Comical: A Forth-based Programming Language for Optimized Array Processor Programming

John E. Lecky
University of Vermont
Department of Electrical Engineering and Computer Science
Votey Building
Burlington, VT 05401

Abstract

A new programming language for directing highly parallel hardware systems such as array processors has been developed. This language is called COncurrent MICrocode Assembly Language, or **Comical**. A Comical assembler for a commercial array processor has been written in Forth. This combination allows array processor programs to be quickly developed and tested in the interactive Forth environment. Comical allows the software designer to take full advantage of the parallel processing paths of the array processor, maximizing execution speed. At the same time, programming in Comical is no more difficult than conventional assembly language programming, and Comical statements are descriptive and readable.

Introduction

Comical, the new language introduced in this paper, grew out of a development effort involving the use of **array processors** (APs) for solving engineering problems. The term "array processor," as used here, refers to a system of computing hardware that has been optimized for analyzing large arrays of data. An array processor offers considerable computational advantages in any discipline requiring the generation or analysis of large volumes of data, such as signal processing, image processing, statistical analysis and graphics work. The array processor's ability to cope elegantly with vast quantities of data arises from a carefully designed mixture of several processing elements which operate independently and simultaneously. The control and orchestration of these individual elements is generally effected through **microprogramming,** a term which refers to the fact that each array processor instruction is made up of several "mini-instructions," each providing a command for one of the individual processing elements. This compound instruction is commonly referred to as a **microinstruction,** or **microcode.**

An understanding of the array processor programming problem requires an understanding of array processor hardware. The set of processing elements typically included in an array processor is, therefore, discussed below.

Array Processor Hardware

One key AP element is the **sequencer.** The sequencer generates addresses for microcode memory, which is kept distinct from data memory. This technique follows the Modified Harvard Approach of maintaining separate program and data information buses. The sequencer selects the instruction that the AP will execute next while the AP is executing the current instruction. In this way, program flow control overhead is eliminated. Typical sequencers contain a microprogram counter (PC) which points to the next instruction. This PC can simply be incremented when executing sequential instructions, or can be more intelligently controlled by the sequencer to initiate jumps, calls, and loops. Sequencers usually have their own internal hardware stack to eliminate the time overhead normally associated with such functions. A counter internal to the sequencer can be used for controlling *do-loop* constructs. The sequencer will generally be in contact with the ALU status register, to allow conditional branching, like *if-then-else, begin-until,* or *while-repeat* constructs.

Another key AP element is the **Arithmetic Processor** (AMP). General-purpose AMPs will usually have a large set of internal *registers,* an extensive *arithmetic logic unit,* and

condition-testing logic to generate status signals. The AMP will be instructed with both the operation that it is to perform and a specific condition for which it is to check the result. The result of this status test, which might be of the form *not zero, negative,* or *carry,* for example, will be used by the sequencer to make conditional branching or calling decisions.

Since many AMPs do not provide multiplication hardware, a separate **multiplier** or **multiplier/accumulator** is often added. Even when the AMP *can* multiply, it is often advantageous to have the capability to perform a multiplication and some other ALU instruction simultaneously. Hardware multipliers generally have two input registers traditionally called X and Y, and produce their result in a third register. The microinstruction will control the flow of data to and from these registers. Since a multiplier always requires two operands, data memory is sometimes partitioned into two pieces with two separate data buses. In this way, the multiplier can be loaded with two operands simultaneously. This architecture is especially useful in signal and image processing.

Data memory, regardless of partitioning for the multiplier, will usually be addressed by one or more **Memory Address Registers,** or MARs. These are intelligent address generators that may be loaded with immediate values, or instructed to automatically increment or decrement. Efficient utilization of the MARs is crucial to the maximization of array processor efficiency.

All array processors have several different buses over which data is transferred. This arrangement allows for simultaneous data transfers of unrelated data between unrelated devices, or the simultaneous routing of a single piece of data to more than one device. To enable such operations, there must be **Bus Multiplexers** that select different sources to drive different buses. By careful control of the multiplexers, two unrelated array processor operations may often be combined to execute in a single cycle.

With these building blocks in mind, we may examine some array processor microinstruction design considerations.

The Microinstruction

The easiest way to design an instruction set for an AP is to simply string together each of the sets of bits required to instruct each element. Microinstruction designs following this format tend to be rather cumbersome, however, and are at the very least 48 bits wide.

In an effort to reduce the amount of required program memory and the width of the instruction bus, designers usually include some special bits in the microinstruction. These bits are not directly associated with a single instruction input on a single processing element, but rather are multiplexed to two or more elements.

The multiplexed bits can only be interpreted by examining the settings of some other bits in the microinstruction. They may be referred to, therefore, as **contextual** bits. Contextual bits are included in the instruction sets of most microprocessors, and so it comes as no surprise that they are found in array processors as well. Without contextual bits, microinstructions would grow to be unmanageably wide.

It will sometimes be physically impossible for the array processor to perform a particular operation because of some timing, busing, or other hardware constraints. Often these physical limitations may be exploited by the designer in defining the contextual bits. Other times, however, the designer will have no choice but to impose some **unnatural constraints.** Faced with a 51-bit microinstruction, for example, a designer will be tempted to somehow squeeze the 51-bits into a 48-bit space, eliminating innumerable tri-state buffers and latches. This squeezing can be accomplished only by throwing away some of the generality of the array processor design. The loss of generality may be judged unimportant when compared with the reduction in hardware complexity.

Unnatural constraints make some otherwise logical parallel operations impossible. They usually create some kind of interaction between seemingly unrelated elements. A typical example

would be some form of restriction on a particular AMP instruction during AP cycles in which the sequencer is branching to a subroutine. Constraints such as these are difficult for the programmer to remember, and grossly complicate high-level language design accordingly.

A New Programming Language

Traditional array processor programming languages fall into two categories: **meta-assemblers,** which interpret cumbersome and complex syntaxes, and **vectorizing compilers,** which can produce horrendously inefficient machine code. Meta assemblers usually offer reasonably independent control of the array processor, however, while vectorizing compilers allow programs to be written in a quite readable form. A new COncurrent MICrocode Assembly Language, **Comical,** offers both of these attractive features.

Comical is based on the principle that each processing element should be instructed independently by the programmer. Each element has its own complete instruction set. An AP instruction is constructed from a list of these individual sub-instructions; if an element is unneeded during a certain AP cycle, however, it may be left out. The sub-instructions may be specified in any order, improving readability.

Any interdependence between elements due to hardware limitations or contextual bits may be monitored inside the Comical assembler. This removes the burden of remembering unnatural constraints from the programmer or from a higher-level compiler designed to generate Comical statements. The contextual bit problem degenerates into merely keeping a record of whether two sub-instructions attempt to set a given microinstruction bit to conflicting values.

Comical statements are thus highly readable, but still maintain a one-to-one correspondence with array processor operations. Such one-to-one correspondence is critical to the success of code optimization.

To concisely illustrate Comical's primary features, a simple AP operation is programmed below. This operation could be part of a larger program. The AP is capable of performing the entire operation in one cycle. In English, the operation may be described as follows:

> Fetch a data word from memory and store it into both of the multiplier registers, preparing to compute a square. Have the AMP subtract the word from its accumulator, storing the result back into the accumulator. Finally, set the MAR equal to the same data. If the result of the last instruction executed by the AMP was negative, repeat the operation again.

Below is a high-level vectorizing compiler version of the routine. In this language, the operation consumes four AP cycles.

```
again   xy = m          ; get data from memory and store into
                        ;    both multiplier registers
        A = A-m         ; subtract data from the accumulator
        a = m           ; set the mar equal to the data
        br(n)again      ; branch back to 'again' if negative
```

Below, the same operation is programmed using a meta-assembler. Because of the syntax constraints on the assembler, the operation still requires three machine cycles. Note the no-ops (NOOPs) that must be passed to the Arithmetic Processor, since the AMP only performs one operation. Also, the sequencer is only passed one instruction (H#37).

```
AGAIN:  NOOP        , ,MXY,000                          ; XY = M
        TONR        , ,TODA,SUBR,NRA,MR,000             ; A = A-M
        NOOP        H#37 , ,MRA,AGAIN                   ; a = M, BR(N)AGAIN
```

Now, the operation is programmed in Comical. The instruction is the sole AP command in an AP routine called "example." The routine now requires just one AP cycle, as indicated by the

single semicolon signifying the end of a microinstruction. This example demonstrates the full power of a Comical-based concurrent programming environment.

```
:apb example
: again   mant ->idb ->xy        | place memory contents on internal data bus
                                 |    latch into both multiplier regs
            idb ->ob  ldmar      | connect data to outbus and load into mar
            acc d acc subr       | acc d - acc !
          again n ?jmp ;         | jump if negative to again
;apb
```

A Comical Assembler in Forth

With a Forth-based system containing an array processor, it is possible to write Forth code, host assembly-language code, and array processor code interchangeably on the same screen. In this environment, Comical can be written, assembled, debugged and integrated into complete systems especially quickly. A fifty-line Comical routine assembles and is ready to execute in about two seconds on an 8086-based Forth system. A similar program written in meta-assembler would take at least three minutes to assemble alone on similar hardware. High-level code would take correspondingly longer. The development time advantages of Comical are derived from the extensible Forth dictionary, which allows an interpreting assembler to be written.

The arguments necessary for each member of the individual instruction sets written for each processing element are simple constants available on the stack when the instruction is encountered. The instruction itself is a Forth word that immediately generates the necessary microcode. The appropriate number of arguments are converted into a single, unique number by a hashing routine. This number is then used as an argument into individual "switch" tables to determine the necessary microcode. Any illegal combinations are thus detected immediately and can be reported precisely. The Forth-based Comical assembler is therefore not only fast and completely general, but provides an error-handling facility which displays exactly where the error is and describes its nature. This feature rivals the error-handling found in conventional assemblers.

A small bit of Forth trickery allows forward references to statement labels. Thus, it is possible to jump forward to statements that have not even been assembled yet. The branch addresses are automatically filled in when the desired label is eventually encountered.

Closing

In summary, the **Comical** programming philosophy offers great advantages over conventional array processor programming techniques. When combined with a Forth-based system, a truly interactive AP programming environment is created. The author is presently using such a system to develop solutions to a variety of image processing problems, as well as some computationally difficult VLSI CAD optimization problems. With programming turn-around times cut from several minutes to less than a second, progress is both predictable and encouraging.

As the price of commercially-available array processors dips toward the thousand-dollar mark, with board-level products available even for PCs, the range of applications for which APs will be feasible will steadily expand. The primary development features of Forth are the elimination of the program loading concept, and the distribution of the compilation and assembly tasks, to the point of making them essentially transparent. These features, recognized for their utility by many designers already, will become increasingly valuable as computational hardware becomes more complex. With massively parallel hardware systems, distributed assembly may be the only viable option.

Forth is one of the few languages that demonstrates a capability for the effective implementation of arbitrary assemblers of enormous complexity. And with custom array processor chips appearing in the marketplace already, such assemblers will become ever more commonplace.