15>	P8	A<2>	P15	D<3>	P22	OE (GND)
P2	A<12>	P9	A<1>	P16	D<4>	P23	A<11>
P3	A<7>	P10	A<0>	P17	D<5>	P24	A<9>
P4	A<6>	P11	D<0>	P18	D<6>	P25	A<8>
P5	A<5>	P12	D<1>	P19	D<7>	P26	A<13>
P6	A<4>	P13	D<2>	P20	CE (GND)	P27	A<14>
P7	A<3>	P14	GND	P21	A<10>	P28	VCC

Appendix C

CRAM Software Details

C.1 CRAM Assembly Language

C.1.1 CRAM Memory Variable Instructions

These are instructions that operate on CRAM memory variables (cvar). They are divided into the following groups:

- **3-Address Arithmetic and Logical Instructions:** In these instructions, AR0 is address of the destination operand, and AR1 and AR2 are the addresses of the two source operands. These instructions include ADD, SUB, XOR, OR, and AND.

  Syntax: *mnemonic* #AR0, #AR1, AR2;
  Example: ADD #24, #16, #8; // mem[24] ⇐ mem[16] + mem[8]

- **2-Address, 1 Immediate Value, Arithmetic and Logical Instructions:** AR0 is the address of destination operand, AR1 is the address of source operand, and *Imm* is an immediate integer value. Example are ADDI, SUBI, ISUB, XORI, ORI, and ANDI. ISUB is subtract from immediate value.

  Syntax: *mnemonic* #AR0, #AR1, #*Imm*;
  mnemonic #AR0, #*Imm*, #AR1; // syntax for ISUB
  Example: ADDI #24, #16, #10; // mem[24] ⇐ mem[16] + 10

- **2-Address Arithmetic and Logical Instructions:** AR0 and AR1 are addresses of destination and source operands, respectively. Examples are INC, DEC, NEG, and NOT.

  Syntax: *mnemonic* #AR0, #AR1;
  Example: INC #24, #16; // mem[24] ⇐ mem[16] + 1

- **Compare Instructions:** These instructions compare two operands at addresses AR1 and AR2. The result of the comparison is put in a cbool or cboolref variable at address AR0. Examples are LT, GT, EQ, LTE, GTE, and NEQ.

  Syntax: *mnemonic* #AR0, #AR1, #AR2;
  Example: LT #24, #16, #8; // mem[24] ⇐ (mem[16] < mem[8])

- **Compare Immediate Instructions:** These instructions compare an operand at address AR1 with an immediate integer, *Imm*. The result is put in a cbool or cboolref variable at address AR0. Examples are LTI, GTI, EQI, LTEI, GTEI, and NEQI.

  Syntax: *mnemonic* #AR0, #AR1, #*Imm*;
  Examples: EQI #8, #12, #16; // mem[8] ⇐ (mem[12] == 16)

- **Minimum/Maximum Search Instructions:** These instructions search for the minimum or maximum value of a cvar variable. The result is a cbool or cboolref (at mem[AR0]) that is set for a PE that contains a minimum or maximum element. The search instructions are MIN and MAX. In both cases, AR1 is the address of the $(n-1)^{th}$ bit of the search source operand (n is the number of bits of the operand).

  Syntax: *mnemonic* #AR0, #AR1;

Example: MAX #17, #16; // mem[17] ⇐ (is maximum element of mem[16])

- **Shift Instructions:** These instructions shift or rotate the positions of the elements of a cvar variable at address AR1, and puts the results at address AR0. If specified, the elements are shifted n positions, otherwise they are shifted one position. Shift instructions are SL, SR, RTL, and RTR. These instructions should not be confused with C++ shift operators (<< and >>) which shift the bits of the individual elements themselves.

  Syntax: *mnemonic* #AR0, #AR1 [, #*n*];
  Example: SL #16, #8, #5; // mem[16] ⇐ (mem[8] shifted left 5 columns)

- **Load Immediate Instruction:** The load immediate instruction, MVI, loads a constant integer, *Imm*, into all the elements of a cvar variable at address AR0.

  Syntax: MVI #AR0, #*Imm*;
  Example: MVI #16, #25; // (All elements of mem[16]) ⇐ 25

- **Copy Instruction:** The cvar copy instruction, MOV, copies the elements of a cvar at address AR1 into a cvar at address AR0. MOVR is a variation of MOV that does the copying in reverse starting with the last bit of the operands. Therefore for MOVR, AR0 and AR1 should point to the last bit of the operands.

  Syntax: MOV #AR0, #AR1; // mem[AR0] ⇐ mem[AR1]

- **Set/Clear Instructions:** MSET and MCLR sets (sets value to 0xFF...) and clears (sets value to 0x000...), respectively, the elements of a cvar at address AR0.

  Syntax: *mnemonic* #AR0;
  Example: MCLR #18; // mem[18] ⇐ 0

- **Copy PE Register Instructions:** The MOVPE instruction copies the contents of a PE register into a cbool or cboolref variable at address AR0. A variant of MOVPE instruction, the NMOVPE instruction, loads all the bits of a cvar variable with the contents of a PE register. Optionally, the source operand can be the inverse of a PE register by preceding the register name with an exclamation mark (!). Valid PE source registers are X, Y, and M.

  Syntax: *mnemonic* #AR0, [!]*PE_reg*;
  Example: MOVPE #16, !X; // mem[16] ⇐ $\overline{\text{PE.X}}$

- **Boolean Instructions:** All instructions involving cvar CRAM variables require that the correct value of the bit length of the operands be present in the WLEN register. This might require loading this value using the LDWLEN instruction before issuing a cvar instruction. But since instructions involving CRAM boolean variables (cbool and cboolref) are very common but always require a bit-length of one, a few of the cvar instructions are provided with their boolean equivalents. For these instructions, the bit-length of 1 has already been incorporated in their microinstructions. Therefore there is no need of setting up WLEN before issuing boolean instructions. The names of these instructions are made from their parent instructions, with a B added at the end. Boolean instructions include ANDB, ORB, NOTB, and MOVB. Their syntax is the same as their parent instructions.

C.1.2 CRAM PE Register Instructions

These are instructions that operate on PE registers. They include instructions to set or clear registers, and instructions to load PE registers from either other PE registers, or from cbool or cboolref variables. The following sections describe the PE register instructions.

- **Set/Clear Register:** CLR and ST instructions set the PE register to 0 and 1 respectively. Valid destination registers (*PE_reg*) are X, Y, and W.

Syntax: *mnemonic PE_reg [PE_reg] [PE_reg];*
Example: CLR X Y; // X[PE] ⇐ 0, Y[PE] ⇐ 0

- **Load from cvar:** The LDPE instruction loads PE registers with the contents of a cbool or cboolref variable at address AR0. Again, the destination register can be X, Y, or W. Optionally, you can load the registers with the inverse of the source operand by preceding the address register with an exclamation mark (!).

Syntax: LDPE *PE_reg [PE_reg] [PE_reg],* [!]#AR0;
Example: LDPE X, !#16; // X[PE] ⇐ $\overline{\text{mem[16]}}$

- **Register-to-Register Copy:** The TXPE instruction copies the contents of a PE register to another. The destination registers (*PE_dst*) can be X, Y, or W. The source register (*PE_src*) can be either X, Y, M, or their inverse (register name preceded by !).

Syntax: TXPE *PE_dst [PE_dst] [PE_dst],* [!]*PE_src;*
Example: TXPE X, !Y; // X[PE] ⇐ $\overline{\text{Y[PE]}}$

C.1.3 CRAM Controller Instructions

CRAM controller instructions include instructions to load, increment and decrement controller registers, as well as instructions for reading and writing data between CRAM and the controller. The following sections explain.

- **Load Register:** These are instructions to load specific registers of the controller (Table A.4 and Table A.5). *Imm* is the integer value to be loaded into the register. Load register instructions include LDCBA, LDIBA, LDWLEN, LDAX0, LDAX1, LDAX2, LDSIC, and SETINT.

Syntax: *mnemonic #Imm;*
Example: LDAX0 #5; // AX0 ⇐ 5

- **Increment/Decrement Register:** Registers AR0, AR1, and AR2 can be incremented and decremented by INCA and DECA instructions, respectively. Register CBA can also be incremented by INCA, but it can not be decremented.

Syntax: *mnemonic ctrl_reg;*
Example: INCA AR1; // AR1 ⇐ AR1 + 1

- **Memory Read/Write:** CRAM memory bytes starting at address [AR0] can be read and put into the controller read buffer at address [RIBA] using the READ instruction. Similarly, bytes can be transferred from the controller write buffer to the CRAM memory using the WRITE instruction. In both cases, like in any other instruction, the number of bytes to be transferred is defined by the contents of the WLEN register, which normally represents the number of bits for the cvar variable being accessed.

Syntax: READ AR0; // read_buffer[RIBA] ⇐ mem[AR0]
WRITE AR0; // mem[AR0] ⇐ write_buffer[WIBA]

- **Conditional Read-Bank:** The conditional read-bank instruction (RDBNK) is used to scan a cbool or cboolref variable to check if there is a 1 at any PE position. This is done by first reading the byte of the variable corresponding to the first eight PEs. If this byte contains a 1 on any of its eight bits, the byte is saved in the DTR register and the RDBNK instruction terminates. Otherwise the bank address register (CBA) is incremented to point to the byte corresponding to the next eight PEs, and the cycle is repeated. This is repeated until either a 1 is found or all the bits of the variable have been scanned. The results of the scan (CBA and DTR) can be used by the host processor to compute the index of a PE that yielded a true value to a search or comparison.

Syntax: RDBNK AR0; // DTR ⇐ mem[CBA:AR0], if mem[CBA:AR0] != 0

C.2 CRAM Microcode Assembly Language

C.2.1 Microroutine Format

Each microroutine begins with its name followed by a colon. This is the name that is used in the CRAM assembly language, e.g. ADD, LDPE. The name declaration is followed by one or more microinstructions. The format of the microroutine is illustrated below. The different types of microinstructions are described in subsequent sections.

Example: MOVPE: // instruction name; moves PE register into cbool
RST, EXT_TTOP;
CWRITE AR0, END;

C.2.2 PE Microinstructions

These instructions act on PE registers. They can be optionally mixed with some controller instructions.

Syntax: *mnemonic* [*PE_dst* [*PE_dst* ...]] [, [!]*PE_src* [[!]*PE_src* ...]]
[, EXT_TTOP] [, EXT_COP] [, *nai*] [, IF *cond*];

where,

PE_dst - is either Y, X, W, LX, RY, or B. LX is the X register of the left-neighbor PE, and hence may not be used simultaneously with X as the destination register. Similarly, RY is the Y register of the right-neighbor PE. RY and Y can not be used simultaneously. B denotes a bus-tie (Global-OR) operation on the results of a PE operation.

PE_src - is either X, Y, M. The inverse of the registers can be used by preceding the register name with the optional exclamation mark (!).

EXT_TTOP - shows that the PE truth table opcode (TTOP) for this microinstruction comes as OPR2 of the (macro)instruction. This TTOP is specified by the source operands of the instruction USE clause (Section C.2.5).

EXT_COP - shows that the PE control opcode (COP) for this microinstruction comes as OPR1 of the (macro)instruction. The actual value of COP is derived from the destination operands of the instruction USE clause (Section C.2.5).

nai - is any of the next address instructions (NAI) shown in Table A.1. Usually, the NEXTu instruction is not explicitly specified, but is assumed in all cases where no *nai* is specified. The keyword END can be (and is usually) used as an alias for NEXTI.

cond - is the condition if *nai* is a conditional instruction such as LOOPE, LOOPN, JMP, or JSR. Valid conditions are listed in Table A.2.

Example: ACARRY X, X !Y M, LOOPE, IF CNZ; // X ⇐ carry(X, !Y, M); loop if CNZ ≠ 0

C.2.3 Memory Access Microinstructions

The five CRAM memory access microinstructions are CREAD, CWRITE, READ, WRITE, and RDBNK. CREAD reads data from a CRAM memory row to the PEs M registers. CWRITE writes data from the PEs ALU outputs to a CRAM memory row. READ and WRITE transfers data between CRAM (pointed to by CBA:AX0:AR0) and the controller read/write buffers. RDBNK is used for conditionally reading a byte from CRAM as described in Appendix C.1.3.

Syntax: *mnemonic addr_reg* [, *cci*]] [, *nai*] [, IF *cond*];

where,

addr_reg - is either AR0, AR1, or AR2, and contains address of the CRAM memory row.

cci - is the controller instruction to increment or decrement the *addr_reg* or CBA after the microinstruction has finished executing. Valid instructions are INCA, DECA, and INCB.

nai and *cond* - are the next address instruction and its condition (see Section C.2.2).

Example: CREAD AR0, SETLP; // PE.M ⇐ mem[AR0]; LPR ⇐ μPC + 1

C.2.4 Controller Microinstructions

These are microinstructions for loading controller registers (Appendix C.1.3).

Syntax: *mnemonic* [, EXT_COP]] [, *nai*] [, IF *cond*];

where,

EXT_COP - denotes that the actual register to be loaded will be identified in the USE clause. Note that COP is used here since the actual controller load-register instruction is identified by microinstruction bits [15:8], which is also the position of PE COP.

nai and *cond* - are the next address instruction and its condition (see Section C.2.2).

Example: LDIBA, EXT_COP; // IBA ⇐ OPR0;

C.2.5 USE Clause

When an instruction microroutine has EXT_TTOP and/or EXT_COP in one of its microinstructions, another instruction can use this microroutine by simply specifying its specific PE source registers (TTOP) and/or PE destination registers or auxiliary controller instruction (COP). The USE clause is used to specify the full microinstruction that should substitute the microinstruction with EXT_TTOP/EXT_COP

Syntax: *instr_name*:

USE *prev_instr*: *new_μinstr*

where,

instr_name - name of instruction whose microroutine is being defined.

prev_instr - name of previously defined instruction whose microroutine should be used.

new_μinstr - the microinstruction that should be used to substitute the microinstruction in the *prev_instr* microroutine. Note that even though the full microinstruction is used, only its components that are relevant in forming TTOP and/or COP are used. The rest (NAI, COND, etc.) use the parameters specified in the original microinstruction.

Example: MSET:// instruction name; used to set the memory to FF..

USE MOVEPE: SET;// by using the MOVPE routine with TTOP of SET

Appendix D

Applications Source Code

D.1 Low-Level Image Processing

```
/*************************************************************/
/* Brightness Adjustment (Uniprocessor)                      */
/*  Brightens the image by adding a value to each pixel value */
/*************************************************************/

void brighten (IMAGE imageIn[IMAGE_SIZE][IMAGE_SIZE],
               IMAGE imageOut[IMAGE_SIZE][IMAGE_SIZE],
               unsigned char adjustment)
{
 int i, j;
 unsigned int new_val;

 for (i=0; i<IMAGE_SIZE; i++)
 {
  for (j=0; j<IMAGE_SIZE; j++)
  {
   new_val = imageIn[i][j] + adjustment;
   imageOut[i][j] = (new_val > 255) ? 255 : new_val;
  }
 }
}
```

```
/*************************************************************/
/* Brightness Adjustment (CRAM)                              */
/*  Brightens the image by adding a value to each pixel value */
/*************************************************************/

void brighten (CRAM_IMAGE& imageIn, CRAM_IMAGE& imageOut,
               unsigned char adjustment)
{
 imageOut = imageIn + adjustment;
 cif (imageOut.bit(8))
  imageOut = 255;
 cend;
}
```

```
/*************************************************************/
/* Pixel Averaging (Uniprocessor)                            */
/*  Noise reduction by averaging pixel values in 3x3 neighborhood */
/*************************************************************/

void average (IMAGE imageIn[IMAGE_SIZE][IMAGE_SIZE],
              IMAGE imageOut[IMAGE_SIZE][IMAGE_SIZE])
{
 int i, j;
 unsigned int ave;

 // leave the edges unchanged
 for (i=0; i<IMAGE_SIZE; i++)
 {
  imageOut[i][0] = imageIn[i][0];
  imageOut[i][IMAGE_SIZE-1] = imageIn[i][IMAGE_SIZE-1];
 }
 for (j=0; j<IMAGE_SIZE; j++)
 {
  imageOut[0][j] = imageIn[0][j];
  imageOut[IMAGE_SIZE-1][j] = imageIn[IMAGE_SIZE-1][j];
 }

 // average the other pixels (avoid edges!)
 for (i=1; i<(IMAGE_SIZE-1); i++)
 {
  for (j=1; j<(IMAGE_SIZE-1); j++)
  {
   ave = (imageIn[i-1][j-1] + imageIn[i-1][j] + imageIn[i-1][j+1] +
          imageIn[i][j-1] + imageIn[i][j] + imageIn[i][j+1] +
          imageIn[i+1][j-1] + imageIn[i+1][j] + imageIn[i+1][j+1])/9;
   imageOut[i][j] = ave;
  }
 }
}
```

```
/*************************************************************/
/* Pixel Averaging (CRAM)                                    */
/*  Noise reduction by averaging pixel values in 3x3 neighborhood */
/*************************************************************/

void average (CRAM_IMAGE& imageIn, CRAM_IMAGE& imageOut,
              cbool& not_edge_pes)
{
  // image shifted 1, IMAGE_SIZE-1, IMAGE_SIZE, IMAGE_SIZE+1
times
 cuchar shifted_image[4];

 cuint sum_left(10), sum_right(10), sum(11);

 // bring pixels from left and top, and add them
 PE.shiftr (shifted_image[0], imageIn, 1, 0);
 PE.shiftr (shifted_image[1], shifted_image[0], IMAGE_SIZE-2, 0);
 PE.shiftr (shifted_image[2], shifted_image[1], 1, 0);
 PE.shiftr (shifted_image[3], shifted_image[2], 1, 0);
 sum_left = (shifted_image[0] + shifted_image[1]) +
        (shifted_image[2] + shifted_image[3]);

 // bring pixels from right and bottom, and add them
 PE.shiftl (shifted_image[0], imageIn, 1, 0);
 PE.shiftl (shifted_image[1], shifted_image[0], IMAGE_SIZE-2, 0);
 PE.shiftl (shifted_image[2], shifted_image[1], 1, 0);
 PE.shiftl (shifted_image[3], shifted_image[2], 1, 0);
 sum_right = (shifted_image[0] + shifted_image[1]) +
        (shifted_image[2] + shifted_image[3]);

 // average
 sum = sum_left + sum_right;
 sum += imageIn;
 imageOut = sum/9;

 cif (!not_edge_pes)
  imageOut = imageIn;
 cend
}
```

```
/*************************************************************/
/* Edge Enhancement (Uniprocessor)                           */
/* Enhances edges by subtracting the Laplacian of a pixel from the pixel */
/*************************************************************/

void enhance_edges (IMAGE imageIn[IMAGE_SIZE][IMAGE_SIZE],
        IMAGE imageOut[IMAGE_SIZE][IMAGE_SIZE])
{
 int i, j;
 int diff;

 // leave the edges unchanged
 for (i=0; i<IMAGE_SIZE; i++)
 {
  imageOut[i][0] = imageIn[i][0];
  imageOut[i][IMAGE_SIZE-1] = imageIn[i][IMAGE_SIZE-1];
 }
 for (j=0; j<IMAGE_SIZE; j++)
 {
  imageOut[0][j] = imageIn[0][j];
  imageOut[IMAGE_SIZE-1][j] = imageIn[IMAGE_SIZE-1][j];
 }

 // subtract Laplacian from pixels (avoid edges!)
 for (i=1; i<(IMAGE_SIZE-1); i++)
 {
  for (j=1; j<(IMAGE_SIZE-1); j++)
  {
   diff = (imageIn[i][j] + 4*imageIn[i][j] - imageIn[i-1][j] -
           imageIn[i][j-1] - imageIn[i][j+1] - imageIn[i+1][j]);
   diff = abs(diff);
   imageOut[i][j] = (diff > 255) ? 255: diff;
  }
 }
}
```

```
/*************************************************************/
/* Edge Enhancement (CRAM)                                   */
/* Enhances edges by subtracting Laplacian of a pixel from the pixel */
/*************************************************************/

void enhance_edges (CRAM_IMAGE& imageIn,
      CRAM_IMAGE& imageOut, cbool& not_edge_pes)
{
 // image shifted 1, IMAGE_SIZE times
 cuchar shifted_image[2];
 cuint sum_left(9), sum_right(9), temp(12);
 cint laplacian (11);

 // bring pixels from left and top, and add them
 PE.shiftr (shifted_image[0], imageIn, 1, 0);
 PE.shiftr (shifted_image[1], shifted_image[0], IMAGE_SIZE-1, 0);
 sum_left = shifted_image[0] + shifted_image[1];

 // bring pixels from right and bottom, and add them
 PE.shiftl (shifted_image[0], imageIn, 1, 0);
 PE.shiftl (shifted_image[1], shifted_image[0], IMAGE_SIZE-1, 0);
 sum_right = shifted_image[0] + shifted_image[1];

 // find laplacian
 laplacian = 4*imageIn - (sum_right + sum_left);
 laplacian += imageIn;
 temp = abs(laplacian);

 cif(temp > 255)
  imageOut = 255;
 celse
  imageOut = temp;
 cend

  // leave edges unchanged
 cif (!not_edge_pes)
  imageOut = imageIn;
 cend
}
```

```
/*************************************************************/
/* Segmentation (Thresholding) (Uniprocessor)                */
/* Segments the image by converting it to binary image       */
/*************************************************************/

void tobinary (IMAGE imageIn[IMAGE_SIZE][IMAGE_SIZE],
          IMAGE imageOut[IMAGE_SIZE][IMAGE_SIZE],
          unsigned char threshold)
{
 int i, j;

 for (i=0; i<IMAGE_SIZE; i++)
 {
  for (j=0; j<IMAGE_SIZE; j++)
   imageOut[i][j] = (imageIn[i][j] >= threshold) ? 1 : 0;
 }
}
```

```
/*************************************************************/
/* Segmentation (Thresholding) (CRAM)                        */
/* Segments the image by converting it to binary image       */
/*************************************************************/

void tobinary (CRAM_IMAGE& imageIn, BINARY_IMAGE& imageOut,
          unsigned char threshold)
{
 if (threshold == 0)
  imageOut = (imageIn >= 0);
 else
  imageOut = (imageIn > (threshold-1));
}
```

```
/*************************************************************/
/* Segmentation (Multiple-Thresholding) (Uniprocessor)       */
/* Segments the image by using multiple thresholding         */
/*************************************************************/

void multi_threshold (IMAGE imageIn[IMAGE_SIZE][IMAGE_SIZE],
          IMAGE imageOut[IMAGE_SIZE][IMAGE_SIZE],
          unsigned char threshold[OBJECTS])
{
 int i, j, objm1;
 int obj; // objm1+1 - to reduce no of additions inside loop

 // for background
 for (i=0; i<IMAGE_SIZE; i++)
 {
  for (j=0; j<IMAGE_SIZE; j++)
   if (imageIn[i][j] < threshold[0]) imageOut[i][j] = 0;
 }

 // for objects 1 to N-1
```

```
/*************************************************************/
/* Segmentation (Multiple-Thresholding) (CRAM)               */
/* Segments the image by using multiple thresholding         */
/*************************************************************/

void multi_threshold (CRAM_IMAGE& imageIn, cuint& imageOut,
          unsigned char threshold[OBJECTS])
{
 int objm1, obj; // to reduce no of additions inside loop

 // for background
 cif (imageIn < threshold[0])
  imageOut = 0;
 cend

 // for objects 1 to N-1
 for (objm1=0, obj=1; objm1<(OBJECTS-1); objm1++,obj++)
 {
  cif ((imageIn >= threshold[objm1]) && (imageIn < threshold[obj]))
   imageOut = obj;
```

```
 for (objm1=0, obj=1; objm1<(OBJECTS-1); objm1++,obj++)
 {
  for (i=0; i<IMAGE_SIZE; i++)
  {
   for (j=0; j<IMAGE_SIZE; j++)
   {
     if ((imageIn[i][j] >= threshold[objm1]) &&
        (imageIn[i][j] < threshold[obj]))
       imageOut[i][j] = obj;
   }
  }
 }

 // for object N
 for (i=0; i<IMAGE_SIZE; i++)
 {
  for (j=0; j<IMAGE_SIZE; j++)
       if (imageIn[i][j] >= threshold[OBJECTS-1]) imageOut[i][j] =
OBJECTS;
  }
}
```

```
  cend
 }

 // for object N
 cif (imageIn >= threshold[OBJECTS-1])
  imageOut = OBJECTS;
 cend
}
```

D.2 Database Applications

```
/*************************************************************/
/* EQUAL-TO SEARCH (Uniprocessor)                            */
/*  An example of database equivalence searches: records with values  */
/*  equal to the search key are searched & replaced with a new value  */
/*************************************************************/

void equal_to_search (RECORD_TYPE records[RECORDS],
      RECORD_TYPE key, RECORD_TYPE new_value)
{
 for (int rec=0; rec<RECORDS; rec++)
  if (records[rec] == key) records[rec] = new_value;
}
```

```
/*************************************************************/
/* GREATER-THAN SEARCH (Uniprocessor)                        */
/*  An example of database threshold searches: records with values  */
/*  greater than the threshold value are searched and replaced  */
/*************************************************************/

void greater_than_search (RECORD_TYPE records[RECORDS],
      RECORD_TYPE threshold_value, RECORD_TYPE new_value)
{
 for (int rec=0; rec<RECORDS; rec++)
  if (records[rec] > threshold_value) records[rec] = new_value;
}
```

```
/*************************************************************/
/* BETWEEN-LIMITS SEARCH (Uniprocessor)                      */
/*  records with values between the two limits are searched & replaced  */
/*************************************************************/

void between_limits_search (RECORD_TYPE records[RECORDS],
        RECORD_TYPE low_limit, RECORD_TYPE high_limit,
        RECORD_TYPE new_value)
{
 for (int rec=0; rec<RECORDS; rec++)
  if ((records[rec] > low_limit) && (records[rec] < high_limit))
   records[rec] = new_value;
}
```

```
/*************************************************************/
/* EQUAL-TO SEARCH  (CRAM)                                   */
/*  An example of database equivalence searches: records with values  */
/*  equal to the search key are searched & replaced with a new value  */
/*************************************************************/

void equal_to_search (RECORD_TYPE& records, KEY_TYPE key,
      KEY_TYPE new_value)
{
 cif (records == key)
  records = new_value;
 cend
}
```

```
/*************************************************************/
/* GREATER-THAN SEARCH  (CRAM)                               */
/*  An example of database threshold searches: records with values  */
/*  greater than the threshold value are searched and replaced  */
/*************************************************************/

void greater_than_search (RECORD_TYPE& records,
      KEY_TYPE threshold_value,
      KEY_TYPE new_value)
{
 cif (records > threshold_value)
  records = new_value;
 cend
}
```

```
/*************************************************************/
/* BETWEEN-LIMITS SEARCH  (CRAM)                             */
/*  records with values between the two limits are searched & replaced  */
/*************************************************************/

void between_limits_search (RECORD_TYPE& records,
      KEY_TYPE low_limit,
      KEY_TYPE high_limit, KEY_TYPE new_value)
{
 cif ((records > low_limit) && (records < high_limit))
  records = new_value;
 cend
}
```

```
/*************************************************************/
/* MAXIMUM SEARCH  (Uniprocessor)                            */
/*  An example of database extreme searches: records with the maximum*/
/*  values are searched and replaced with a new value;       */
/*************************************************************/

void maximum_search (RECORD_TYPE records[RECORDS],
       RECORD_TYPE new_value)
{
 int rec;
 int prev_match[RECORDS];
 int tail = 0;
 RECORD_TYPE max_record = records[0];
 prev_match[0] = -1;

 for (rec=1; rec<RECORDS; rec++)
 {
  if (records[rec] > max_record)
  {
   max_record = records[rec];
   prev_match[rec] = -1;
   tail = rec;
  }
  else if (records[rec] == max_record)
  {
   prev_match[rec] = tail;
   tail = rec;
  }
 }

 // replace records with new value
 rec = tail;
 do
 {
  records[rec] = new_value;
  rec = prev_match[rec];
 } while (rec != -1);
}
```

```
/*************************************************************/
/* MAXIMUM SEARCH (CRAM)                                     */
/*  An example of database extreme searches: records with the maximum*/
/*  values are searched and replaced with a new value;       */
/*************************************************************/

void maximum_search (RECORD_TYPE& records,
       KEY_TYPE new_value)
{
 cif (ismax(records))
  records = new_value;
 cend
}
```

```
/*************************************************************/
/* LMS (Uniprocessor)                                        */
/*  multi-criteria records that best match the search keye are searched */
/*  using the least mean squared match and matching records are */
/*  replaced with new values                                 */
/*************************************************************/

void lms (RECORD_TYPE records[RECORDS][VEC_SIZE],
       RECORD_TYPE key[VEC_SIZE],
       RECORD_TYPE new_data[VEC_SIZE])
{
 int rec, vec;
 int field_error; // error between records and search key fields
 int rec_error; // accumulated squared errors for the vector

 int prev_match[RECORDS];
 int tail = 0;
 int min_error;
 prev_match[0] = -1;

 // case record 0
 rec_error = 0;
 for (vec=0; vec<VEC_SIZE; vec++)
 {
  field_error = records[0][vec] - key[vec];
  rec_error += (field_error*field_error);
 }
 min_error = rec_error;

 for (rec=1; rec<RECORDS; rec++)
 {
  rec_error = 0;
  for (vec=0; vec<VEC_SIZE; vec++)
  {
   field_error = records[rec][vec] - key[vec];
   rec_error += (field_error*field_error);
  }
```

```
/*************************************************************/
/* LMS (CRAM)                                                */
/*  multi-criteria records that best match the search keye are searched */
/*  using the least mean squared match and matching records are */
/*  replaced with new values                                 */
/*************************************************************/

void lms (RECORD_TYPE records[VEC_SIZE],
       KEY_TYPE key[VEC_SIZE],
       KEY_TYPE new_data[VEC_SIZE])
{
 // temporary CRAM variables
 cint field_error(17); // error between records and search key fields
 cuint rec_error(32);  // accumulated squared errors for the vector

 // find squared error between record and search key fields
 int vec;
 rec_error = 0;
 for (vec=0; vec<VEC_SIZE; vec++)
 {
  field_error = records[vec] - key[vec];
  rec_error += (field_error*field_error);
 }

 // update records with best LMS match
 cif (ismin(rec_error))
  for (vec=0; vec<VEC_SIZE; vec++) records[vec] = new_data[vec];
 cend
}
```

```
/*************************************************************/
/* Uniprocessor Motion Estimation                            */
/*  finds motion vectors of blocks between reference and current frames; */
/*  edged blocks are not coded (i.e. use Intra-frame information)  */
/*************************************************************/

void motion_estimation(
      unsigned char ref_frame[IMAGE_SIZE][IMAGE_SIZE],
      unsigned char cur_frame[IMAGE_SIZE][IMAGE_SIZE],
      unsigned int motion_vec[BLOCKS][2])
{
 int i, j, refi, refj;
 int blki, blkj, srchi, srchj;
 int vecx, vecy;
 int block_no = 0;
 int mae, min_mae;
 int MAX_MAE = 255*BLK_SIZE*BLK_SIZE;

 // do for each block at these positions
 for (blki=BLK_SIZE; blki<(IMAGE_SIZE-BLK_SIZE);
 blki+=BLK_SIZE)
 {
  for (blkj=BLK_SIZE; blkj<(IMAGE_SIZE-BLK_SIZE);
  blkj+=BLK_SIZE)
  {
   min_mae = MAX_MAE;

   // for each block there are (2*BLK_SIZE+1)^2 search posions
   for (srchi=(blki-BLK_SIZE); srchi<=(blki+BLK_SIZE); srchi++)
   {
     for (srchj=(blkj-BLK_SIZE); srchj<=(blkj+BLK_SIZE); srchj++)
     {
      mae = 0;

      // at each seach position, compare BLK_SIZExBLK_SIZE pixels
      for (i=blki, refi=srchi; i<(blki+BLK_SIZE); i++, refi++)
      {
       for (j=blkj, refj=srchj; j<(blkj+BLK_SIZE); j++, refj++)
       {
        mae += abs(cur_frame[i][j] - ref_frame[refi][refj]);
       }
      }

      // save if this has minimum distortion
      if (mae < min_mae)
      {
       min_mae = mae;
       vecx = srchj;
       vecy = srchi;
      }
     }
   }

   // save the motion vector
   motion_vec[block_no][0] = vecx;
   motion_vec[block_no][1] = vecy;
   block_no++;
  }
 }

}
```

```
/*************************************************************/
/* CRAM Motion Estimation                                    */
/*  finds motion vectors of blocks between reference and current frames; */
/*  edge blocks are not coded (i.e. use Intra-frame information);    */
/*  frames are divided into blocks, 1 block/PE                */
/*************************************************************/

void check_best_match (cuint& mae, cuint& min_mae,
          cuint& motion_vec, unsigned short* mvec);
void check_best_or_equal_match (cuint& mae, cuint& min_mae,
       cuint& motion_vec, unsigned short* mvec);

void motion_estimation(cuchar ref_blk[BLK_SIZE][BLK_SIZE],
          cuchar cur_blk[BLK_SIZE][BLK_SIZE],
          cuint& motion_vec)
{
 int i, j, srchi, srchj, refi, refj;
 unsigned short mvec = 0;

 cuchar temp0[BLK_SIZE][BLK_SIZE];
 cuchar temp1[BLK_SIZE][BLK_SIZE];
 cuchar temp2[BLK_SIZE][BLK_SIZE];
 cuchar temp3[BLK_SIZE][BLK_SIZE];

 cint diff(9);
 cuint mae(14), min_mae(14);

 min_mae = -1; // set to FFFF, max possible

 // shift blocks to get the 3 above and 1 to the left
 // temp3=left, temp2=top-right, temp1=top-centre, temp0=top-left
 for (i=0; i<BLK_SIZE; i++)
 {
  for (j=0; j<BLK_SIZE; j++)
  {
   PE.shiftl (temp3[i][j], ref_blk[i][j], 1, 0);
   PE.shiftl (temp2[i][j], temp3[i][j], BLOCK_COLS-2, 0);
   PE.shiftl (temp1[i][j], temp2[i][j], 1, 0);
   PE.shiftl (temp0[i][j], temp1[i][j], 1, 0);
  }
 }
 /*************************************************************/
 // use search positions in the top-left block
 for (srchi=0; srchi<BLK_SIZE; srchi++)
 {
  for (srchj=0; srchj<BLK_SIZE; srchj++)
  {
   // for each search position, compare all pixels
   mae = 0;
   for (i=0, refi=srchi; i<BLK_SIZE; i++, refi++)
   {
     for (j=0, refj=srchj; j<BLK_SIZE; j++, refj++)
     {
      if (refi < BLK_SIZE)
      {
       if (refj < BLK_SIZE) diff = cur_blk[i][j]-temp0[refi][refj];
       else diff = cur_blk[i][j]-temp1[refi][refj-BLK_SIZE];
      } else {
       if (refj < BLK_SIZE)
        diff = cur_blk[i][j]-temp3[refi-BLK_SIZE][refj];
       else diff = cur_blk[i][j]-
         ref_blk[refi-BLK_SIZE][refj-BLK_SIZE];
      }
      mae += abs(diff);
     }
   }
   check_best_match (mae, min_mae, motion_vec, &mvec);
  }
 }

 /*************************************************************/
 // use search positions in the top-centre block
 // top-left block no longer needed, transfer right block into temp0
 for (i=0; i<BLK_SIZE; i++)
 {
  for (j=0; j<BLK_SIZE; j++)
   PE.shiftr (temp0[i][j], ref_blk[i][j], 1, 0);
 }
```

```
 for (srchi=0; srchi<BLK_SIZE; srchi++)
 {
  for (srchj=0; srchj<BLK_SIZE; srchj++)
  {
   // for each search position, compare all pixels
   mae = 0;
   for (i=0, refi=srchi; i<BLK_SIZE; i++, refi++)
   {
      for (j=0, refj=srchj; j<BLK_SIZE; j++, refj++)
      {
       if (refi < BLK_SIZE)
       {
        if (refj < BLK_SIZE) diff = cur_blk[i][j]-temp1[refi][refj];
        else diff = cur_blk[i][j]-temp2[refi][refj-BLK_SIZE];
       } else {
        if (refj < BLK_SIZE)
          diff = cur_blk[i][j]-ref_blk[refi-BLK_SIZE][refj];
        else diff = cur_blk[i][j]-temp0[refi-BLK_SIZE][refj-BLK_SIZE];
       }
       mae += abs(diff);
      }
   }
   check_best_match (mae, min_mae, motion_vec, &mvec);
  }
 }
/*************************************************************/
 // use search positions in the top-right block
 for (srchi=0; srchi<BLK_SIZE; srchi++)
 {
  // for each search position, compare all pixels
  mae = 0;
  for (i=0, refi=srchi; i<BLK_SIZE; i++, refi++)
  {
   for (j=0; j<BLK_SIZE; j++)
   {
      if (refi < BLK_SIZE) diff = cur_blk[i][j]-temp2[refi][j];
      else diff = cur_blk[i][j]-temp0[refi-BLK_SIZE][j];
      mae += abs(diff);
   }
  }
  check_best_match (mae, min_mae, motion_vec, &mvec);
 }
/*************************************************************/
 // use search positions in the left block
 // top blocks no longer needed, transfer two botton blocks into temp1/2
 for (i=0; i<BLK_SIZE; i++)
 {
  for (j=0; j<BLK_SIZE; j++)
  {
   PE.shiftr (temp1[i][j], temp0[i][j], BLOCK_COLS-2, 0);
   PE.shiftr (temp2[i][j], temp1[i][j], 1, 0);
  }
 }

 for (srchi=0; srchi<BLK_SIZE; srchi++)
 {
  for (srchj=0; srchj<BLK_SIZE; srchj++)
  {
   // for each search position, compare all pixels
   mae = 0;
   for (i=0, refi=srchi; i<BLK_SIZE; i++, refi++)
   {
      for (j=0, refj=srchj; j<BLK_SIZE; j++, refj++)
      {
       if (refi < BLK_SIZE)
       {
        if (refj < BLK_SIZE) diff = cur_blk[i][j]-temp3[refi][refj];
        else diff = cur_blk[i][j]-ref_blk[refi][refj-BLK_SIZE];
       } else {
        if (refj < BLK_SIZE)
          diff = cur_blk[i][j]-temp1[refi-BLK_SIZE][refj];
        else diff = cur_blk[i][j]-temp2[refi-BLK_SIZE][refj-BLK_SIZE];
       }
       mae += abs(diff);
      }
   }
   check_best_match (mae, min_mae, motion_vec, &mvec);
  }
 }
/*************************************************************/
 // use search positions in the centre block
 // left block no longer needed, transfer botton-right block into temp3
 for (i=0; i<BLK_SIZE; i++)
 {
  for (j=0; j<BLK_SIZE; j++)
    PE.shiftr (temp3[i][j], temp2[i][j], 1, 0);
 }

 for (srchi=0; srchi<BLK_SIZE; srchi++)
 {
  for (srchj=0; srchj<BLK_SIZE; srchj++)
  {
   // for each search position, compare all pixels
   mae = 0;
   for (i=0, refi=srchi; i<BLK_SIZE; i++, refi++)
   {
      for (j=0, refj=srchj; j<BLK_SIZE; j++, refj++)
      {
       if (refi < BLK_SIZE)
       {
        if (refj < BLK_SIZE) diff = cur_blk[i][j]-ref_blk[refi][refj];
        else diff = cur_blk[i][j]-temp0[refi][refj-BLK_SIZE];
       } else {
        if (refj < BLK_SIZE)
          diff = cur_blk[i][j]-temp2[refi-BLK_SIZE][refj];
        else diff = cur_blk[i][j]-temp3[refi-BLK_SIZE][refj-BLK_SIZE];
       }
       mae += abs(diff);
      }
   }
   if ((srchi == 0) && (srchj == 0))
      check_best_or_equal_match (mae, min_mae, motion_vec, &mvec);
   else check_best_match (mae, min_mae, motion_vec, &mvec);
  }
 }
/*************************************************************/
 // use search positions in the right block
 for (srchi=0; srchi<BLK_SIZE; srchi++)
 {
  // for each search position, compare all pixels
  mae = 0;
  for (i=0, refi=srchi; i<BLK_SIZE; i++, refi++)
  {
   for (j=0; j<BLK_SIZE; j++)
   {
      if (refi < BLK_SIZE) diff = cur_blk[i][j]-temp0[refi][j];
      else diff = cur_blk[i][j]-temp3[refi-BLK_SIZE][j];
      mae += abs(diff);
   }
  }
  check_best_match (mae, min_mae, motion_vec, &mvec);
 }
/*************************************************************/
 // use search positions in the bottom-left block
 for (srchj=0; srchj<BLK_SIZE; srchj++)
 {
  // for each search position, compare all pixels
  mae = 0;
  for (i=0; i<BLK_SIZE; i++)
  {
   for (j=0, refj=srchj; j<BLK_SIZE; j++, refj++)
   {
      if (refj < BLK_SIZE) diff = cur_blk[i][j]-temp1[i][refj];
      else diff = cur_blk[i][j]-temp2[i][refj-BLK_SIZE];
      mae += abs(diff);
   }
  }
  check_best_match (mae, min_mae, motion_vec, &mvec);
 }
```

```
 /*************************************************************/
 // use search positions in the bottom-centre block
 for (srchj=0; srchj<BLK_SIZE; srchj++)
 {
  // for each search position, compare all pixels
  mae = 0;
  for (i=0; i<BLK_SIZE; i++)
  {
   for (j=0, refj=srchj; j<BLK_SIZE; j++, refj++)
   {
     if (refj < BLK_SIZE) diff = cur_blk[i][j]-temp2[i][refj];
     else diff = cur_blk[i][j]-temp3[i][refj-BLK_SIZE];
     mae += abs(diff);
   }
  }
  check_best_match (mae, min_mae, motion_vec, &mvec);
 }
 /*************************************************************/
 // use search positions in the bottom-right block
 mae = 0;
 for (i=0; i<BLK_SIZE; i++)
 {
  for (j=0; j<BLK_SIZE; j++)
  {
   diff = cur_blk[i][j]-temp3[i][j];
   mae += abs(diff);
  }
 }
 check_best_match (mae, min_mae, motion_vec, &mvec);
}
/*************************************************************/
void check_best_match (cuint& mae, cuint& min_mae,
            cuint& motion_vec, unsigned short* mvec)
{
 // check if this search position is the best match so far
 cif (mae < min_mae)
 {
  min_mae = mae;
  motion_vec = *mvec;
 }
 cend
 (*mvec)++;
}

void check_best_or_equal_match (cuint& mae, cuint& min_mae,
       cuint& motion_vec, unsigned short* mvec)
{
 // special case for no movement in case similar object elsewhere
 cif ((mae < min_mae) || (mae == min_mae))
 {
  min_mae = mae;
  motion_vec = *mvec;
 }
 cend
 (*mvec)++;
}
```