APL as a development tool for special-purpose processors

by SAMUEL LEVY, RICHARD H. DOYLE, and RALPH M. HELLER

IBM Corporation
Gaithersburg, Maryland

INTRODUCTION

APL, used as a programming language has been implemented in computing systems to provide time-sharing with remote terminals. Since the operative principles guiding the design of APL include simplicity and practicality of application, APL is extremely effective as a development aid for special purpose processors.

This paper describes two unrelated processor design projects which were significantly aided through the utilization of APL. The paper will emphasize the common features of many processor development tasks and show how APL may be applied advantageously.

The first example concerns an APL model which assisted in the conceptual development and evaluation of arithmetic algorithms for an associative processor whose arithmetic operations are based on residue arithmetic computations. The second example is an APL simulation of a processor with subsequent use of this simulation to develop object code. Both examples, APL provided timely and meaningful assistance.

It is not within the scope of this paper to compare APL with discrete simulation languages. The GPSS and SIMSCRIPT family of languages as well as APL program packages such as SUPERMOD provide far more detailed and comprehensive analyses than we are addressing in this paper. Methods and advantages of using APL specifically as an interactive simulator generating system for small computers are discussed in Reference 10.

Some of the basic features of APL which make it attractive for our purposes are as follows: A repertoire of application-oriented APL functions can be built incrementally, with interactive testing performed at each stage of development. At first, these functions can be quite primitive, such as representing the behavior of a single processor component (e.g., a shift register). Later, as a hierarchy of functions are built, testing will become more complete. The advantage of interactive development is that interspersed development and testing can proceed with confidence at every stage until a preliminary processor model is available. The model can then be used for activities such as processor design, arithmetic algorithm development, and object code development.

The APL notation is mathematically rigorous. Just a decade ago, a forerunner of this notation was used to present a precise formal description of a complete computer system, the IBM System/360. The primitive APL functions, expressible as distinct symbols and the APL operators provide a concise convenient notation for handling algorithms, with the added advantage of direct execution in the notation. APL accrues additional advantages from its simplicity, versatility and flexibility. These are properties of the language itself which are familiar to APL users, and so, are not emphasized here. Their presence will be obvious as the examples are unfolded.

RAAP MODEL AND ALGORITHM DEVELOPMENT

The RAAP (residue arithmetic associative processor) is an associative processor having a number of semi-independently operating associative memories, and whose basic arithmetic operations are performed in residue arithmetic format. The processor design is not fixed and the work reported on herein concerns preliminary development work on basis RAAP memory structure, and algorithm development and evaluation.

The RAAP consists of a group of associative arrays operating in parallel under the control of one program which resides in the control store (refer to Figure 1). Each associative array is associated with a different modulus of the residue
base representation of a data word. One microcoded program simultaneously controls all residue moduli computations (associative memory operations).

Although no attempt was made to finalize the architecture or design of a specific RAAP configuration, an approach to processor organization functioning and control was needed for the development and evaluation of the arithmetic algorithms. Figure 2 is a general block diagram illustrating present thinking.

For conceptual and practical convenience we consider the RAAP to consist of a number, N, of basic associative memory units, A1, A2, . . . , AN, having a similar form. Each associative memory unit contains a control field, and from one to four fields to store data. The conventional mask, data and selector registers are associated with each memory.

The associative memories are controlled by one master control unit which implements the arithmetic and input/output conversion algorithms. A key feature is that the associative memories can interact through their selector registers. The memories are interconnected through a Selector Transfer Register which permits the transfer of selector register contents from one memory unit to another.

Memories A1 and A2 have respective storage fields F1 and F2. These memories will handle input/output to the

RAAP and are also involved in the input/output conversion to residue base representation. Each of the associative arrays, A3, A4, . . . , AN, handles one modulus of the residue base representation. As indicated in Figure 1, array AK has control field CK, residue fields RK1, RK2, . . . , RK4, and data, mask and selector registers, DK, MK, and SK, respectively. Each memory in Figure 2 has an area labeled storage associated with it. This represents the portion of the memory which contains the necessary constants for residue conversion, overflow computations, masking and other operations.

Our study was confined to problems of processing positive integers only, there being no loss in generality for multiplication. Each associative array was assumed to contain up to four data fields and one field of control bits. Since residue base representations require significantly fewer bits per operating field than standard representation, the application of residue arithmetic in an associative processor might be expected to provide a processing speed advantage over a conventional associative processor. An APL model was developed to aid in investigating required arithmetic algorithms, processor architecture, and RAAP processing speeds as compared to conventional associative processing.

The APL model consists of two segments representing both a decimal residue processor and a binary RAAP model as illustrated in Figure 3. Input data consists of a vector of positive integers. The model converts the input data to a matrix containing the residue values corresponding to each integer. The decimal residue processor is simply a hypothetical processor which serves as a vehicle for preliminary examination of the properties of residue arithmetic and

From the collection of the Computer History Museum (www.computerhistory.org)
convenience in output display. The residue matrix is transformed into a three-dimensional binary array which serves as input to one data field of the RAAP architecture. After arithmetic processing by the RAAP, the contents of a data field are transformed into a matrix of residues. The outputs from the RAAP or the decimal residue processor are then converted to numerical representation through the use of the Chinese Remainder Theorem. The output appears as a vector of positive integers.

In the APL model (refer to Figure 4) an associative array is represented by four data fields (FLD1-FLD4) and a group of control bits (CTL). The number of control bits varies with the algorithm under test. All associative arrays are under the control of the SEARCH and WRITE functions.

In an associative processor a search operation interrogates specified bits (usually denoted by a mask register) of all words in the memory and identifies to a selector register, those words which identically compare with the contents of a data register. In this model, the right argument of a search statement specifies the following:

1. Bit mask—four data fields and M control bits
2. Selector register—one of two selector registers with/without logical OR capability
3. Data register—content associated with bit mask.

In an associative processor, a write operation transfers the contents of a data register to specified bits (based on the mask register) of those memory words identified by the content of a selector register. In this model, the right argument of a write statement specifies the following:

1. Bit mask—three data fields and M control bits
2. Selector register—one of two registers with true or complement form
3. Data register—content associated with bit mask.

One of the basic problems in developing a computer model for a conceptual processor configuration is the requirement that the model be readily adaptable to change. The dynamic linkage of APL functions considerably alleviates the necessity for recoding call sequences. For example, if a variable NAMES contains a name list of APL functions and a variable FLAG contains a vector of indices to NAMES, then an APL function INVOKE can be defined to invoke a call sequence based on FLAG. Figure 5 illustrates this example. The execute function is used by INVOKE to perform the construction and execution of statements under program control.

The structure of the RAAP model is illustrated in Figure 6. One may run the model by assigning the indices to FLAG and executing the function RESIDUE. The function RESIDUE initializes the model and INVOKE dispatches each of the major functions. Names starting with the letter "B" apply to functions which operate on binary data and names starting with the letter "P" apply to functions which printout data. TRACE lists the names of all functions invoked.

The basic operation of the model is illustrated in Table I. The contents of the variable FLAG show the call sequence. Run number 8 of RESIDUE produces a listing of all functions invoked, the input data and its residue representation.
the residue representation of the output data, and the numerical value of the output data. Each of the six input words was converted to eight residue values. For example, an input of 500 is represented by 27 with respect to a modulus of 43. The contents of data field 1 were transferred to data field 2 by the function TRANSFER2 and the function ADD added the number to itself. For example, 27 + 27 with respect to modulus 43 is 11. Taking into account all eight moduli, the output value is 1000 which indeed is 500 doubled.

The results of binary multiplication are shown in Table II for run number 23. One input (1473) was converted to residues with respect to six moduli. The contents of data field 1 were transferred to data field 2 and the function ENCODE2 converted these values to binary form. The algorithm under test, BMULTIPLY2, initialized FLD2 with the multiplicand and FLD3 with the multiplier (which are equal). The results of performing binary multiplication in the RAAP are shown under output. For example, 42 times 42 with respect to modulus 53 is 15. Each residue was then converted back to its decimal equivalent by the function DECODE1 and the function CRT (Chinese Remainder Theorem) computed the value 2169729 from the six residues. Thus, since 1473 squared is equal to 2169729 the run was successful.

In Reference 4 much more elaborate examples of APL utilization for RAAP structure and algorithm development are presented. The interactive use of the model with printout of binary field content during intermediate arithmetic stages permitted the quick isolation of deficiencies. Considerations of carry and overflow techniques, field utilization, modulus adjustment, selector transfer and control bit usage became evident as successive versions of the algorithms were tested. In this manner several addition and multiplication algorithms were developed.
APL As a Development Tool for Special-Purpose Processors

The first task was to create, in an APL workspace, a functional replica of the set of object machine registers and the memory. This included general registers (REGS), a “push down” stack of address registers (STACK) and flip flops (FF) to indicate the result of arithmetic operations and parity (refer to Figure 7). These were easily represented as APL Boolean arrays, (vectors and matrices). APL routines to perform elementary machine functions such as STEPIC (STEP Instruction Counter), GETOP (GET OPeration code), etc., were then coded in APL, and a main control routine was written to call the other routines, treating them as subroutines. Each of the approximately 50 instructions of the object machine repertoire was representable within this main routine by not more than three short lines of APL code, most by one line.

As the process of building this package progressed, it was possible at all stages to check proper emulated processor operation by entering sample machine code into the simulated memory, limiting the subset of machine code to that which was then possible to interpret. Interaction was of particular value during this process, since program errors were eliminated en route, and not allowed to go undetected until the package was thought to be complete. The package was supplemented by APL support software that facilitated loading, trace during execution, and code error diagnostics of object machine code.

The final task consisted of writing the machine object code that was to interpret and execute the problem-oriented language as entered from a keyboard. Using the APL terminal, both in its normal role of user-APL interface, and in the role of a simulated prototype keyboard for machine code entry, and again, later, for problem-oriented language source-code entry, the prototype package was developed and entirely checked out through processor emulation. Again, the interactive feature of APL was exploited, this time with machine source code developed and tested incrementally. At every stage of code development, trial runs were invoked, and each partial package was thoroughly checked out before proceeding with the next development phase.

It is estimated that, counting both tasks, more than 150 “turnarounds” were invoked. The effective total turnaround time, measured in human terms, amounted to no more than 10 to 15 minutes.

CONCLUSIONS

In this paper we have illustrated ways in which APL can be useful as a development tool for special-purpose processors. It provides simplicity and practicality of application in a time-sharing environment and, as such, provides a powerful capability for processor modeling.

In the first example, the APL model was extremely useful during the study of algorithms for a residue arithmetic associative processor. It proved quite flexible for conceptual processor features such as: bit precision per residue, number of associative memories, modulus per associative memory, number of control bits per memory word, number of words per memory and bit size per I/O word. Dynamic reconfiguration of the APL model aided in reducing program debug time as well as enabling interactive development and analysis of the arithmetic algorithms. The APL model enabled prediction of timing characteristics for the arithmetic algorithms operating in the associative processor.

For the second example, an APL simulation of a microcomputer enabled development of an interpreter in machine language code and validated the machine specifications. APL was then successfully used as an emulator for the prototype processor. The entire effort was accomplished in three weeks, and the developed software ran faultlessly in the microcomputer on the first and all subsequent trials.

ACKNOWLEDGMENT

The authors wish to acknowledge the support and encouragement of Col. R. Ives and Col. T. Wachowski of the Air Force Office of Scientific Research and Dr. R. Van Blerkom of IBM.
REFERENCES