HIGH DENSITY PARALLEL PROCESSING

II. Software and Programming

H. T. Nguyen, R. Raghavan, C. H. Ting, H. S. Truong

Lockheed Palo Alto Research Laboratories, Palo Alto, CA

Summary

Tools, utilities, assemblers and compilers are needed to develop programs which can be run on our parallel processor system, making use of the full power of the GAPP processor array and the Distributed Macro Controller. Some of the software tools are described here and a few examples are also given to illustrated the process of software development on this system.

In order to make the parallel processor system generally useful for experimentation by those not familiar with the hardware, a set of software tools is being developed. A host-resident console program was produced that allows program loading, program execution at arbitrary starting addresses, program halting, and examination of status and error flags.

An assembler program has been developed to assemble GAPP instructions and address macros for the Macro Generator Units. This program has a unique structure as it must deal with three concurrent instruction streams, and keep track of relative timing or program lengths among them. This assembler has also to take features of a high level compiler so as to generate appropriate instruction streams for the Flow Control Unit, which is the focal point of the entire DDP system, coordinating the GAPP processor array with the two Macro Generator Units.

1. The Console Monitor

The Console Monitor is a program which runs on the IBM AT host computer. It allows a user to perform some primitive operations on the Distributed Macro Controller (DMC) system, such as initiation, loading code into the Flow Control Unit and the Macro Generator Units, running a flow program, and monitoring the status of the DMC-GAPP system while it is running.

The DMC is designed so that all its writable control store memory areas can be accessed by the host computer. In fact, all the writable control store memory and many of the important internal registers in both the Flow Control Unit and the Macro Generator Units are mapped to a contiguous 128 Kbytes of memory. The mapped memory locations can be interrogated by the host, and new code or data can be loaded into these registers and memory areas through the Console Monitor. Effective use of this feature allows a user to assemble GAPP programs and flow programs directly. It is extremely valuable during testing and debugging phases of program development.

The Console Monitor also has many built in high level functions. One is loading programs in either binary form or hexadecimal form. The binary form of program is simply the image of the 128 Kbyte mapped memory, which can be saved as a binary file on the disk of the host computer. The saved binary file can be loaded back into DMC to restore DMC to the state when the saved file was generated. A hexadecimal file format was defined so that code written in hexadecimal numbers can be translated and downloaded into various selected parts of DMC. This file format also specifies the output file produced by GAPP assemblers and DMC compilers which can be run on other host computers for off line program development.

The other important feature of the Console Monitor is that it can load data into the GAPP corner-turn processor array and unload results from the corner-turn array, through a DMA board inside the IBM AT. This is necessary for testing GAPP and DMC programs and verify that the hardware and software are in working conditions. It is also useful in testing various algorithms and evaluate their performance. The Console Monitor thus serves as the major user interface to the Development Demonstration System.

2. GAPP and Address Macro Assembler

GAPP is a Single-Instruction-Multiple-Datapath (SIMD) machine. Its ALU unit is an one bit full-adder-subtractor, and there are many data paths around the ALU and the internal registers and memory. A GAPP instruction is used to specify precisely the data paths to and from the registers and the memory. A GAPP instruction is 20 bit wide, 13 bits for data multiplexers and 7 bits for memory selection, as shown in Figure 1. A GAPP program is thus a sequence of these 20 bit instructions commanding the GAPP array how to route data and results inside the processor array. It is fully programmable in the sense that any function that does not exceed the memory capacity of the machine can be executed. However, all processing units execute identical instruction so that a general MIMD (Multiple Instruction Multiple Datapath) command is executed at lower efficiency.

To facilitate programming efforts, a set of GAPP mnemonics is defined to specify the source and destination of data in each clock cycle. The Assembler translates these mnemonics into GAPP code and memory address specifications, and constructs macro routines which are callable by flow programs. As there are many data paths independently controllable by a single GAPP instruction, the assembler allows the user to specify multiple data paths in a single instruction, as well as the detailed function which has to be carried out by the address macro generator, such as pushing, popping or incrementing the memory pointers on the stacks.

The Macro Assembler generates two concurrent macro's from a set of mnemonic code sequence, terminated by the special operator '_$_'. The address macro takes the most significant 16 bits and the GAPP macro takes the least significant 16 bits of an assembler 32 bit code. The two parts will be separated and downloaded to their respective writable control store memory in the Macro Generator Units.

A library of macro routines are assembled and kept in the Macro Generator Units for the flow control program to call. For specific applications, special macro routines can be define by the user and downloaded to be used together with the library macros. Figure 2 shows a sample of the macro routines required by the Game of Life flow program.

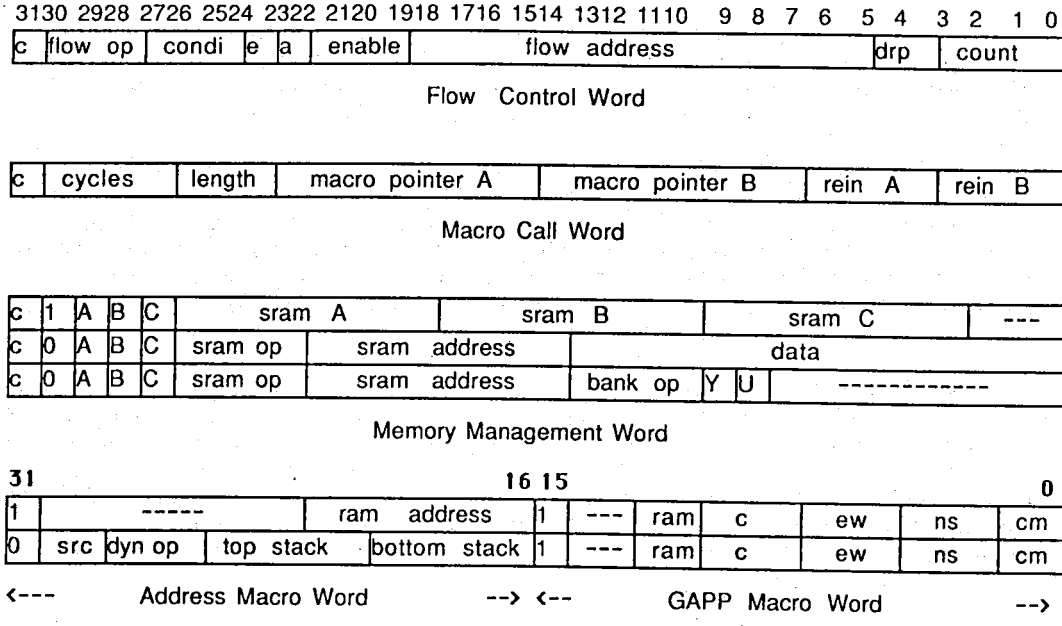

Figure 1. Instructions Formats for Distributed Macro Controller

```
g: init-life   8005 gadr c=p _$_   _$_   _$_

g: ew1sum   8005 gadr p:ns:ew=c _$_
      ew=w _$_
      ns=ew ew=ns _$_
      ew=e _$_
      8000 gadr ns:ew=plus carry _$_
      8001 gadr p=c _$_

g: ns2sum   c=0 ns=s _$_
      8002 gadr plus carry _$_
      8001 gadr ns:ew=p _$_
      ns=s _$_
      8003 gadr plus carry _$_
      8004 gadr p=c _$_


g: ns3sum   8000 gadr ns=p c=1 _$_
      8002 gadr ew=p ns=n _$_   8002 gadr plus carry _$_
      8001 gadr ns=p _$_
      8003 gadr ew=p ns=n _$_
      8003 gadr plus carry _$_
      8004 gadr ns=0 ew=p _$_   8004 gadr ew=plus _$_

g: final   8003 gadr ns=p c=0 _$_
       8002 gadr ns=p borrow _$_
       8002 gadr p=c c=1 _$_
       8005 gadr ew=p _$_
       8002 gadr borrow ew=p ns=0 _$_
       carry _$_


: scratch   807f gadr ;

g: c-to-scratch   scratch   p=c _$_   _$_   _$_

g: shift-scratch   scratch cm=p _$_   south _$_
     scratch p=cm _$_

g: scratch-to-ew   scratch ew=p _$_   _$_   _$_

g: ew-to-scratch   c=ew _$_   scratch p=c _$_   _$_

g: mm-to-scratch   top c=p _$_   scratch p=c _$_   _$_

g: scratch-to-mm   scratch c=p _$_   top p=c _$_   _$_
```

Figure 2. Macro routines for Game of Life.

3. Flow Control Assembler

The Flow Control Unit stores its instructions in 32 bit words, each flow instructions requires 3 to 18 flow words. The first word in a flow instruction is called Flow Control Word which specifies the program flow, such as JUMP, LOOP, CALL, and RETURN. The second flow word specifies the macro's to be executed from the Macro Generator Units and the clock cycles and number of macro instructions to be executed in this macro. It is called Macro Call Word. The third and following flow words are Memory Management Words, which controls the memory management mechanism in the Address Macro Generator Unit. The formats and the functional fields in these flow words are show in Figure 1.

The Flow Control Assembler compiles the flow control mnemonic code, similar to those commonly used in high level programming languages, and generate sequences of flow words. The flow words will then be downloaded into the Flow Control Unit. Any of the flow instruction, starting with a Flow Control Word, can be executed and the DDP will perform algorithm specified by this and subsequent flow instructions until completion.

Since the Flow Control Unit supports all the fundamental programming structures, such as conditional and unconditional branching, looping, subroutine calling and returning, flow programs can be modularized and written in highly structured form. This practice greatly enhances program readability and eases debugging and maintainence. The Flow Control Assembler produces efficient code within this structured framework.

An example of flow program is show in Figure 3, implementing the Game of Life. It uses the macros assembled by the macro assembler. The core or the Game of Life program is a 25 machine cycle sequence of GAPP instructions, assembled into 4 macros which are called by the main program. The most time consuming part of the program is to dump the map of lifes out to the display device, which is isolated as a subroutine called from the main program loop.

4. Conclusion

We have built enough tools and utilities to program and use the high density SIMD processor arrays efficiently. Complicated algorithms can be broken into address macros, op-code macros and flow control sequences, which can be assembled from high level source program into machine code loaded into the parallel processor system and executed. Effective use of macros thus tremendously simplifies the programming efforts and greatly compress the program. The limited experience we have gained in the past few weeks exceeds our expectation that the DMC controller allows massively parallel processors to be conviently programmed using high level language without compromising the performance. The program compression will allow much more complicated algorithms to be expressed concisely for the evantual execution on huge processor arrays.

Several articles of this project, as well as research results on parallel computations and on applications have been prepared in more detail for publication.

```
flow-block output-life    ( from gadr 7f scratch plane)
   loop 9 times   4 gre ( frame mark)  _$$$_
   loop 0c times _$$$_
      noop ram-en _$_   shift-scratch macro _$$_
      noop ct-en _$_   shift-scratch macro _$$_ ( 2 msb bits)
      loop 4 times _$_    scratch-to-ew macro _$$_
      loop 10 times _$$$_
         noop _$_    shift-w macro _$$_
      end-loop _$$$_
      end-loop _$$$_
   end-loop _$$$_
   end-loop _$$$_
   return _$$$_
end-block

flow-block game-of-life
   noop mm-en _$_    init-life macro _$$_
flow-block repeat-life
   noop mm-en _$_    6 cycles 6 instructions ew1sum macro _$_ _$_
   noop mm-en _$_    6 cycles 6 instructions ns2sum macro _$_ _$_
   noop mm-en _$_    8 cycles 0 instructions ns3sum macro _$_ _$_
   noop mm-en _$_    6 cycles 6 instructions final macro _$_ _$_
   call output-life adr mm-en _$_    c-to-scratch macro _$$_
   jump repeat-life adr _$$$_
end-block
   stops
end-block
```

Figure 3. Flow program of Game of Life.