# A Computer Architecture for Fast Approximate Pattern Matching

Rickard E. Faith    Doug L. Hoffman

TR93-051
27 April 1993

# Contents

# List of Tables



# List of Figures



# List of Programs

# 1 Introduction

Several highly specialized pattern matching machines currently exist. These machines tend to be very expensive and are suitable only for a very narrow range of applications. Our goal is to design a pattern matching machine with the following characteristics:

- Support for many types of pattern matching applications, including text and biological sequence searching
- Inexpensive hardware that can be added to existing workstations
- At least one order of magnitude improvement in search speed over a sequential algorithm

# 2 Characteristics of the Application

## 2.1 Background

Recently, a class of algorithms have been proposed for rapid non-exact pattern matching [3, 1]. The ability of these algorithms to match strings containing arbitrary insertions, deletions, or substitutions make them applicable to a large number of problems, including text searching for information retrieval, biological sequence analysis (e.g., DNA and protein sequence matching), and protein structure analysis.

The "shift-and" algorithm, originally described by Baeza-Yates and Gonnet in 1989 [1] and subsequently modified by Wu and Manber [8], has characteristics which make it ideal for implementation in hardware. The algorithm, as actually implemented in the `agrep` program, is one of the fastest sequential pattern matching algorithms available. We have designed a coprocessor that will implement the fundamental operations necessary for the Wu and Manber algorithm in a highly parallel fashion.

## 2.2 Database Characteristics

The volume of data available for searching is increasing at an alarming rate. The Online Book Initiative, an Internet accessible e-text database, has doubled in size over the last year, and currently contains over 400 million characters of data. The GenBank database [4] has more than eighty thousand DNA and RNA sequences. The SWISS-PROT database [4] contains over twenty-six thousand protein sequences. These databases have been growing at an exponential rate with a doubling time between 18 and 24 months [2]. The GenBank database is expected to contain 3 billion characters by the year 2000. Given this volume of data, hardware-assisted pattern matching is not only desirable, but imperative. The sizes of some typical databases are shown in Table 1.

| Database Name | Size (in kilobytes) |
|---|---|
| Collins English Dictionary | 16603 |
| Merriam-Webster (Pocket) Dictionary | 3246 |
| King James Bible | 4794 |
| Complete Works of Shakespeare | 5219 |
| SWISS-PROT (protein database, v24.0) | 46662 |
| NCBI-GenBank (v75.0) | 329867 |
| Protein Information Resource (PIR) | 78718 |

Table 1: Typical Database Sizes

## 2.3 Pattern Length

The pattern length for various search requests may be quite different. For example, human-readable text searches are usually less than 64 characters long [8], whereas typical DNA sequence matches are 400 characters long.

The results of word-list analysis is shown in Table 2. These data indicate that,

| Word List | Items | Mean | Median | Standard Deviation |
|---|---|---|---|---|
| words | 25144 | 8.2 | 8 | 2.4 |
| web2 | 234936 | 10.6 | 10 | 2.9 |

Table 2: English Word Lengths

for searches based on single words, a maximum length of 14 characters is sufficient to accommodate over 98% of all requests. We have not, as yet, obtained reliable statistics for real-world text search requests. However, we feel that a generous estimate is that most single phrase requests will be less than 32–64 characters in length. In our design, we will show how a boolean expression consisting of several short words or phrases can be accommodated efficiently within the substantially longer pattern length necessary for biological sequence analysis.

The results of an analysis of an NCBI[1] log file for November 1992 is shown in Table 3. Assuming a normal distribution, the mean and standard deviations for these

[1]The National Center for Biological Information, part of the NIH, provides biological database search services via the Internet.

| Database | Items | Mean | Median | Standard Deviation |
|---|---|---|---|---|
| dbest | 2959 | 541 | 417 | 553 |
| emblu | 41 | 716 | 600 | 833 |
| epd | 15 | 103 | 20 | 187 |
| genbank | 2095 | 474 | 195 | 1216 |
| genpept | 193 | 306 | 115 | 554 |
| gpupdate | 110 | 460 | 260 | 861 |
| kabatnuc | 23 | 436 | 167 | 927 |
| nr | 21694 | 348 | 206 | 797 |
| nrdb | 13842 | 396 | 336 | 449 |
| pir | 226 | 368 | 150 | 1640 |
| swiss-prot | 113 | 202 | 152 | 172 |
| swissprot | 6983 | 571 | 378 | 767 |
| tfd | 92 | 275 | 140 | 511 |
| tfdaa | 731 | 265 | 93 | 440 |
| All | 49654 | 413 | 295 | 789 |

Table 3: NCBI Database Query Lengths

data suggest that a 2048 maximum character length limit would be sufficient for 84 to 98% of all requests. We note, however, that the median is often less than the mean, so the distribution is actually skewed to the left of the mean. This indicates that our estimates are conservative, and that 2048 characters would probably be sufficient for a higher percentage of requests.

In summary, the architecture should be flexible enough to process several short patterns in parallel. This would allow the construction of a text database query which involves AND and OR operations among several short patterns, as well as the simultaneous scan of a large DNA sequence in both the C-terminal and N-terminal directions.

## 2.4 Algorithm

Wu and Manber [8] succinctly describe the basic algorithm:

> Let $R$ be a bit array of size $m$ (the size of the pattern). We denote by $R_j$ the value of the array $R$ after the $j$ character of the text has been processed. The array $R_j$ contains information about all matches of prefixes of $P$ that end at $j$. More precisely, $R_j[i]$ =1 if the first $i$ characters of the pattern match exactly the last $i$ characters up to $j$ in the text (i.e., $p_1 p_{2 \cdots i} = t_{j-i+1} t_{j-i+2} \ldots t_j$). When we read $t_{j+1}$ we need to determine whether $t_{j+1}$ can extend any of the partial matches so far. For each $i$ such that $R_j[i] = 1$ we need to check whether $t_{j+1}$ is equal to $p_{i+1}$. If $R_j[i] = 0$ then there is no match up to $i$ and there cannot be a match up to $i + 1$. If $t_{j+1} = p_1$ then $R_{j+1}[1] = 1$. If $R_{j+1}[m] = 1$ then we have a complete match, starting at $j - m + 2$, and we output it. The transition from $R_j$ to $R_{j+1}$ can be summarized as follows:
>
> Initially, $R_0[k] = 0$ for all $k$, $1 \le k \le m$, and $R_0[0] = 1$, for all $j$, $0 \le j \le n$.
>
> $$R_{j+1}[i] = \begin{cases} 1 & \text{if } R_j[i-1] = 1 \text{ and } p_i = t_{j+1} \\ 0 & \text{otherwise} \end{cases}$$
>
> If $R_{j+1}[m] = 1$ then we output a match at $j - m + 2$;
>
> This transition, which we have to compute once for every text character, seems quite complicated. Other fast string-matching algorithms avoid the need to maintain the whole array. But, as observed by Baeza-Yates and Gonnet, this transition can be computed very fast in practice as follows. Let the alphabet be $\Sigma = s_1, s_2, \ldots, s_{|\Sigma|}$. For each character $s_i$ in the alphabet we construct a bit array $S_i$ of size $m$ such that $S_i[r] = 1$ if $p_r = s_i$. (It is sufficient to construct the $S$ arrays only for the characters that appear in the pattern.) In other words, $S_i$ denotes the indices in the pattern that contain $s_i$. It is easy to verify now that the transition from $R_j$ to $R_{j+1}$ amounts to no more than a *right shift* of $R_j$ and an AND operation with $S_i$, where $s_i = t_{j+1}$. So, each transition can be executed with only two simple arithmetic operations, a shift and an AND. We assume that the right shift fills the first position with a 1.

We have chosen this algorithm as the basis for our architecture because of the small number of fundamental operations required to implement the algorithm's inner program loop. For our architecture, $m = 2048$ and $|\Sigma| = 256$, allowing for a pattern size of up to 2048 characters and an alphabet size of up to 256 characters. The $R$ bit array becomes the 2048 bit `Accumulator`, and the $S$ bit arrays become the 256 by 2048 bit `BitVectorTable`. (The reasons for these specific choices are delineated later in this paper.)

Consider a search pattern of 1000 characters, a typical length for a polypeptide sequence. In a general purpose, 32 bit architecture, processing such a pattern would require breaking the logical and shift functions into 32 single-word sub-operations. Several additional machine instructions are required to chain the separate, single-word shifts into what is functionally a single 1000-bit register shift. There is additional overhead in the loop branching instructions. Each iteration of the inner loop also requires the fetching of two words of result (the word being operated on and its predecessor to provide carry-in information) and a word from one of the $|\Sigma|$ character position vectors. By providing 2048-bit parallel operations this inner loop is eliminated.

Our proposed architecture, utilizing bit vector operations and a large set of bit vector registers, both parallelizes the comparison algorithm and significantly reduces memory bandwidth requirements. The inner loop in the above example becomes a shift of $R$, a logical AND of $R$ and $S_j$, and a bit test of a bit in $R$.

Wu and Manber [8] explain an important enhancement to the algorithm which allows for approximate matching. Assume that $q$ arbitrary insertions or deletions are allowed in the pattern $P$. Then, the enhanced algorithm will shift $R$ by $q+1$ bits, and will test the last $q+1$ bits in $R$ to see if a match occurred. Our architecture implements this enhancement by allowing up to 63 arbitrary insertions or deletions. This requires that the `Accumulator` can be shifted by up to 64 bits at a time, and that the last 64 bits in the `Accumulator` can be logically OR'd together to test for a match.

We have also made several enhancements which make our architecture more general. Specifically, we divide the 2048-bit `Accumulator` into 32 64-bit "segments." By setting bits in the `SegmentRegister`, it is possible to treat each segment (or consecutive set of segments—a "logical segment" for this discussion) as a separate $R$ array for the shift and test operations. For example, by dividing the `Accumulator` into two 1024-bit "logical segments," we can simultaneously search the database for two unique patterns.[2] These multiple-pattern searches require the ability to test the last 64 bits of each "logical segment" to detect a match. We extended the architecture to support 64 insertions or deletions per "logical segment."

During this discussion, we have assumed that the search pattern, $P$, will fit nicely into a "logical segment." To simplify testing of the logical bits in the `Accumulator`, we will require that each search pattern be right justified within a "logical segment" and that any unused bits in the `Accumulator` be padded with 1's before a search is started. This restriction will make a test of the last $q+1$ bits sufficient even for short patterns.

[2] This nicely supports protein searches where both the forward and reverse sequence need to be checked against a database.

# 3 Principles of Operation

## 3.1 Highlights

### *History*

**Architects.** Rickard Faith and Doug Hoffman were the architects.

**Dates.** The RFDH-1 was designed during Dr. Fred Brooks' Advanced Computer Architecture course in the spring of 1993.

**Family tree.** To our knowledge the RFDH-1 has established no family tree. In fact, no example of the RFDH-1 has ever been built.

### *Noteworthy*

**Hardware support for parallel string comparison.** The RFDH-1 directly supports the "shift-and" variant of the string comparison algorithm of Baeza-Yates and Gonnet [1, 8] for string sizes of up to 2048 characters. This support is provided by a large set (256) of bit vector registers that are used as operands on a 2 kilobit accumulator. A single instruction executes a segmented $n$ bit right shift, an indexed bit vector AND, and a segmented bit extraction to a 32 bit result register.

**Large alphabet size** Hardware support is provided for alphabets of up to 256 characters in size. This allows the use of the full ASCII character set for text applications.

**Inexact matching.** Pattern strings containing wild cards and character classes (i.e., `[0-9]`) are supported transparently at full processing speed. Further, patterns that ignore certain characters [3] can be specified easily.

**Multiple search patterns.** The RFDH-1 is able to search for multiple patterns simultaneously. The total length of all patterns must be less than 2048 characters. This feature is ideal for text searches that contain conditional clauses and for searching biological sequences in the forward and reverse directions. Patterns are fit into subvectors that are multiples of 64 characters in length.

[3] For example, the pattern "foo" can be specified such that "a" is always ignored. Then "fooa," "faoao," and other variants will match the pattern.

### *Peculiarities*

**Designed as coprocessor.** The RFDH-1 was designed as a coprocessor for the MIPS R3000 RISC processor. The design reflects this dependency on the host processor in several ways: all control instructions are provided by the host, the contents of a single host register may be transferred to or from the coprocessor during a single instruction cycle, and the contents of a single coprocessor register may be transferred to or from memory in a single cycle.

The relevant "Coprocessor Instructions" supported by the MIPS R3000 [6, p. 3-11] are shown in Table 4.

**Context Switch Not Supported.** Because of the size of the `BitVectorTable`, context switches which involve the coprocessor are not realistic. In fact, no provision is made in the architecture to allow direct read access to the `BitVectorTable`.[4] This is consistent with the use of the coprocessor as a system resource which is managed by server software, much like a video display is managed by a graphical user interface server. The server can process all requests from user programs in batch mode, eliminating the need for more than one process to directly access the coprocessor.

[4] Actually, it is possible to read the contents of the `BitVectorTable` using the instruction set provided, but it would be too costly to be used for a context switch.

| Instruction | Format and Description |
| --- | --- |
| Load Word to Coprocessor | *LWCz rt,offset(base)*<br>Sign-extend 16-bit *offset* and add to *base* to form address. Load contents of addressed word into coprocessor register *rt* of coprocessor unit *z*. |
| Store Word from Coprocessor | *SWCz rt,offset(base)*<br>Sign-extend 16-bit *offset* and add to *base* to form address. Store contents of coprocessor register *rt* from coprocessor unit *z* at addressed memory word. |
| Move To Coprocessor | *MTCz rt,rd*<br>Move contents of CPU register *rt* into coprocessor register *rd* of coprocessor unit *z*. (*MTCz* is represented as `COP3.MT` in Program 3.) |
| Move From Coprocessor | *MFCz rt,rd*<br>Move contents of coprocessor register *rd* from coprocessor unit *z* to CPU register *rt*. (*MFCz* is represented as `COP3.MF` in Program 3.) |
| Move Control To Coprocessor | *CTCz rt,rd*<br>Move contents of CPU register *rt* into coprocessor register *rd* of coprocessor unit *z*. (*CTCz* is represented as `COP3.CT` in Program 3.) |
| Move Control From Coprocessor | *CFCz rt,rd*<br>Move contents of coprocessor register *rd* from coprocessor unit *z* to CPU register *rt*. (*CFCz* is represented as `COP3.CF` in Program 3.) |
| Coprocessor Operation | *COPz cofun*<br>Coprocessor *z* performs an operation. The state of the R3000 is not modified by a coprocessor operation. (*COPz* is represented as `COP3.CO` in Program 3.) |
| Branch on Coprocessor *z* True | *BCzT offset*<br>Compute a branch target address by adding address of instruction in the 16-bit *offset* (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor *z*'s condition line is true. (*BCzT* is represented as `COP3.BCT` in Program 3.) |
| Branch on Coprocessor *z* False | *BCzF offset*<br>Compute a branch target address by adding address of instruction in the 16-bit *offset* (shifted left two bits and sign-extended to 32 bits). Branch to the target address (with a delay of one instruction) if coprocessor *z*'s condition line is false. (*BCzF* is represented as `COP3.BCF` in Program 3.) |

Table 4: R3000 Coprocessor Instruction Summary

## 3.2 Machine Language

### *Language Level*

Because of the special-purpose nature of the machine, only 19 instructions are required. All instructions can be executed in one machine cycle. Most of the instructions are used to set bits or otherwise modify registers. One instruction is provided to perform the complex `SAT` (Shift, AND, Test) operation.

### *Unit System*

Bytes are 8 bits long, words are 32 bits long, and the bit vectors are 2048 bits long. A "segment" is 64 bits long. The `Accumulator` is divided into 32 segments.

```
 initiateRFDH1
⍝ initiation of the RFDH-1
 configureRFDH1
 formatRFDH1
 spaceRFDH1
 controlRFDH1

 configureRFDH1
⍝ configuration for the RFDH-1
⍝ memory capacity
 adrcap←radix*word

 controlRFDH
⍝ RFDH-1 control allocation
 instructionRFDH1
 indicatorRFDH1
```

```
 formatRFDH1
⍝ RFDH-1 information units
⍝ representation radix
 radix←2
⍝ information units
 byte←8
 word←32
 segments←32
 logsegments←radix⍟segments
 characters←256
 logcharacters←radix⍟characters
 accsize←2048
 logaccsize←radix⍟accsize
 segmentsize←accsize÷segments
 logsegmentsize←radix⍟segmentsize
⍝ address capacity
 adrcap←radix*word
```

Program 1: Basic parameters for the RFDH-1

***Spaces***

**Memory name-space.** The memory name space is shared with the host processor.

**Working store.** There are several special-purpose registers. The `RowRegister` and the `ColumnRegister` are used to address the 256 by 2048 bit `BitVectorTable`. All shift and logical AND operations are performed on the `Accumulator`, which can be accessed by the host processor as 64 32-bit words. The `ShiftRegister` specifies the length of the shift and the number of bits which will be tested by the `SAT` (Shift, AND, Test) instruction. The `TestRegister` receives the logical sum of the last `ShiftRegister` bits in each segment if the corresponding `SegmentRegister` bit is set to 1.

**Control store.** The control store is minimal: the `InstructionRegister` and the `StatusRegister`. The `InstructionRegister` is updated during each host processor instruction cycle, but only legal coprocessor instructions are executed. Instructions that are not coprocessor instructions are ignored and cause no change in the RFDH-1's state.

```
 spaceRFDH1
⍝ RFDH-1 spaces
⍝ host registers
 GeneralRegisters←?(32,word)⍴radix
⍝ coprocessor registers
 RowRegister←?byte⍴radix
 ColumnRegister←?logaccsize⍴radix
 BitVectorTable←?(characters,accsize)⍴radix
 ShiftRegister←?logsegmentsize⍴radix
 SegmentRegister←?word⍴radix
⍝ read-only coprocessor registers
 TestRegister←?word⍴radix
 StatusRegister←?word⍴radix      ⍝ similar to ind
⍝ coprocessor accumulator
 Accumulator←?accsize⍴radix
⍝ control store
 InstructionRegister←?word⍴radix
```

Program 2: Spaces of the RFDH-1

**Programming model.** The RFDH-1 is a coprocessor that has been designed to efficiently perform shifts (of up to 64-bits per shift) on a 2048-bit word, and to perform a logical AND operation on two 2048-bit words. The RFDH-1 can selectively test sets of bits in the 2048-bit word. Each set of bits are right-aligned on a 64-bit boundary and may contain up to 64-bits per set.

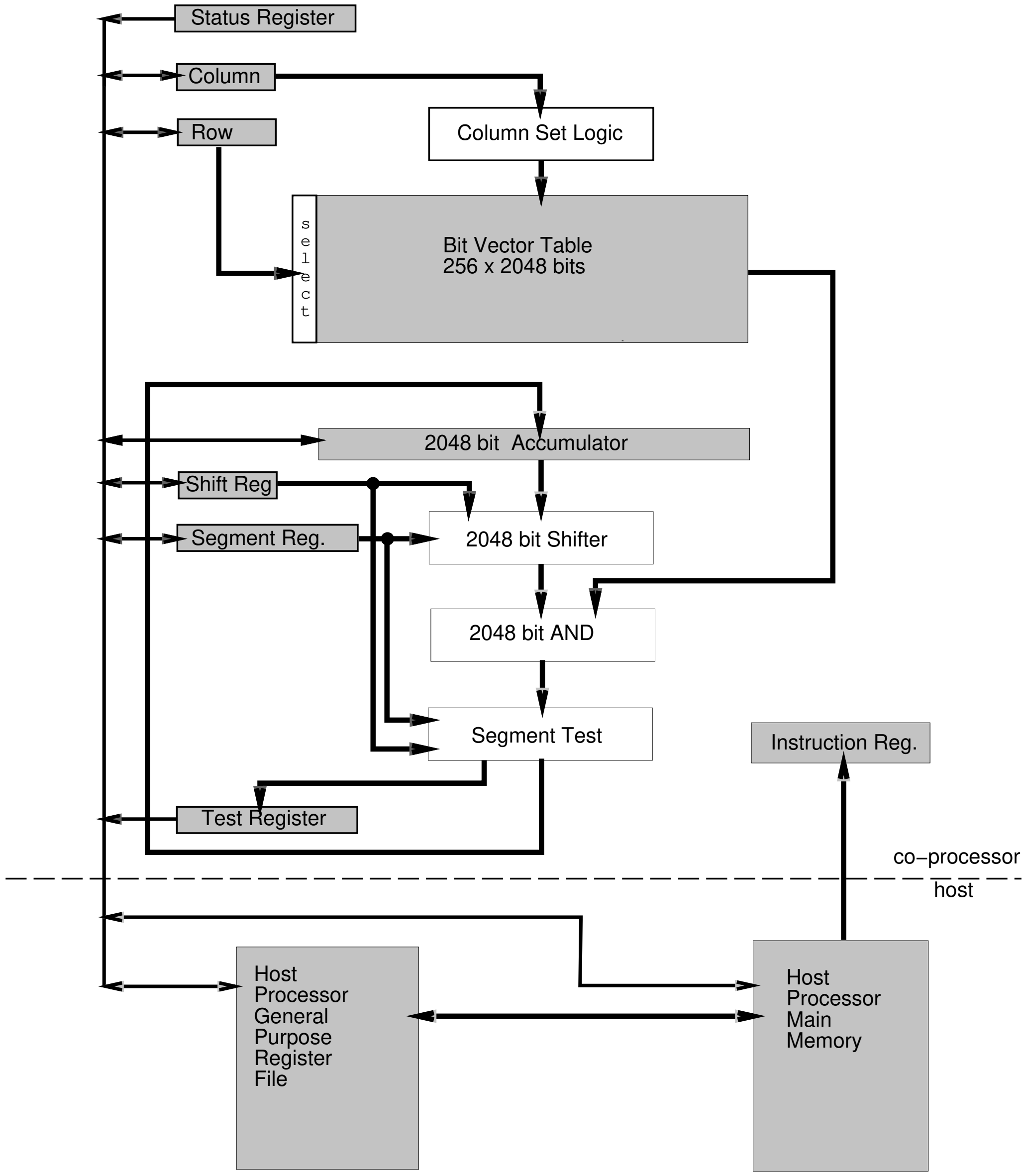


Figure 1: Programming model for the RFDH-1

### *Operation Specification*

Since the RFDH-1 is a coprocessor, the instruction set will be described in terms of the MIPS R3000 coprocessor instructions. Instructions are provided to set bits in the `BitVectorTable`, to access the registers, and to perform an atomic Shift, AND, Test operation. No provision is made to save the `BitVectorTable`, since the large amount of state in the coprocessor make context switches unreasonable. Since context switches are not supported, all of the instructions which read registers other than the `TestRegister` and the `StatusRegister` are not normally needed. These instructions are provided for hardware debugging and for verifying the current state during diagnostic testing.

| | | |
|---|---|---|
| | A *bit vector table initialization* | |
| a | STB | Set table bit |
| a | RTB | Reset table bit |
| a | STR | Set table row |
| a | RTR | Reset table row |
| a | STC | Set table column |
| a | RTC | Reset table column |
| a | STA | Set table all |
| a | RTA | Reset table all |
| | A *logic and shift* | |
| a | SAT | Shift, AND, test |
| | A *data handling* | |
| b | MTC | Move register to coprocessor |
| b | MFC | Move register from coprocessor |
| c | LTC | Load memory to coprocessor |
| c | SFC | Store from coprocessor to memory |
| a | SA | Set accumulator |
| a | RA | Reset accumulator |
| d | MTA | Move register to accumulator |
| d | MFA | Move register from accumulator |
| e | BTS | Branch on TestRegister set |
| e | BTR | Branch on TestRegister reset |

Table 5: Instruction list of the RFDH-1

### *Instruction Structure*

**Machine-language syntax.** The syntactic patterns have been divided into nine groups.

```
  syntaxRFDH1
a COP3.CO.CoprocessorFunction
b COP3.MT.Rg.Ra.11ρ0
b COP3.MF.Rg.Ra.11ρ0
c LWC3.base.Rb.offset
c SWC3.base.Rb.offset
d COP3.CT.Rg.Acc.10ρ0
d COP3.CF.Rg.Acc.10ρ0
e COP3.BCT.offset
e COP3.BCF.offset
```

Program 3: Instruction syntax for RFDH-1

**Instruction format.** Host processor instructions are 32 bits long, with opcodes specified by the MIPS R3000 coprocessor specification. These opcodes and their modifiers are mentioned here in symbolic form only.

```
a |COP3  |CO|CoprocessorFunction          |
   0      6  7                             32
b |COP3  |Mod  |Rg   |Ra   |00000000000   |
   0      6     11    16    21             32
c |Opcode|base |Rb   |offset              |
   0      6     11    16                   32
d |COP3  |Mod  |Rg   |Acc    |0000000000  |
   0      6     11    16      22           32
e |COP3  |Branch     |offset              |
   0      6           16                   32
```

```
 instructionRFDH1
⍝ RFDH-1 instruction allocation
⍝ operation specification
 COP3←0+⍳6
 Opcode←0+⍳6    ⍝ LWC3 or SWC3
 CO←6+⍳1
 Mod←6+⍳5       ⍝ modifier
 Branch←6+⍳10   ⍝ modifier
 CoprocessorFunction←7+⍳25
⍝ operand specification
 offset←16+⍳16 ⍝ memory
 base←6+⍳5      ⍝ host register
 Rg←11+⍳5       ⍝ host register
 Ra←16+⍳5       ⍝ coproc. reg.
 Rb←11+⍳5       ⍝ coproc. reg.
 Acc←16+⍳6      ⍝ accumulator
```

Program 4: Instruction allocation for RFDH-1

The `CoprocessorFunction` specifies one of 16 coprocessor opcodes and one of 16 possible coprocessor opcode modifiers. Currently, only 3 opcodes and 7 modifiers are used. Since these codes must fit within the 25 bit area reserved by the host for the coprocessor function specification, reserving several extra bits for the opcode and modifier was not unreasonable, and will allow policing in the current implementation which will be consistent with future enhancements.

| Instruction | Coprocessor Opcode | Modifier |
|---|---|---|
| STB (Set Table Bit) | 0.0.0.0 | 1.1.1.0 |
| RTB (Reset Table Bit) | 0.0.0.0 | 0.1.1.0 |
| STR (Set Table Row) | 0.0.0.0 | 1.1.0.0 |
| RTR (Reset Table Row) | 0.0.0.0 | 0.1.0.0 |
| STC (Set Table Column) | 0.0.0.0 | 1.0.1.0 |
| RTC (Reset Table Column) | 0.0.0.0 | 0.0.1.0 |
| STA (Set Table All) | 0.0.0.0 | 1.0.0.0 |
| RTA (Reset Table All) | 0.0.0.0 | 0.0.0.0 |
| SAT (Shift, AND, Test) | 1.0.0.0 | 0.0.0.0 |
| SA (Set Accumulator) | 0.1.0.0 | 1.0.0.0 |
| RA (Reset Accumulator) | 0.1.0.0 | 0.0.0.0 |

Table 6: `CoprocessorFunction` allocation for RFDH-1

**Status format.** At the start of each coprocessor instruction, the `StatusRegister` is cleared. When a coprocessor-specified exception occurs, a bit will be set in the `StatusRegister` which specifies the cause of the exception, and the appropriate R3000 hardware line will be signaled.

The `InvalidInstruction` bit will be set when the hardware detects a MIPS R3000 coprocessor instruction that is not used by the RFDH-1 or when the RFDH-1 cannot decode the `CoprocessorFunction`. Only instructions which are meant for coprocessor 3 will be fully decoded and tested for validity.

```
 which reportRFDH1 condition
⍝ RFDH-1 set bit in StatusRegister
 StatusRegister[which]←condition
 →IF condition
⍝ The RFDH-1 raises a hardware line
⍝ to signal the exception.
   raisecoprocessorexceptionline
 ENDIF:
```

```
 indicatorRFDH1
⍝ RFDH-1 indicator allocation
 RowOutOfRange←0
 ColumnOutOfRange←1
 ShiftOutOfRange←2
 InvalidRegister←3
 ReadOnlyRegister←4
 InvalidInstruction←5
```

Program 5: Indicator allocation and error handling for RFDH-1

## 3.3 Addressing

#### *Address Modification*

**Register addressing.** The host processor contains 32 general registers which are specified by the `Rg` instruction field. The general coprocessor registers are specified by the `Ra` and `Rb` instruction fields. Only three bits of these fields are used, although policing is provided for values which are out of range. The 64 32-bit segments in the `Accumulator` are specified by the `Acc` instruction field, which uses 7 bits.

**Address calculation.** The MIPS R3000 supports only one addressing mode: base register plus a signed 16-bit offset. The RFDH-1 load and store instructions use this format, but all address calculation is performed by the host processor.

## 3.4 Data

The R3000 defines a 32-bit word, a 16-bit half word, and an 8-bit byte [6]. The 16-bit half-word is not used by the RFDH-1. The R3000 byte ordering is configurable into either big-endian or little-endian ordering. The RFDH-1 description given here assumes a big-endian configuration.

#### *Character Strings*

**Character set and size.** The coprocessor assumes an 8-bit character set, but is independent of the character-set encoding.

#### *Logical*

**Logical formats.** The `Accumulator`, the `TestRegister`, and the `StatusRegister` are logical bit vectors. *True* is binary 1, and *False* is binary 0.

## 3.5 Operations

### *Bit Vector Table Initialization*

Data movement for the `BitVectorTable` uses special set and reset instructions. These instructions operate on the whole table, on a row or column of the table, or on a single bit in the table. The `BitVectorTable` *cannot* be read directly by the host processor.

```
 STB
⍝ RFDH-1 Set Table Bit
 BitVectorTable[RowRegister;ColumnRegister]←1

 RTB
⍝ RFDH-1 Reset Table Bit
 BitVectorTable[RowRegister;ColumnRegister]←0

 STR
⍝ RFDH-1 Set Table Row
 BitVectorTable[RowRegister;⍳accsize]←accsize⍴1

 RTR
⍝ RFDH-1 Reset Table Row
 BitVectorTable[RowRegister;⍳accsize]←accsize⍴0

 STC
⍝ RFDH-1 Set Table Column
 BitVectorTable[⍳characters;ColumnRegister]←characters⍴1

 RTC
⍝ RFDH-1 Reset Table Column
 BitVectorTable[⍳characters;ColumnRegister]←characters⍴0

 STA
⍝ RFDH-1 Set Table All
 BitVectorTable[⍳characters;⍳accsize]←(characters accsize)⍴1

 RTA
⍝ RFDH-1 Reset Table All
 BitVectorTable[⍳characters;⍳accsize]←(characters accsize)⍴0
```

Program 6: RFDH-1 Bit Vector Table Instructions

### *Logic and Shift*

The SAT (Shift, AND, Test) instruction is the only instruction that performs logic and shift operations. This single instruction forms the heart of the RFDH-1 instruction set.

**Segmented shift.** The segmented shift section shifts the Accumulator right by ShiftRegister bits. If a bit in ShiftRegister is set, then 1's are shifted into the corresponding segment.

**Logical AND operation.** After the segmented shift, the RowRegister is used to select a 2048-bit vector from the BitVectorTable. A logical AND operation is performed between this vector and the Accumulator.

**Logical OR operation.** If the bit in the SegmentRegister that corresponds to a segment is set, then the last ShiftRegister bits of that segment are logically OR'd together. The result for each segment is placed in the corresponding bit in TestRegister. If the bit in the SegmentRegister is reset, then the TestRegister bit is set to 0.

```
 SAT;words;shiftin;squareacc;index;hugeacc
⍝ RFDH-1 Shift, AND, Test
⍝ This code is a logical implementation of the SAT instruction.
⍝ The hardware implementation will perform the shift and test
⍝ functions with about logaccsize (or 11) layers of logic each
⍝ Intermediate results are stored in the Accumulator for clarity only.
⍝
⍝ Segmented shift
 squareacc←(segments+1) segmentsizeρAccumulator,(segmentsizeρ1)
 shiftin←(segments+1) segmentsizeρ(segmentsizeρ1),Accumulator
 shiftin←shiftin∨⍉segmentsize (segments+1)ρ(SegmentRegister,1)
 words←(¯radixcompi ShiftRegister)⌽shiftin,squareacc
 Accumulator←accsizeρwords[⍳segmentsize;segmentsize+⍳segmentsize]
⍝ Logical AND operation
 Accumulator←Accumulator∧BitVectorTable[radixcompi RowRegister;]
⍝ Logical OR operation
 index←(segmentsize-1)-radixcompi ShiftRegister
 hugeacc←segments accsizeρAccumulator
 TestRegister←SegmentRegister ∧∨/hugeacc[;index⍳radixcompi ShiftRegister]
```

Program 7: RFDH-1 Shift, AND, Test Instruction

### *Data Handling*

**General Registers** Data movement for the 6 general registers uses load, store, and move instructions. The `TestRegister` and the `StatusRegister` cannot be written by the host processor (the coprocessor will generate a `ReadOnlyRegister` exception if this is attempted).

With the exception of the `LTC` (Load Memory To Coprocessor) and the `SFC` (Store From Coprocessor To Memory) instructions, all of the coprocessor instructions are described below. The `LTC` and `SFC` instructions are similar to the `MTC` (Move Register To Coprocessor) and `MFC` (Move Register From Coprocessor) instructions, and are not described here because they are infrequently used and are provided in the instruction set for completeness and for conformance with the MIPS coprocessor description.

```
 MTC;tmp
⍝ RFDH-1 Move Register To Coprocessor
 →IF fld Ra<6
 THEN:
   →CASE(fld Ra)
   C0:tmp←(radixcompi GeneralRegisters[fld Rg;]
      RowOutOfRange reportRFDH1 tmp≥characters
      →OUT StatusRegister[RowOutOfRange]
      RowRegister←tmp
      →ENDCASE
   C1:tmp←GeneralRegisters[fld Rg;]
      ColumnOutOfRange reportRFDH1 tmp≥accsize
      →OUT StatusRegister[ColumnOutOfRange]
      ColumnRegister←tmp
      →ENDCASE
   C2:tmp←GeneralRegisters[fld Rg;]
      ShiftOutOfRange reportRFDH1 tmp≥segmentsize
      →OUT StatusRegister[ShiftOutOfRange]
      ShiftRegister←tmp
      →ENDCASE
   C3:SegmentRegister←GeneralRegisters[fld Rg;]
      →ENDCASE
   C4: C5:ReadOnlyRegister reportRFDH1 1
          →ENDCASE
   ENDCASE:
 ELSE:
   InvalidRegister reportRFDH1 1
 ENDIF:
```

Program 8: RFDH-1 Move Register To Coprocessor Instruction

```
MFC
⍝ RFDH-1 Move Register From Coprocessor
 →IF fld Ra<6
 THEN:
   →CASE(fld Ra)
   C0:GeneralRegisters[fld Rg;]←
           RowRegister,(word-byte)⍴0
      →ENDCASE
   C1:GeneralRegisters[fld Rg;]←
           ColumnRegister,(word-logaccsize)⍴0
      →ENDCASE
   C2:GeneralRegisters[fld Rg;]←
           ShiftRegister,(word-logsegmentsize)⍴0
      →ENDCASE
   C3:GeneralRegisters[fld Rg;]←SegmentRegister
      →ENDCASE
   C4:GeneralRegisters[fld Rg;]←TestRegister
      →ENDCASE
   C5:GeneralRegisters[fld Rg;]←StatusRegister
      →ENDCASE
   ENDCASE:
 ELSE:
   InvalidRegister reportRFDH1 1
 ENDIF:
```

Program 9: RFDH-1 Move Register From Coprocessor Instruction

**Accumulator data movement.** The 64 32-bit words in the `Accumulator` can be accessed by the host coprocessor using move instructions—load and store are not available. Note that although he MIPS R3000 reserves only 5 bits for the coprocessor register specification, the instruction which was used contains extra bits and is padded with zeros. The existing 5 bit field was expanded to provide a total of 6 bits for the coprocessor register specification. This is sufficient to specify one of the 64 "segments" in the `Accumulator`.

```
 CLA
⍝ RFDH-1 Clear Accumulator
 Accumulator←accsizeρ0

 MTA
⍝ RFDH-1 Move Register To Accumulator
 Accumulator[(32×fld Acc)+⍳32]←
      GeneralRegisters[fld Rg;]

 MFA
⍝ RFDH-1 Move Register From Accumulator
 GeneralRegisters[fld Rg;]←
      Accumulator[(32×fld Acc)+⍳32]
```

Program 10: RFDH-1 Accumulator Instructions

## 3.6 Instruction Sequencing

Instruction sequencing is performed by the host processor. When an instruction is loaded into the coprocessor `InstructionRegister`, that instruction is decoded. If the instruction is a valid RFDH-1 instruction, it is executed. Instructions that are not coprocessor instructions are ignored.

### *Decision*

**Conditional Branching** The MIPS R3000 coprocessor specification supports the notion of a conditional branch instruction that is dependent on the logical state of a coprocessor line. The RFDH-1 supports this special coprocessor branch instruction. The description given here for `BTS` (Branch On TestRegister Set) is logical only—the actual implementation would set a physical hardware line whenever the contents of the `TestRegister` are non-zero. Branch target address calculation is performed by the host processor, not by the RFDH-1.

```
 BTS
⍝ RFDH-1 Branch On TestRegister Set
 →IF ∨/TestRegister = 1
 THEN: iadr←iadr + 4×radixcompi inst[offset]
 BTR
⍝ RFDH-1 Branch On TestRegister Reset
 →IF ∨/TestRegister = 0
 THEN: iadr←iadr + 4×radixcompi inst[offset]
```

Program 11: RFDH-1 Branch On TestRegister Instructions

## 3.7 Supervision

There is no supervisory capability.

## 3.8 Input/Output

The RFDH-1 has no input or output capabilities. The host processor handles all I/O operations.

# 4 Design Rationale

The RFDH-1 is designed to function as a coprocessor to a MIPS R3000 RISC microprocessor. By choosing to design a coprocessor, rather than a full general purpose computer, the need to include control, I/O, and arithmetic operations is eliminated. The instructions needed for loop closure, address arithmetic, and other general operations are provided by the host processor.

Although this decision simplifies the design of the RFDH-1, it also imposes limitations that impact instruction format, memory addressing modes, register size, and instruction complexity. Because of these restrictions, the justification of design decisions for the RFDH-1 can be divided into two categories: decisions dictated by the MIPS coprocessor specification and decisions based on application requirements. The first category comprises host data access instructions; general purpose register size; exception handling; and the format of the R3000-supplied, coprocessor specific, instructions. The second category comprises the coprocessor's private working storage, register widths, and logical operations. Both of these categories will be examined in detail.

## 4.1 Host/Coprocessor Interface Design

### *Spaces*

**Memory name-space.** The memory name space is shared with the host processor. The MIPS R3000 provides an addressing range of 4 gigabytes (32 bits). All instructions and addresses are 32 bits in length. The load and store instructions generate addresses of this size.

**Working store.** The MIPS R3000 also provides 32 general-purpose 32-bit registers and a 32 bit program counter. Instructions exist to move words from these host registers to the coprocessor's registers. This implies that registers internal to the coprocessor must be addressable in units of 32 bits or less.

### *Instruction Structure*

**Machine-language syntax.** The structure of the machine instructions is dictated by the coprocessor instructions provided by the MIPS R3000. The instruction formats that are detailed in the previous section are taken directly from *MIPS RISC Architecture* by Gerry Kane [6]. Of the nine syntactic patterns, only type `a`, Coprocessor Operation, allows the specification of part of the instruction format by the coprocessor's designers. The instruction formats that conform to this syntactic pattern are those that execute exclusively on the coprocessor.

**Instruction format.** The nine syntactic patterns give rise to five instruction formats with a total of nineteen distinct instructions. The coprocessor-specific instructions (e.g., bit vector table and accumulator commands) are the only instructions for which we created the format. The Coprocessor Operation instruction leaves 25 bits for specifying the coprocessor instruction. We chose a single bit to indicate a set or reset operation and two bits to indicate single bit (`11`), row (`10`), column (`01`), or table (`00`) operation.

**Status format.** The status of the coprocessor is not directly available to the host processor. When an exception is raised in the coprocessor, it signals the host by asserting the external interrupt condition. Each of the size coprocessor exceptions is indicated by a bit in the coprocessor's `StatusRegister`. The host, upon trapping the external interrupt condition, must move the contents of the `StatusRegister` to one of its general purpose registers for interpretation.

***Address Modification***

**Register addressing.** Register addressing for the host processor's general purpose registers are defined by the R3000 architecture. The coprocessor instruction formats provide 5-bit fields for specifying source and destination register for `Load`, `Store`, `Move To`, and `Move From` instructions. This effectively restricts the coprocessor's register file to 32 registers though additional registers can be designated as control registers. These additional registers are accessed by the R3000 `Move Control To Coprocessor` and `Move Control From Coprocessor` instructions. These instructions allow the transfer of values between the R3000's general purpose registers and the coprocessor's control registers.

For the RFDH-1 `MTA` (Move Register To Accumulator) and `MFA` (Move Register From Accumulator) instructions, we decided to use the `Move Control` instructions to access the coprocessor's accumulator. The size of the accumulator (2048 bits) requires 64 32-bit control registers. This forces the use of a 6-bit control register address, exceeding the normal 5-bit register address field. The `Move Control` commands are the only R3000 coprocessor instructions that have expansion room for the additional register address bit, hence the designation of the accumulator space as "control registers" in the MIPS R3000 sense.

Note that there are no `Load` or `Store` instructions that operate on the coprocessor's control registers. This non-orthogonality present in the host processor specification causes an unavoidable difference between the accumulator registers and the other coprocessor registers.

**Address calculation.** Address calculation is defined by the host processor. All memory addresses are 32 bits in length. From the coprocessor's point of view, the only addresses that have to be dealt with are its own internal register addresses.

The `Load` and `Store` instructions calculate addresses as a 16-bit word displacement from a base address contained in one of the host's general purpose registers. This is a function of the host processor architecture.

## 4.2 Coprocessor Internal Design

The RFDH-1 is not intended to be the ultimate string search engine; much more powerful, and expensive, designs already exist. We intend the RFDH-1 to be an inexpensive add on to existing workstations that will provide a significant improvement in string search speed for very little cost. We are aiming for the "greatest improvement for least amount of cost" end of the market. In many ways, the design reflects that of a floating point coprocessor (FPU). We expect the RFDH-1 to enhance string search operations by roughly the same amount as an FPU enhances floating point operations. If fact, our initial investigation indicated a ten fold improvement in search speed over the same algorithm on an unassisted CPU. This is significantly better than the speedup provided by most FPUs and, under favorable conditions, even greater speedup is possible.

Specialized systems like BioSCAN and B-sys can provide greater performance for very specialized applications. These systems are massively parallel architectures that, if commercially available, would cost hundreds of thousands of dollars. Though the RFDH-1 will not provide the same level of performance as one of these fully custom systems, it should cost no more than an add-in FPU (<\$1000).

#### *Spaces*

Our initial concept for the RFDH-1 centered around the idea of a large register file containing the bit vectors used by the "shift-and" string matching algorithm. Running the "shift-and" algorithm with a 2048 character string on a 32-bit sequential processor requires the loading of 64 words of accumulator value and 64 words of bit vector value to process each character in the database being scanned. The 64 words of the resulting new accumulator value must be stored each step. By keeping the bit vectors and an accumulator vector in on-chip registers, memory bandwidth requirements could be greatly reduced.

The two most obvious questions that arise are: how big should the bit vectors be, and how many bit vector registers are needed? The answer to the first question depends on the size of the character strings you wish to search on. The answer to the second question depends on the alphabet size used. The reasoning behind our size selections is detailed in the following sections.

#### 4.2.1 Bit Vector Width

Selection of the bit vector width was made on the basis of statistical data from several sources. While we both have a strong interest in general purpose string matching for plain text databases, we also wanted the RFDH-1 to have uses in other areas—particularly biological sequence matching. As cited in the introductory section of this document, 98% of typical biosequence queries can be accommodated by a string length of 2048. Therefore, the size of the RFDH-1 bit vectors is 2048 bits. Though this is less than the 12992 characters accommodated by the BioSCAN custom processor [7] it is much larger than the 470 character capacity of the B-sys processor [5].

To allow multiple shorter test vectors to be processed in parallel, it was decided to segment the `Accumulator` into 32 64-bit segments. This fits nicely with the observation that most text queries are less that 64 characters in length [8]. The segments can be combined to form any combination of $64n$-bit "logical segments" that will fit into the 2048 bit `Accumulator`. Unfortunately, this organization of the `Accumulator` requires the extension of the R3000 5-bit register address to 6 bits. Fortunately the the `MTA` (Move Register To Accumulator) and `MFA` (Move Register From Accumulator) instruction formats have room for the extra bit.

We felt that this arrangement gave a maximum of flexibility while allowing the `SegmentRegister` and the `TestRegister` to be the same size as the host processor's general purpose registers (32 bits).

#### 4.2.2 Alphabet Size

The alphabet size dictates the width of the `RowRegister`. The alphabet size of 256 was selected because of the prevalence of 8-bit character codes. Experience with the BioSCAN custom processor has led us to believe that 32 characters are not enough for general purpose string matching. Even 64 characters is cramped for general string or phoneme matching.

Since ASCII is, essentially, a 7-bit code, an alphabet size of 128 would be a logical choice. Though it is possible that 128 would be sufficient for many applications, an 8-bit character code allows full ASCII representation as well as extensions used in European language character sets. There are other applications, besides full text matching, that can benefit from the large alphabet size. One such application is the comparison of protein structures by the matching of secondary structure components. One of the lessons learned by the BioSCAN project is that a large alphabet size opens the door to unanticipated applications.

Technologically, the move from 128 bits to 256 bits increases the on chip bit vector table size from 32 kilobytes to 64 kilobytes, well within current fabrication capabilities (many SRAM caches are twice this size). There do not seem to be any overwhelming reasons to extend the size above 256 characters, though this would certainly possible in future versions.

#### 4.2.3 The TestRegister

The size of the TestRegister, 32 bits, is a direct consequence of the number of segments in the accumulator. Thirty two segments seemed a more than sufficient number of items for conditional clauses. This size also allows the test register to fit into a single general purpose host register so that the result of each SAT (Shift, AND, Test) instruction can be transferred to the host in a single instruction cycle.

#### 4.2.4 Other Registers

The sizes of the other registers come from the ranges they need to address. The ColumnRegister must be able to address 2048 bit positions and so requires 11 bits. The RowRegister accommodates 256 character alphabet and is 8 bits in width.

The ShiftRegister reflects the number of insertions or deletions allowed in a comparison match. We decided that a rational limit for the shift amount would be the width of one of the accumulator segments, 64 bits. This justifies the ShiftRegister width of 6 bits.

### 4.3 Other Design Considerations

By allowing the accumulator to be segmented into combinations of 64-bit segments, the task of checking for matches is greatly complicated. The original algorithm tests the last $q+1$ bits of each sub-vector, where $q$ is the number of insertions or deletions. Since the sub-vectors are unlikely to exactly fill each segment and the sizes of the sub-vectors are expected to vary, our initial design contained a set of 32 "segment offset registers." These registers were to contain the lengths of each of the sub-vectors. The test operation would combine these sub-vector lengths with the shift amount to decide which bits from each segment to OR together. Aside from requiring extra register space and logic circuitry, this feature seemed improper.

We observed that, if you could load the accumulator with the proper initial bit pattern, all the sub-vectors could be right-justified within their respective segments. This would allow the testing of the last $q+1$ bits of each segment, eliminating the need for the segment offset registers. Since the initialization of the coprocessor is an infrequent operation, compared to the processing of data values, the additional work required to generate and load the initial accumulator pattern is not a major impact on the systems performance.

The use of the BTS (Branch On TestRegister Set) and BTR (Branch On TestRegister Reset) instructions was also a late addition to the architecture. Our initial design required that the `TestRegister`'s value be transferred to one of the host processor's general purpose registers for examination after each `SAT` (Shift, AND, Test) instruction. By using the `BTS` and `BTR` instructions, the additional register transfer is eliminated except when a pattern match is detected.

As can be seen from the RFDH-1's architecture, its design is a very parsimonious design. The coprocessor is mostly register space with some simple logical operations added to compute and test bit vector values. The interface to the host processor is not as clean as we would like. If the coprocessor had direct access to the host's general purpose registers, an even tighter design could be accommodated. Since the register and memory addressing options were dictated by the existing R3000 architecture, and since we wished to design a coprocessor for a real system, we feel that the architecture presented here is a good solution to the problem of accelerating pattern matching.

# 5 Sample Programs

## 5.1 Simple String Comparison

The following code will perform a simple string comparison for a 2048 character pattern. The positions of matching strings is the database are recorded in the result array. For this example, no insertions or deletions are allowed.

Please note that the purpose of this example is *not* to provide code that can be easily read and interpreted. The purpose is to compare the size and efficiency of the code with and without the RFDH-1 coprocessor in place.

The R3000 instruction mnemonics are given in a readable form. The need for branch trailing slot instructions has been ignored. The RFDH-1 coprocessor instructions are shown in boxes.

### 5.1.1 Unassisted MIPS R3000 Code

```
!  example program 1, string match without RFDH-1
!
!   test for 2k string exact match, 1000000 character database in memory.
!
    load        database,r16    ! r16 <- database address
    load        accumulator,r17 ! r17 <- accumulator address
    load        bitvectors,r19  ! r19 <- bit vector table address
    load        result,r20      ! r20 <- result array address
    load        1000000,r8      ! r8 <- database character count
    load        1,r25           ! a one for later use
    load        0x80000000,r26  ! and a mask value
!
!   initialize the accumulator
!
    load        63,r2           ! r2 <- 63
L0: add         r17,r2,r3       ! r3 <- r17[r2]
    store       r3,r0           ! accumulator[i] <- 0
    sub         r2,r25,r2       ! decrement i
    branchlt    r2,r0,L0        ! branch on less than 0
    move        r0,r7           ! clear loop index
!
!   outer loop, iterate database character count times.
!
L1: add         r16,r7,r2       ! r2 <- address database[n],
                                !      n = 0 to count-1
```

```
    loadbyte    r2,r1           ! r1 <- database[n]
    loadi       64,r18          ! r18 <- bit vector row size in words
    mult        r18,r1          ! lo <- row offset
    mflo        r18             ! r18 <- multiply result (lower 32 bits)
    add         r18,r19,r18     ! r18 <- address of bitvector[database[n]]
    load        63,r9           ! r9 <- inner loop count
!
!   inner loop to shift & AND each 32 bit piece of the accumulator
!
L2: add         r17,r9,r21      ! r21 <- r17[r9]
    load        r21,r2          ! r2 <- accumulator[j], j = 63 to 1 by -1
    shiftright  r2,r2,1         ! r2 <- r2 >> 1
    add         r18,r9,r22      ! r22 <- r18[r9]
    load        r22,r3          ! r3 <- bitvector[i][j]
    move        r21,r22         ! save old i value
    sub         r25,r21,r21     ! r21 <- (r21 - 1)
    load        r21,r4          ! r4 <- accumulator[j-1]
    andi        r4,r4,1         ! r4 <- (r4 AND 1)
    shiftleft   r4,r4,31        ! r4 <- (r4 << 31)
    or          r2,r4,r2        ! r2 <- (r2 OR r4)
    and         r2,r3,r2        ! r2 <- (r2 AND r3)
    store       r22,r2          ! accumulator[j] <- new value
    sub         r9,r1,r9        ! decrement loop counter
    branchgt    r9,r0,L2        ! branch on greater than 0
!
!   the last piece is a special case
!
    load        r17,r2          ! r2 <- accumulator[0]
    load        r18,r4          ! r4 <- bitvector[i][0]
    shiftright  r2,r2,1         ! r2 <- (r2 >> 1)
    or          r2,r26,r2       ! r2 <- (r2 OR 0x80000000)
    and         r2,r4,r2        ! r2 <- (r2 AND r4)
    store       r17,r2          ! accumulator[0] <- new value
!
!   test for a match
!
    load        r17,r2,63       ! r2 <- accumulator[63]
    andi        r2,r2,1         ! r2 <- (r2 AND 1)
    brancheq    r2,r0,L3        ! test for a match
    store       r20,r8          ! result[k] <- n
    addi        r20,r20,4       ! increment k
!
```

```
!   end of outer loop
!
L3: addi         r7,r7,1         ! increment n
    branchlt     r8,r7,L1        ! branch on less than character count
    end
```

### 5.1.2 MIPS R3000 Code Augmented for Use With the RFDH-1

The following code performs the same function as does the code for example program 1. However, example program 2 uses the RFDH-1 operations to enhance performance.

```
!  example program 2, string match with RFDH-1
!
!   test for 2k string exact match, 1000000 character database in memory.
!   RFDH-1 coprocessor instructions are shown in boxes
!
    load        database,r16    ! r16 <- database address
    load        result,r20      ! r20 <- result array address
    load        1000000,r8      ! r8 <- database character count
    load        1,r25           ! a one for later use
!
!   initialize the accumulator
!
    ResetAccumulator                                 ! set accumulator to 0
    MoveRegToCoprocessor    r25,SegmentRegister ! init segment register
    move                    r0,r7                ! r7 <- 0
!
!   processing loop, iterate database character count times.
!
L1: add                     r16,r7,r2       ! r2 <- address database[n],
                                            !      n = 0 to count-1
    loadbyte                r2,r1           ! r1 <- database[n]
    MoveRegToCoprocessor    r1,RowRegister  ! row register <- character
    ShiftAndTest                            ! Shift, AND, and Test
    BranchOnTestRegReset    L2              ! branch if no match
    store                   r20,r7          ! result[k] <- n
    addi                    r20,r20,4       ! increment k
!
!   end of processing loop
!
L2: addi        r7,r7,1         ! increment n
    branchlt    r8,r7,L1        ! branch on n = character count
    end
```

### 5.1.3 Comparison

A rough comparison of the performance of the RFDH-1 can be made by counting main loop instructions in example program 1 and example program 2. Example program 2, the unassisted MIPS R3000 code, has a main loop instruction count of $16 + 15n$ where

$$n = \left\lceil \frac{\text{StringLength}}{32} \right\rceil - 1$$

The inner loop for example program 2, the coprocessor assisted code, contains 7 instructions. Since matches are assumed to occur very infrequently, both of these instruction counts exclude the instructions needed to store result data. Ignoring any additional cycles needed by branch and load instructions a direct comparison of the inner loop instruction counts yields the data shown in Table 7.

| String length | Pgm 1 instructions | Pgm 2 instructions | Ratio |
|---:|---:|---:|---:|
| $< 32$ | 16 | 7 | 2.3 |
| 1024 | 496 | 7 | 70.9 |
| 2048 | 976 | 7 | 139.4 |

Table 7: Performance Comparison

Our design goal of at least one order of magnitude faster search time is met for strings of length greater than 128 characters or searches with four or more conditional clauses. Assuming 1.2 machine cycles per instruction, to compensate for branch and load instructions, a 33 MHz workstation with a RFDH-1 installed should be able to scan data at the rate of 3.9 million characters per second. For string sizes of up to 2048 characters, this is nearly twice as fast as the BioSCAN board.

## 5.2 Initialization

As can be seen from the code in example program 2, the actual search loop is quite simple and compact. The hardest part of programming the RFDH-1 is initializing the bit vector table and the accumulator. We have provided several instructions to simplify the loading of the bit vector table, but setting the initial bit mask in the accumulator is a little clumsy. We consider the amount of work required to perform the initialization to be of secondary importance since initialization takes place once for every several million iterations of the search loop.

### 5.2.1 Simple Patterns

Example program 3 will initialize the RFDH-1 to search for a single string of any length up to 2048 characters.

```
!  example program 3, with RFDH-1
!
!   initialize the bit vector table and accumulator from a single text string.
!   The length of the string is in r10 and a pointer to the start of the string
!   is in r11.
!
    addi        r10,r13,-1      ! r13 <- string length - 1
    loadi       2047,r2         ! r2 <- column pointer
!
!   initialize the bit vector table
!
    SetTableAll                 ! set bit vector table to all 1s
!
!   Scan the test string, setting table bits as we go
!
L1: add                     r11,r13,r14      ! r14 <- address of string[n]
    loadbyte                r14,r1           ! r1 <- string[n]
    MoveRegToCoprocessor    r1,RowRegister   ! select row
    MoveRegToCoprocessor    r2,ColumRegister ! select column
    SetTableBit                              ! set single bit
    addi                    r13,r13,-1       ! decrement n
    addi                    r2,r2,-1         ! decrement column pointer
    branchge                r13,r0,L1        ! branch i grater than or eq 0
!
!   Initialize the accumulator to the proper pattern of 1s and 0s. This is
!   accomplished in the following way: the entire accumulator is set to 1's,
```

```
!   the accumulator sub-registers that correspond to valid string data are
!   set to 0's, and any don't care bits at the boundry are set to 1's.
!
    SetAccumulator                  ! set accumulator to all 1s
!
!   reset the sub-registers corresponding to valid data to 0's
!
    shiftright               r2,r3,5  ! r3 <- r2 / 32
    loadi                    L2,r8    ! r8 <- address of L2:
    shiftleft                r3,r3,2  ! make it a word offset
    add                      r8,r3,r8 ! r8 <- address of L2 + word offset
    jump                     r8       ! jump to address in r8
L2: MoveRegToAccumulator     r0,r0    ! zero accumulator[0]
    MoveRegToAccumulator     r0,r1    ! zero accumulator[1]
    MoveRegToAccumulator     r0,r2    ! zero accumulator[2]
    MoveRegToAccumulator     r0,r3    ! zero accumulator[3]
             ⋮                 ⋮               ⋮
    MoveRegToAccumulator     r0,r61   ! zero accumulator[61]
    MoveRegToAccumulator     r0,r62   ! zero accumulator[62]
    MoveRegToAccumulator     r0,r63   ! zero accumulator[63]
!
! set the boundary sub-register bits.
!
    shiftleft           r3,r4,5       ! r4 <- r3 * 32
    sub                 r2,r4,r4      ! r4 <- don't care bits
                                      !     in boundary sub-reg.
    loadi               0x80000000,r5 ! r5 <- 0x80000000
    shiftrightarithvar  r4,r5,r5      ! r5 <- bit mask
    loadi               L3,r8         ! r8 <- address of L3
    shiftleft           r3,r3,1       ! make it a two word offset
    add                 r8,r3,r8      ! r8 <- address of L3 + word offset
    jump                r8            ! jump to address in r8
L3: MoveRegToAccumulator     r5,r0    ! load accumulator[0]
    jump                     L4       !
    MoveRegToAccumulator     r5,r1    ! load  accumulator[1]
    jump                     L4       !
    MoveRegToAccumulator     r5,r2    ! load  accumulator[2]
    jump                     L4       !
    MoveRegToAccumulator     r5,r3    ! load  accumulator[3]
    jump                     L4       !
             ⋮                 ⋮               ⋮
```

```
    MoveRegToAccumulator    r5,r61   ! load accumulator[61]
    jump                    L4       !
    MoveRegToAccumulator    r5,r62   ! load accumulator[62]
    jump                    L4       !
    MoveRegToAccumulator    r5,r63   ! load accumulator[63]
    jump                    L4       !
!
!   end of coprocessor initialization.
!
L4:
    end
```

The jumps with offset are needed to allow a computed selection of the accumulator registers. Though the tables of instructions needed for these jumps look large the execution time is very good. The large block of code beginning at L2 is an expanded conditional loop and the block beginning at L3 is a 64 entry case statement.

### 5.2.2 Patterns with Wild Cards

Given program 3 as a starting place, it is possible to add code to translate strings with special characters for wild carding. For example, if the character "?" represents a single position match with any character, the following code could be added to the initialization of the bit vector, replacing the SetTableBit instruction.

```
!
!   code to detect "?" - don't care character
!
    loadi           '?',r15    ! character to test with
    branchne        r14,r15,L6 ! branch if not "?"
    SetTableColumn             ! set entire table column to match
    jump            L8         ! jump to loop end
!
L6: SetTableBit                ! set single bit
L8:                            ! (continue)
```

### 5.2.3 Patterns with Character Classes

A another example, consider the use of the "ˆ" to indicate a match with any character other that the one following the current position. This is a bit more complicated as another character from the test string must be read without advancing the column pointer. This situation show the utility of being able to set an entire column to 1's and resetting a specific bit to 0.

```
!
!   code to detect "^" -  don't match next character
!
L6: loadi                  '^',r15           ! character to test with
    branchne               r14,r15,L7        ! branch if not "?"
    addi                   r13,r13,-1        ! decrement n
    add                    r11,r13,r14       ! r14 <- address of string[n]
    loadbyte               r14,r1            ! r1 <- string[n]
    MoveRegToCoprocessor   r1,RowRegister    ! select row
    MoveRegToCoprocessor   r2,ColumRegister  ! select column
    SetTableColumn                           ! set column to 1's
    ResetTableBit                            ! reset single bit
    jump                   L8                ! jump to loop end
!
L7: SetTableBit                              ! set single bit
L8:                                          ! (continue)
```

Other multiple character matching conditions can be easily implemented using the row, column, and bit instructions. A "match any number of anything" condition (the UNIX `egrep` ".*") can be implemented by dividing the string into "logical segments." To allow for a given number of insertions or deletions anywhere in the test string requires the string be cut and stacked before loading the bit vector table. The `ShiftRegister` is then set to one less than the number of permissible insertions and deletions.

The same basic search loop is used for all these search variations though some post search processing may be required. If a string is segmented into "logical segments" it must be verified that all "logical segments" found matches in the proper sequence for a full string match to be found. If a series of short strings are joined in a logical expression, such as "find the book abstracts with 'white whales' or 'albino whales' and 'dog food'," then post processing will be required to verify that the logical conditions have been satisfied within a particular database item.

# References

[1] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[2] C. Burks. How much sequence data the data banks will be processing in the near future. In R. R. Colwell, editor, *Biomolecular Data*, pages 17–26. New York: Oxford University Press, 1989.

[3] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992.

[4] M. Gribskov and J. Devereux, editors. *Sequence Analysis Primer*. New York: Stockton Press, 1991.

[5] R. P. Hughey and D. P. Lopresti. B-sys: A 470-processor programmable systolic array. *Proceedings of the International Conference on Parallel Processing*, pages 580–586, 1991.

[6] Gerry Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[7] R. K. Singh, S. G. Tell, C. T. White, D. L. Hoffman, V. L. Chi, and B. W. Erickson. A scalable systolic multiprocessor system for analysis of biological sequences. *Proceedings of the Symposium on Integrated Systems*, pages 167–182, 1993.

[8] Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.