GAPP instruction simulator

1988

Ronald P. Bauman
University of Central Florida

Find similar works at: https://stars.library.ucf.edu/rtd

University of Central Florida Libraries http://library.ucf.edu

STARS Citation

https://stars.library.ucf.edu/rtd/4258

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
ABSTRACT

A need was recognized for the development of a hardware independent simulator for the Geometric Arithmetic Parallel Processor, or GAPP. This simulator would allow the user to test algorithms targeted to GAPP systems using readily available personnel computers. The development of the simulation, a GAPP Instruction Simulator or GIS, is the basis of this research report.

This report includes a discussion of earlier parallel processors of similar type to the GAPP, a discussion of the GAPP's operation and instructions, and illustrations of GAPP applications. The report focuses on the GAPP Instruction Simulator with a discussion of its operation and a listing of the software implementation.

The GIS software was compiled and test executed with results discussed in the conclusion of this report. The GIS provides a useful simulator for GAPP application design testing without having GAPP hardware available.
# TABLE OF CONTENTS

**LIST OF FIGURES** .............................. iv

**Chapter**

I. INTRODUCTION .................................. 1

II. EARLY SYSTOLIC ARCHITECTURES ............... 2
   Am2903 ........................................ 2
   Illiac IV ....................................... 4
   STARAN ......................................... 6

III. GAPP .......................................... 9
   Description .................................... 10
   Instructions ................................... 12
   Applications ................................... 12

IV. INSTRUCTION SIMULATOR .......................... 14
   Operations ..................................... 14
   Instructions ................................... 15

V. CONCLUSIONS ................................... 18

APPENDICES ...................................... 19
   A. GIS Instructions ............................ 20
   B. GIS Sample Programs  ....................... 26
   C. GIS Program Listing ....................... 28

REFERENCES ...................................... 48
LIST OF FIGURES

1. Block Diagram of AM2903 Bit Slice Processor . . . 3
2. Illiac IV Array Block Diagram . . . . . . . . . . . . 5
3. STARAN Block Diagram . . . . . . . . . . . . . . . . 7
4. Addition On STARAN . . . . . . . . . . . . . . . . 7
5. GAPP Processing Element . . . . . . . . . . . . . . . 10
6. Pattern Detection . . . . . . . . . . . . . . . . . . 12
7. Digital Filter . . . . . . . . . . . . . . . . . . . . 13
8. Example GIS Program . . . . . . . . . . . . . . . . 17
CHAPTER 1

INTRODUCTION

Geometric parallel processors have had long proven performance for increasing data throughput rates in specialized applications. A geometric parallel processor is one which the processing elements are arranged geometrically, such as in a matrix-like array. A geometric parallel processor can provide a powerful computational tool for such applications as array manipulations, bit pattern recognition, and digital filters.

Earlier geometric parallel processors provide good examples of parallel architectures. Examples of these processors are the Am2903, Illiac IV, and STARAN. The recent Geometric Arithmetic Parallel Processor chip (GAPP) has put geometric parallel processing on a microprocessor scale. The GAPP chip contains 72 bit processing elements, and the chip may be cascaded together with other chips.

This report describes the GAPP Instruction Simulator, or GIS, and its program requirements for operation. The operations and instructions for the GIS system are demonstrated.
CHAPTER II

EARLY SYSTOLIC ARCHITECTURES

Three early parallel array processors and systems are focused on. The GAPP processors falls under the category of a Single Instruction Multiple Data computer or SIMD. The GAPP is made up of identical multiple processing elements or PEs. Each PE is connected to its closest neighbor for the purpose of passing data. Three other architectures are very similar in principles and design to the GAPP, and will be discussed. They are the Am2903 bit slice processor, the Illiac IV parallel array system, and the STARAN associative array processor.

Am2903

The Am2903 is a four bit cascadable microprocessor with microinstruction input. The Am2903 consists of 16 four bit registers, any two of which can input into the ALU (Figure 1). Input from the outside can also be used with the ALU. Further, the Am2903 contains a Q register for the purpose of multiplication (White 1981, 127).

The Am2903 designed in 1976 is a simple architecture in terms of what it can do, and it is important because of its flexibility and cascade ability. For example, eight
Figure 1. Block Diagram of AM2903 Bit Slice Processor (White 1981, 127).
chips may be connected together giving it a 32 bit word. The Am2903 must be supported by a microprogram sequencer and control logic.

**Iliiac IV**

The Illiac IV is a large parallel array system developed in the 1960s. It contains 64 separate processing elements or PEs. Each PE is 64 bits long and capable of doing floating point operations in hardware. Each PE has associated with it 2K by 64 bits of memory (Figure 2).

The Illiac IV is a stand-alone computer because of its control unit (CU). The control unit is a scalar computer, and it also feeds instructions to the 64 PEs. Each PE may be disabled which disable instructions in the PE. The CU can process scalar functions in parallel with the 64 PEs processing vector functions.

The PEs are connected to each other in an 8 by 8 array. Each PE (0-63) can connect to four other PEs for the purposes of passing data between each other. For example, PE 1 can connect to PEs 57, 2, 9, and 0. The PE can also handle data of 48, 32, 24, and 8 bit units (Siewiorek 1982, 308).

Because of its flexibility, there are many applications for the Illiac. One of these applications is statistical processing where large arrays of numbers must be multiplied or added together hundreds of times.
Figure 2. Illiac IV Array Block Diagram (Siewiorek 1982, 310).
The STARAN is an associative array processor designed in 1981. It has an associative array memory of 256 words. Each word is 256 bits long (Figure 3). Each of the 256 bit columns has a one bit processing element.

Associative memory is a type of content addressable memory. If a data word is to be searched for, each bit of the data word is loaded into a separate PE. All bits in memory are then compared at once to the word in the PE. For each word that matches in memory, a corresponding tag bit is set. The tag bits may then be used as a mask so that operations may only be performed on the matched words.

The associative memory can be accessed in two ways. First, the control logic unit of the STARAN can access memory in a standard word configuration. It can read/write to any one of the 256 words of the array or any part of the 256 bit word. This is done to load memory so that the PEs can work on the bits in parallel. Second, each of the 256 PEs can access memory in a parallel bit arrangement. This access may be masked by the M register of the PEs. The 256 PEs may be viewed as several long registers and ALUs (Figure 3) (Siewiorek 1982, 318).

Each PE has two similar registers X and Y. Both X and Y have an ALU associated with them. The X ALU has two inputs, the current state of X and a bit slice of memory. The X register is loaded with the current value of the ALU.
Figure 3. STARAN Block Diagram (Siewiorek 1982, 318).

Figure 4. Addition On STARAN. A and B are added together to get C. 256 of these bit serial additions may be done in parallel (Siewiorek, 1982, 320).
The Y register works in the same way, and the X and Y ALUs may operate in parallel (Siewiorek, 1982, 319).

The following is an example of addition of two arrays which contain columns of numbers. Each number in the column is $N$ bits long. A typical row in the columns is memory word K (Figure 4). The numbers $A$ and $B$ are stored in a part of word K, and the result $C$ is also stored in word K. A loop of $N$ times would be necessary to perform the addition of $A$ and $B$. At the same time, 256 separate additions in the columns may be done in parallel.

The STARAN shifts bits in the PEs with the Flip Network. It may perform one multi-bit shift in multiples of 2 to the power of $N$ bits. This shift is done in one instruction cycle (Siewiorek 1982, 320).
CHAPTER III

GAPP

The GAPP, or Geometric Arithmetic Parallel Processor is a bit array chip very similar to the STARAN. The GAPP is a single chip processor capable of being cascaded together both vertically and horizontally. This chip does not contain any instruction control logic.

Description

A single GAPP chip consists of 72 processing elements (PE). Each PE contains 4 one bit registers. The registers are labeled CM, NS, EW, and C (Figure 5). Each PE contains a full adder and its own RAM.

The 72 PEs are connected in a 6 by 12 array fashion. PE 11 connects to PEs 1, 12, 21, and 10. Each PE passes data to its neighbors and to other cascaded chips in three ways. The first is the East/West register (EW) which can shift in both an east and west direction. The second is the North/South register (NW) which can shift in both a north and south direction. The third is the CM register which can shift data in a north direction.
Figure 5. GAPP Processing Element (NCR 1987, 6).
Table 1. Instruction Set For the GAPP (NCR 1987, 7).

<table>
<thead>
<tr>
<th>Register</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>CM := CM</td>
<td>Micro NOP</td>
</tr>
<tr>
<td></td>
<td>CM := RAM</td>
<td>Load CM from RAM</td>
</tr>
<tr>
<td></td>
<td>CM := CMS</td>
<td>Move from CMS into CM</td>
</tr>
<tr>
<td></td>
<td>CM := 0</td>
<td>Load zero into CM</td>
</tr>
<tr>
<td>NS</td>
<td>NS := NS</td>
<td>Micro NOP</td>
</tr>
<tr>
<td></td>
<td>NS := RAM</td>
<td>Load NS from RAM</td>
</tr>
<tr>
<td></td>
<td>NS := N</td>
<td>Move from N into NS</td>
</tr>
<tr>
<td></td>
<td>NS := S</td>
<td>Move from S into NS</td>
</tr>
<tr>
<td></td>
<td>NS := EW</td>
<td>Move from EW into NS</td>
</tr>
<tr>
<td></td>
<td>NS := C</td>
<td>Move from C into NS</td>
</tr>
<tr>
<td></td>
<td>NS := 0</td>
<td>Load zero into NS</td>
</tr>
<tr>
<td>EW</td>
<td>EW := EW</td>
<td>Micro NOP</td>
</tr>
<tr>
<td></td>
<td>EW := RAM</td>
<td>Load EW from RAM</td>
</tr>
<tr>
<td></td>
<td>EW := E</td>
<td>Move from E into EW</td>
</tr>
<tr>
<td></td>
<td>EW := W</td>
<td>Move from W into EW</td>
</tr>
<tr>
<td></td>
<td>EW := NS</td>
<td>Move from NS into EW</td>
</tr>
<tr>
<td></td>
<td>EW := C</td>
<td>Move from C into EW</td>
</tr>
<tr>
<td></td>
<td>EW := 0</td>
<td>Load zero into EW</td>
</tr>
<tr>
<td>C</td>
<td>C := C</td>
<td>Micro NOP</td>
</tr>
<tr>
<td></td>
<td>C := RAM</td>
<td>Load C from RAM</td>
</tr>
<tr>
<td></td>
<td>C := NS</td>
<td>Move from NS into C</td>
</tr>
<tr>
<td></td>
<td>C := EW</td>
<td>Move from EW into C</td>
</tr>
<tr>
<td></td>
<td>C := CY</td>
<td>Load C from Carry</td>
</tr>
<tr>
<td></td>
<td>C := BW</td>
<td>Load C from Borrow</td>
</tr>
<tr>
<td></td>
<td>C := 0</td>
<td>Load zero into C</td>
</tr>
<tr>
<td></td>
<td>C := 1</td>
<td>Load one into C</td>
</tr>
<tr>
<td>RAM</td>
<td>Read</td>
<td>Read from RAM</td>
</tr>
<tr>
<td></td>
<td>RAM := CM</td>
<td>Load RAM from CM</td>
</tr>
<tr>
<td></td>
<td>RAM := C</td>
<td>Load RAM from C</td>
</tr>
<tr>
<td></td>
<td>RAM := SM</td>
<td>Load RAM from Sum</td>
</tr>
</tbody>
</table>
Instructions

Each of the four registers and a write operation may function in parallel with no conflict. This gives a total of five commands which the user may execute in a single instruction cycle (Table 1).

Applications

The GAPP chip has applications in the following areas: pattern recognition, image processing, parallel data processing, and as an associative processor (NCR 1987, 1).

The pattern recognition of a series "110" is a straightforward demonstration of the GAPP's logical capabilities. If a pattern of "110" is detected, a "1" is sent to the output memory. The input example is an array of six rows of eight bits (Figure 6) (Davis 1984, 213).

<table>
<thead>
<tr>
<th>Input plane</th>
<th>Output plane</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 1 1 0 0 1 1 0</td>
</tr>
<tr>
<td>2</td>
<td>1 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>1 1 1 0 1 1 0 0</td>
</tr>
<tr>
<td>4</td>
<td>1 0 1 0 1 0 1 1</td>
</tr>
<tr>
<td>5</td>
<td>1 1 0 1 1 0 1 1</td>
</tr>
<tr>
<td>6</td>
<td>1 1 0 0 0 1 1 1</td>
</tr>
</tbody>
</table>

1. EW := RAM(0); Load pattern into EW reg.
2. EW := E; Shift EW reg. right
3. NS := EW, EW := E, C := 0; CY <- NS and EW when C=0
4. NS := RAM(0), C := CY; Load NS with pattern
5. EW := C, C := 0; BW <- NS' and EW when C=0
6. C := BW;
7. RAM(1) := C; write output pattern

Figure 6. Pattern Detection (Davis 1984, 213).
The next example is a digital filter. A digital signal of two "1" or two "0" must come together. If there is a "01" or "10" pattern, then an error has occurred, and it must be corrected. Figure 7 is an example of sample data and the GAPP instructions to filter the input signal.

Input stream: stored at RAM(0)
1 1 0 0 1 0 1 1 0 0 0 1 1 1 1 0 1 1 1 1 0 1 1

Output stream: stored at RAM(3)
1 1 0 0 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1

Pattern to determine first and second bit: RAM(1)
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0

1. NS := RAM(1), C := 0; Get 1st and 2nd pattern
2. EW := RAM(0); Load input signal
3. C := BW; BW <- NS' & EW
4. RAM(2) := C, C := 0, EW := W; Shift EW reg. left save C in RAM(2)
5. C := CY, EW := RAM(2), NS := 0; CY := EW & NS
6. C := BW; BW := EW + C
7. RAM(3) := C; write output pattern

Figure 7. Digital Filter.
CHAPTER IV

INSTRUCTION SIMULATOR

With the expensive cost of a GAPP system, and the lack of debugging tools for the GAPP system, a need existed for a GAPP Instruction Simulator (GIS) to run on a personnel computer. The GIS simulates the instruction set of the GAPP processor (Table 1). The GIS simulates two GAPP chips cascaded together and uses a simple control language resembling Pascal. The control language allows branching and looping of the GAPP instructions. Furthermore, the GAPP Instruction Simulator includes special commands to look at the present state of the GAPP registers and RAM memory. The most important of these commands is for the terminal display of memory, which allows the operator to examine an entire array of data instantly.

Operation

The GIS operates like an interpreter. It is written in Turbo Pascal language, and it runs on an IBM PC computer. The GAPP Instruction Simulator reads a set of GIS instructions which include the actual GAPP instructions (Table 1).
The GIS will initially read from a file a list of numbers to load the GAPP's 128 by 144 bit memory representing the two chips. There are 144 bits in a word. The GIS breaks this up into twelve integers of 12 bits each, for loading of the GAPP memory. Moreover, it is how the GIS will display memory, and how it will display the seven registers (CM, NS, EW, C, SM, CY, and BW).

After the GIS reads all the instructions, it will read and load the GAPP memory file. All or any part of the memory may be loaded. The load file contains optional addresses from 0 to 127. The addresses are followed by twelve integers representing the bit pattern for that word. For example, to load a twelve bit binary pattern of "000000011000" into ram locations five and six, the following would be entered in the file. The phrase "$18" represents a hexadecimal number.

A5 $18 $18 $18 $18 $18 $18 $18 $18 $18 $18 $18 $18
$18 $18 $18 $18 $18 $18 $18 $18 $18 $18 $18 $18

Instructions

The instructions in GIS fall into three categories: GAPP instructions, control commands, and special commands. GAPP instructions are shown in Table 1. GIS has the following commands: FOR; WHILE; IF THEN ELSE; and an assignment statement. These commands are similar to Pascal in nature (Appendix A). The special commands are as
follows: PRINT REG; PRINT MEM; SHOW MEM; INC; DEC; and ROTATE.

The PRINT REG statement prints out a select group of the seven registers. The PRINT MEM statement lists a select group of memory addresses. While the PRINT MEM statement lists GAPP memory as groups of twelve numbers, the SHOW MEM statement displays GAPP memory in graphic pixel formation.

The INC statement aids in memory references made to GAPP memory. If a GAPP memory reference is made inside a loop, an INC statement will increment by one the memory address of the previous GAPP instruction. This is done each time a memory reference is encountered in the loop. The DEC statement works in the same manner except the GAPP memory address is decremented by one for each encounter.

The ROTATE statement works with the EW register of the GAPP. When the EW outputs of a GAPP chip are not connected to any inputs, an EW register shift (EW := E) will loose a bit. When the ROTATE statement is enabled, the ordinarily lost bit will go to the PE in the row below the PE where the bit is coming from. This allows 144 bits (with two GAPP chips) to be shifted as on word. When the ROTATE statement is not used, the word length for a shift is 12 bits.

Figure 8 is a sample GIS program. The program loads a memory word from location zero, and it performs an "or"
operation with location five. The result is loaded into memory location ten. The program does this five times, and increments each memory location after it is accessed.

*FOR I := 1 TO 5 DO;
NS := RAM(0);
*INC;
EW := RAM(5);  C:=1;
*INC;
C := CY;
$PRINT REG,$32;
RAM(10) := C;
*INC;
*END;
END;

Figure 8. Example GIS Program.
CHAPTER V

CONCLUSION

Early parallel architectures provide a good understanding of the principles of the GAPP operations. The GAPP chip does not present any new concepts over the earlier STARAN and Illiac IV computers except possibly for its cascade ability. The big advantage is that the GAPP is a microprocessor where the STARAN and Illiac are main frame type computers.

The GAPP provides for many different applications. The biggest of these is real time data applications. Some specific real time applications are image recognition, image enhancement, and digital pattern recognition.

The GIS program lets simulation of two cascaded GAPP chips to take place. GIS also provides control language for looping and decision making inside GAPP instructions. The ROTATE statement which is a special feature of GIS, provides two different hardware shifts. The type of shift is an important concept in designing the GAPP chip to do a particular application, and is always required when the GAPP is used for image processing.
APPENDICES
Some general rules that apply to all GIS instructions are as follows.

- Instructions must end with a ";".
- Only one instruction may be on a line.
- Any text after the ";" is treated as a comment.
- Last line of the program must be the statement "End;".
- Instructions may either be in lower or upper case letters or both.

GAPP Instructions

GAPP instruction may be any of the instructions in Table 1. Five of the instructions may be placed on one line. Each of the four registers may have an operation plus a ram operation. Some examples are as follows:

1. CM := 0; NS := EW, EW := E, C := BW, RAM(0) := SM;
2. NS :=RAM(99); EW :=NS; C :=0;
3. C:=CY;

Each register load must be separated by a ",". The order the register loads appear in is optional. In the first instruction, all four registers are loaded, and a memory write is performed ( RAM(0) := SM ). Also the NS register is loaded with the EW register; at the same time, the EW register is shifted East. These two register operations do not interfere with each other. In the second instruction, three registers are loaded. The NS register is loaded from memory position 99. In the third instruction, only the C register is loaded. The other three registers are left in the same state.
Control Commands

Control commands must be prefaced by a "*". The "*" must be in column one of the command to help GIS decide which type of command is coming. All GIS commands may be nested inside each other.

Assignment Statement

An assignment statement may have two operands and may have the following operators "+", "-", "*", and "/". The first operand must be a variable and the second a constant if two operands are to be used. Operand variables may be up to four characters long. Some examples follow.

*JJ1 := KK1;
*N := M * 2;
*VAL := 2000;

FOR Statement

The for statement has the following format. Where "var" is a variable, and "var_k1" and "var_k2" are either constants or variables.

*FOR var := var_k1 TO var_k2 DO; var_k1 <= var_k2
{ block of statement(s) } *END;

IF THEN Statement

If the expression "var comp val" is true the block of statements will be executed. The phrases "var" and "val" are any variable or constant. The phrase "comp" is one of the following symbols: ">", "=", "<", ">", "<=", 


"\geq". The phrase "val" may also be a GAPP RAM value or REG (register) value. The RAM value may be indexed as RAM(i,j) where "i" may be from 0 to 127, and "j" may be a value from 0 to 11. The REG value may be indexed as REG(i,j). The variable "i" may be a value from 1 to 7. This value represents the GAPP registers CM, NS, EW, C, SM, CY, and BW respectfully. The variable "j" may be a value from 0 to 11.

*IF var comp val THEN
  { block of statement(s) }
*END;

IF THEN ELSE Statement

The form of the IF THEN ELSE statement is as follows.

*IF var comp val THEN;
  { block 1 of statement(s) }
*ELSE;
  { block 2 of statement(s) }
*end;

The "var comp val" expression may take on the same values as the IF THEN statement.

WHILE Statement

The WHILE statement repeats the block of statements until the "var comp val" expression takes on the same values as the IF THEN statement. The WHILE statement has the following form.

*WHILE var comp val DO;
  { block of statement(s) }
*END;
INC and DEC Statement

INC (increment) and DEC (decrement) statements add one and subtract one from the previous GAPP memory reference. The INC and DEC statement must be placed directly after the memory reference statements.

*FOR i := 1 TO 5 DO; loop 5 times
EW := RAM(5); load EW <- RAM(5,6,7,8,9)
*INC; increment previous address
*END; end FOR loop

In the preceding example, EW will be loaded with location five the first time through the FOR loop. The second time through, EW will be loaded with memory location, and the last time through, EW will be loaded with memory location nine.

Special Commands

Special commands must be prefaced by a "$". The "$" must be placed in column one of the commands. Special commands may be placed anywhere in the GIS program.

PRINT REG statement

The PRINT REG statement displays on the screen a select group of the seven registers (CM, NS, EW, C, SM, CY, and, BW). Any or all of these registers may be displayed by specifying a hexadecimal number. Some examples are:

$PRINT REG,$78; Register Selected
$PRINT REG,$32; CM, NS, EW, C
$PRINT REG,$7F; NS, EW, CY
Display all registers
PRINT MEM Statement

The PRINT MEM statement displays GAPP memory. The statement as the following form. where "begin" is the
$PRINT MEM, begin, end;
beginning address of memory, and "end" is the ending
address of memory. Memory ranges from address 0 to 127.
Each word of memory is printed out as twelve integers.
These integers may range from 0 to FFF hexadecimal.

SHOW MEM Statement

SHOW MEM displays memory as the print mem does except
the memory is displayed graphicly. Only the first ninety
memory words are displayed. The form of the statement is
as follows.

$SHOW MEM;
APPENDIX B

GIS SAMPLE PROGRAMS
The following program adds GAPP memory row 0 with the memory row 1, and stores the result in memory row 2. This is done on a 12 bit word.

**ENTER NAME OF GAPP PROGRAM INPUT FILE**
MY-FILE { GIS PROGRAM }

**ENTER NAME OF MEMORY (RAM) INITIAL LOAD FILE**
MY-DATA { GIS DATA }

```
a0 2000 6 100 50 44 1 500 0 0 2 3 8
 1000 14 400 50 56 2 5 0 7 2 2 2

$PRINT MEM,0,3;
C := 0, NS := RAM(0);
EW := RAM(1);
*for i := 1 to 12 do;
  C := CY, RAM(2) := SM;
  EW <- CY * 2
  EW := C, NS := RAM(2);
  NS <- SM
  EW := W, C := 0;
*END;
$PRINT REG,$3E;
$PRINT MEM,0,2;
end;
```

**ADDR**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>6</td>
<td>100</td>
<td>50</td>
<td>44</td>
<td>1</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>14</td>
<td>400</td>
<td>50</td>
<td>56</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**REG**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>20</td>
<td>500</td>
<td>100</td>
<td>100</td>
<td>3</td>
<td>505</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>20</td>
<td>500</td>
<td>100</td>
<td>100</td>
<td>3</td>
<td>505</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**ADD**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>6</td>
<td>100</td>
<td>50</td>
<td>44</td>
<td>1</td>
<td>500</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>1000</td>
<td>14</td>
<td>400</td>
<td>50</td>
<td>56</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3000</td>
<td>20</td>
<td>500</td>
<td>100</td>
<td>100</td>
<td>3</td>
<td>505</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
PROGRAM GAPP;

TYPE
V5T = ARRAY[1..5,1..8] of byte;
str = string[20];
stack = array[1..20] of integer;

CONST
  'CMS', 'N', 'S', 'E', 'W', 'CY', 'SM', '0', '1', 'EW');
V3: array[1..5] of byte = (4, 7, 7, 8, 3);
V5: V5T = ((1,5,6,13,0,0,0,0),(2,5,7,8,3,4,13,0),
  (3,5,9,10,2,4,13,0),(4,5,2,3,11,15,13,14),(1,4,12,0,0,0,0,0));
V8: array[1..3] of byte = (1, 4, 5); { cm,c,sm for load from ram}
V9: array[1..7] of byte = (1, 2, 3, 4, 12, 11, 15);
KA: integer = 6;
operators: string[22] = '/ - + * = > < <<>>=';

VAR
{ 2 CPUs } { 1 2 3 4 5 6 7 }
REG: array[1..7,0..11] of integer; { cm, ns, ew, cr, sm, cy, bw }
RAM: array[0..127,0..11] of integer; { GAPP memory }
CODE: string[80]; { one line of gapp program }
LINE: integer; { line of GIS code on }
SOURCE: text; { GIS program input file }
SOURCEFILE: string[14]; { GIS program file name }
MER: text; { initial memory (RAM) load values }
RamLoadFile: string[14]; { RAM load file name }
MA: char; { mem addr indicator for RAM }
COL_LEN: Byte; { occurrence of colon }
OPNUM: array[1..7] of byte; { last one is address field }
OPA: array[1..999,1..6] of byte; { list of op codes ka }
C: char; { variable character }
CS: string[20]; { variable char string }
FST_CHR: char; { first char of code }
parn_val: string[8]; { parameter char string }
parn_num: integer; { value of parn_val }
i,j,k,e,x,y,g,h: integer; { variables }
tar: 0..5; { target register of opcode }
B_dat: integer; { bit for bit maping of graphics }
NR: integer; { }
parn: boolean; { parentheses indicator }
icOL, ERROR, ECODE: integer; { error flag and code }
ERR_MES: string[50]; { error messages }
addr: 0..127; { RAM address }
NS,EW,CR: integer; { North/South, East/West, Carry }
NSS,NEW: integer; { not NS, not EW }
Beg_Mem, Len_Mem: integer; { Begging of Memory, length of }
REGMASK: integer; { for PRINT REG select which reg }
rotate: boolean; { east west register shifts }
fspace: boolean; { blank space in code }

}
begin
writeln('');
case ERROR of
  1: ERR_MES := ' = expected for register ';
  2: ERR_MES := ' invalid register' ;
  3: ERR_MES := ' input register not found for register ';
  4: ERR_MES := ' # invalide place' ;
  5: ERR_MES := ' bad parenthises' ;
  6: ERR_MES := ' address to small or large ' ;
  7: ERR_MES := ' illegal char in address ' ;
  8: ERR_MES := ' illegal char' ;
  9: ERR_MES := ' semi-colon expected' ;
 10: ERR_MES := ' bad special command' ;
 11: ERR_MES := ' bad special command PRINT MEM,begin,length;' ;
 12: ERR_MES := ' bad special command PRINT REG,regmask;' ;
 20: ERR_MES := ' Bad comparison operators' ;
 21: ERR_MES := ' Bad operands' ;
 22: ERR_MES := ' "THEN" expected' ;
 23: ERR_MES := ' Bad comparison statments' ;
 24: ERR_MES := ' "DO" expected' ;
 25: ERR_MES := ' ":=" expected' ;
 26: ERR_MES := ' bad assignment constant' ;
 27: ERR_MES := ' "TO" expected' ;
 28: ERR_MES := ' Unexpected END statement' ;
 29: ERR_MES := ' To many commands on a line' ;
 31: ERR_MES := ' Bad CS stack, program problem' ;
 32: ERR_MES := ' Word to big' ;
{ CM = RAM(x), RAM(x) = SM }
else ERR_MES := '';
end; {case}
writeln('ERROR : ',ERROR,' ',ERR_MES);
end; {proc}

{##### Convert GIS code into uppercase characters }
Procedure ReadUpCs;
begin
readln(SOURCE, CODE);
writeln(CODE);
COL_Len := pos(';',CODE);
for j := 1 to COL_Len-1 do CODE[j] := UpCase(CODE[j]);
end;

{##### Shift NS register South }
procedure SOUTH( REGx: byte);
begin
for j := 0 to 10 do REG[regx, j] := REG[regx, j+1];
REG[regx,11] := 0;
end; {proc}

{##### Move Zeros -> Register }
procedure ZERO( REGx: byte);
begin
for j := 0 to 11 do REG[regx, j] := 0;
end; {proc}

{##### Shift EW register east. }
var MD1, MD2: integer;
begin
if rotate then begin { bit 12 -> 13 ; 144 -> 1 }
MD2 := REG[3,11] mod 2;
for j := 0 to 11 do begin
MD1 := REG[3,j] mod 2;
MD2 := MD1;
end;
end
else
for j := 0 to 11 do
REG[3,j] := REG[3,j] DIV 2; { East most bits will be lost}
{ West most bits <- 0 }
end; {proc}

{##### Shift EW register west. }
var MD1, MD2: integer;
begin
if rotate then begin { bit 13 -> 12 }
MD2 := REG[3,0] div 2048;
for j := 11 downto 0 do begin
MD1 := REG[3,j] div 2048;
REG[3,J] := (REG[3,j] * 2 + MD2) and $fff;
MD2 := MD1;
end;
else
for j := 0 to 11 do
   {West most bit lost}
   REG[3,J] := (REG[3,j] * 2) and $fff; {East most bit <- 0}
end; {proc}

[##### load Register <- Memory ]
procedure ReadR( REGx: byte);
begin
for j := 0 to 11 do REG[regx, j] := RAM[addr,j];
end; {proc}

[##### Move Register to Register ]
procedure Equal( TARGET, SOURCE: byte);
begin
for j := 0 to 11 do REG[target, j] := REG[source, j];
end; {proc}

[##### Print Memory. Print memory from a BEG location and print a specified amount LEN. One word is 144 bits broken up into twelve 12 bit integers.]
procedure Print_mem;
begin
for j := BEG_Mem to BEG_Mem+LEN_Mem do begin
   write(j:5);
   for k := 0 to 10 do
      write(RAM[j,k]:6);
   writeln(RAM[j,11]:6);
end;
end; {proc}

[##### Print Registers. Print selected registers( CM, NS, EW, CR, SM, CY, BW). Registers are displayed as in the format of the Print_mem. ]
procedure Print_Reg;
begin
for j := 1 to 7 do begin
   x := y and $40;
   y := (y shl 1) and $7f;
   writeln('Reg 0 1 2 3 4 5 6 7 8 9 A B');
end;
y := REGMASK;
for j := 1 to 7 do begin
   writeln('Reg 0 1 2 3 4 5 6 7 8 9 A B');
end;

if $x = \$40$ then begin
  write( Vl[V9[j]]:3, ' ');
  for k := 0 to 10 do
    write(REG[j,k]:6);
  writeln(REG[j,11]:6);
end;
end; {proc}

{##### Graph Memory. Assembler program to map memory on to the video screen}
procedure GRAPH(ram_ofs: integer); external 'graph.com';

{##### Show Gapp Memory. Draws bit number scales (1 - 144) and Gapp memory address numbers (0-90). Then calls GRAPH }
procedure ShowVid; { IBM }

var
  i, j, k, g, h, ii, jj: integer;
  jr: real;
  ram_offst: integer;
begin
{ computer start b800 rows cols byts/col
ibm 320 640 80 }
{ one line - 5 80= 5*16 }
{ assign a pointer segment & offset}
if not hirs_flg then begin
  HiRes;
  HiResColor(Green);
  hirs_flg := true;
  i := 10; h := 9; jj := 5; g := 5;
  while (i < 144) do begin
    for j := h to h+5 do plot(g,j,1); { plot(x,y) }
    gotoxy(jj,1); write(i);
    jj := jj + 5;
    i := i + 10;
    g := g + 40;
  end;
  gotoxy(l,1); write(' GAPP ');
  gotoxy(76,3); write('RAM ');
  i := 36; g:= 10; h := 582; jr := 5;
  while (i < 180 ) do begin
    for j := h to h+5 do plot(j,i,1);
    jj := trunc(jr); gotoxy(75,jj); write(g); g := g +10;
    jr := jr + 2.6;
    i := i + 20;
  end;
  if Seg(ram) <> DSeg then
    write('d seg not equal. program error');
end;
ram_offst := ofs(ram);
graph(ram_offst);
end; {proc}

{ following procedures used for GIS code interpretation }
{##### Get Parameter. Get two Parameters for Print_Mem. BEG and LEN, then determine their numeric value }
Procedure Get_Param(er:integer);
beg
j := pos(',,CODE);
if j = 0 then ERROR := er
else begin
CS:= copy(CODE, j+1, COL_LEN-j-1);
k := pos (',,CS);
if k = 0 then ERROR := er
else begin
val ( copy(CS, 1, k-1),Beg_Mem,ECODE);
if ECODE <> 0 { no spaces }
val ( copy(CS, k+1, 10), Len_Mem,ECODE);
if (ECODE <> 0) or (Len_Mem = 0) then ERROR := er;
end;
end;
end; {proc}

{ ACS is a stack for * GIS control commands such as IF THEN }
{ This is a 'first in first out stack }
{##### Push phrase on ACS stack }
Procedure Pushc(cp: str);
beg
ACS[acs_i] := CP;
acs_i := acs_i + 1;
EMPTY_CS := false;
end;

{##### PoP phrase off ACS stack}
Procedure Popc(var cp: str);
beg
{ Empty_CS = true if no more in stack, last one has been poped
if pop while empty then a null string is returned }
acs_i := acs_i -1;
if acs_i = 0 then begin
ERROR := 31;
CP := ' ';
end
else begin
CP := ACS[1];
if acs_i = 1 then
   EMPTY_CS := true
else for g := 1 to acs_i-1 do ACS[g] := ACS[G+1];
end;
end;
Function SEARCH(cp: str): integer;

var cs4: string[4];
begin
CS4 := CP;
Search := 0;
for e := 1 to var_i-1 do
  if CP = VARIABLE[e] then begin
    Search := DATV_PTR[e];
e := var_i-1;
  end;
end;

{LINA and LINTYA are stacks for nested control statements}
{LINA are line numbers of nested control statements}
{LINTYA are control statement types}

Procedure Pushlf(lin,lin_ty: integer);
begin
  LINA[li] := lin;
  LINTYA[li] := lin_ty;
  li := li + 1;
  EMPTY_IF := false;
end;

Procedure Poplf(var lin,lin_ty: integer);
begin
  li := li - 1;
  if li = 0 then begin
    EMPTY_IF := true;
    lin := 0;
    end
  else begin
    lin := LINA[LI];
    lin_ty := LINTYA[li];
    end;
end;

Procedure Operad(opn_i:byte);
begin
  j := Search(CS);
  if j = 0 then begin
    val(CS, K, ECODE);
    if ECODE <> 0 then ERROR := 21;
    DATV[FR_DATV] := K;
    OPNUM[opn_i] := FR_DATV;
    FR_DATV := FR_DATV + 1;
  end;
end
else
  OPNUM[opn_i] := j;
end;  { proc}

{##### Get value of Parameters}
Procedure Param2(OPADD: byte);
begin
  POPC(CS);
  val(CS, K, ECODE);
  if ECODE <> 0 then ERROR := 21;
  OPNUM[3] := K;
  POPC(CS);
  val(CS, K, ECODE);
  if ECODE <> 0 then ERROR := 21;
  OPNUM[4] := K;
end; {proc}

{##### Comparison statement }
Procedure Compare;  { for IF, WHILE }
begin
  POPC(CS);
  Operad(2);
  POPC(CS);
  j := pos(CS, OPERATORS);
  if j = 0 then ERROR := 20
  else begin
    OPNUM[1] := (j-2) div 2;
    POPC(CS);
    if CS = 'RAM' then Param2( 20)  { 127, 11 }
    else if CS = 'REG' then Param2( 30)  { 7, 11 }
    else begin
      Operad(3);
    end;
  end;  {else}
end;  {proc}

{##### Assignment statement }
Procedure Assign_k;
begin
  OPNUM[1] := 40;  { constant assignment}
  H := Search(CS);
  if H = 0 then begin
    VARIABLE[var_i] := CS;
    DATV_PTR[var_i] := FR_DATV;
    var_i := var_i + 1;
    H := FR_DATV;
    FR_DATV := FR_DATV + 1;
  end;
Popc(CS);
if CS <> ':' then ERROR := 25
else begin
    Popc(CS);
    Operad(3);
    end;
end; {proc}

Procedure Next_line;
begin
    for e := 1 to KA do
        OPA[line, e] := OPNUM[e];
    line := line + 1;
end; { proc}

BEGIN
    { Read in GIS program name }

SOURCEFILE := 'A:source.gap';
    { Default GIS program name }
Repeat
    Assign(SOURCE, SOURCEFILE);
    {I-} reset(SOURCE); {I+}
i := IOResult;
    if I <> 0 then Begin
        writeln('Enter name of gapp program input file');
        readln(SOURCEFILE);
        END;
until (I = 0);
    {zero out memory RAM}

for i := 0 to 11 do for j := 0 to 127 do
    RAM[j, i] := 0;
    { Read in GAPP memory load file name}

RamLoadFile := 'a:ram.gap';
    { Default Ram load file name }
Repeat
    Assign(MER, RamLoadFile);
    {I-} reset(MER); {I+}
i := IOResult;
    if I <> 0 then Begin
        writeln('Enter name of Memory (RAM) initial load file');
        readln(RamLoadFile);
        END;
until (I = 0);
    { Read the Memory file and load RAM }
i := 0;
while (i < 128 ) and not(EoF(MER)) do begin
    { EOF does a look ahead }
    read(MER, MA); write(ma);
    if MA <> '"' then begin  { possible set memory address of data to load}
        Read(MER, addr); write(addr);
i := addr;
end;
for j := 0 to 10 do begin
  read(MER, RAM[i,j]);
  write(': ',RAM[i,j]);
end;
readln(MER, RAM[i,11]);
write( ' : ' ,RAM[i,11] ) ;
i := i + 1;
writeln(' ');
end; {while}

{ initialize }
acs_i := 1;
EMPTY_CS := true;
li := 1;
EMPTY_IF := true;
var_i := 1;
DOLINE := true;
ERROR := 0;
PARN_VAL := '\'';
LINE := 1;
FR_DATV := 1;
hirs_flg := false;
rotate := false;

{ echo to screen memroy data }

{ ReadUpCs; }
while ( CODE <> 'END;' ) and ( ERROR = 0 ) do begin
  K := 1;
  fspace := true;
PARN := false;
  CS := '\'';
tar := 0;
  tar := 0;

  { test for blank line }
  FST_CHR := CODE[1];
  if (COL_LEN = 0) and (CODE[1] <> ' ') and (length(CODE) <> 0)
    then ERROR := 9;
  if (COL_LEN <> 0) then begin { skip if blank line}
    for i := 1 to ka do OPNUM[i] := O; { set opcodes to no-op ?? }
    i := 1;
    while (i <= COL_LEN) do begin
      c := CODE[i];
      if FST_CHR <> '*' then begin
        case c of
          '\', '\', ';':
            begin
              { Take apart GAPP instructions }
              begin
                OPNUM[tar] := 0;
                for e:= 1 to V3[tar] do
                  if (cs = V1[V5[tar,e]]) then OPNUM[tar] := e;
                if (OPNUM[tar] = 0) then ERROR := 3;
                if tar <> 5 then OPNUM[tar] := OPNUM[tar] -1;
                { if RAM = register}
              tar := 0;
            end;
          case c of
            CM NS EW C RAM Addr
        end;
      end;
    end;
  end;
end; {ReadUpCs; }

cs := ''; 
k := 1;
end;
';':begin
end;
'A'..'Z': begin
    cs := cs + c;
k := k + 1;
end;
'0'..'9':begin
    k := k + 1;
    if (c = '0') or (c = '1')
        then CS := CS + C
    else ERROR := 4;
end;
begin
    if ( CODE[i+1] <> '=' ) then error := 1;
end;
'=': begin
    for e := 1 to 5 do
        if cs = v1[e]
            then tar := e;
        if tar = 0
            then ERROR := 2;
    CS := '';
end;
')': begin
    if PARN then ERROR := 5;
PARN := true;
k := k + 1;
end;
')': begin
    k := k + 1;
    if PARN then begin
        PARN := false;
        ECODE := 0;
        val ( PARN_VAL, PARN_NUM, ECODE);
        if ECODE = 0
            then if (PARN_NUM > 127) or (PARN_NUM < 0)
                then begin
                    ERROR := 6;
                    write(PARN_NUM, ' ');
                end
            else begin
                PARN_VAL := '';
                if 'RAM' = copy(CS,1,3)
                    then cs := 'RAM';
                end
            else ERROR := 7;
end {if Parn}
else ERROR := 5;
end;
'$': begin
  { special output commands }
  { Print Mem, Print Reg, Show Mem, Rotate}
  if pos('PRINT', CODE) <> 0 then begin
    if pos('MEM', CODE) <> 0 then begin
      OPNUM[1] := 4;
      Get_Param(11);
    end;
    if pos('REG', CODE) <> 0 then begin
      OPNUM[1] := 5;
      j := pos(',', CODE);
      if j = 0 then ERROR := 12
        else begin
        CS := copy(CODE, j+1, COL_LEN-j-1);
        val(CS, REGMASK, ECODE);
        if (ECODE <> 0) or (not REGMASK in [1..127])
          then ERROR := 12;
        end;
    end;
  end;
else { graphicly print out memory RAM }
  if (pos('SHOW', CODE) <> 0) and (pos('MEM', CODE) <> 0) then
    OPNUM[1] := 6
  else if pos('ROTATE', CODE) <> 0 then {Rotate }
    OPNUM[1] := 7
  else
    ERROR := 10;
i := COL_LEN;
end;
else
  ERROR := 8; { no statement = error }
end; { case }
end { if ' *' }
else if i <> 1 then
  { if end, if else end, for end; while end}
case C of
' ;', ' ', '[', ']', ' ', ': begin
  if (CS <> '') and Fspace then Pushc(CS);
  FSPACE := false;
  CS := ' '
end;
' -', '*', '+', '<', '>', ' =', ':': begin
  if Fspace then Pushc(CS);
  Fspace := false;
  if CODE[i+1] in ['>', '='] then begin
i := i + 1;
CS := C + CODE[i];
end
else
CS := C;
Pushc(CS);
CS := ' ';
end;
if c in ['A'..'Z','0'..'9'] then begin
CS := CS + C;
Fspace := true;
end
else ERROR := 8;
end; { case c}

if ERROR <> 0 then begin { get out of while loop if error }
icol := i;
i := COL_LEN;
end
else i := i + 1;
end; { while }

if (FST_CHR = '*') and (ERROR = 0) then begin
quit := false;
Popc(CS);
if CS = 'IF' then begin { I F }
Compare;
if ERROR = 0 then begin
Popc(CS);
if CS <> 'THEN' then ERROR := 22;
if EMPTY_CS then Pushlf(line, 1)
else ERROR := 23;
end;
end { IF}
else if CS = 'WHILE' then begin { W H I L E }
Compare;
if ERROR = 0 then begin
Popc(CS);
if CS <> 'DO' then ERROR := 24;
if EMPTY_CS then Pushlf(line, 3)
else ERROR := 23;
end;
end { if}
else if CS = 'FOR' then begin { F O R }
Popc(CS); Assign_k;
Next_Line;
Popc(CS);
if CS <> 'TO' then ERROR := 27;
if ERROR = 0 then begin
OPNUM[1] := 18;                         { < }  
OPNUM[2] := h;  
Popc(CS);  
Operad(3);  
PushIf(line, 4);  
Popc(CS);  
if CS <> 'DO' then ERROR := 24;  
if not EMPTY_CS then ERROR := 23;  
end;  
end { else if}  
else if CS = 'ELSE' then begin        { ELSE}  
PopIf( lin_x, OpTyp);  
if EMPTY_IF then ERROR := 28  
else if Optyp <> 1 then ERROR := 30  
else begin  
PushIf(line, 1);  
OPNUM[1] := 50; {goto}  
OPA[lin_x, 6] := line + 1;  
end;  
end  
else if CS = 'INC' then                { INC}  
OPNUM[1] := 52  
else if CS = 'DEC' then                { END }  
OPNUM[1] := 53  
else if CS = 'END' then begin          { and else is an implied end }  
{ and else is an implied end }  
if EMPTY_CS then begin  
PopIf(lin_x, OpTyp);  
if EMPTY_IF then ERROR := 28  
else  
case OpTyp of  
1: begin  
DOLINE := false;  
OPA[lin_x, 6] := line;  
end;  
3: begin  
OPA[lin_x, 6] := line + 1;  
OPNUM[1] := 50; { goto}  
end;  
4: begin  
OPNUM[1] := 41; { assign}  
OPNUM[2] := OPA[lin_x, 2];  
DATV[FR_DATV] := 1;  
FR_DATV := FR_DATV + 1;  
Next_line;  
OPNUM[1] := 50; { goto}  
OPA[lin_x, 6] := line + 1;
end; {4} 
end; {case} 
end { if} 
else ERROR := 29; 
end {else if} 

else begin 
Assign_k; { assignment statement } 
if (ERROR = 0 ) and not EMPTY_CS then begin 
writeln(' empty_cs',empty_cs); 
Popc(CS); 
C := CS; 
case C of 
  '+' : OPNUM[1] := 41; {addition } 
  '-' : OPNUM[1] := 42; { sub } 
  '*' : OPNUM[1] := 43; { mult } 
  '/' : OPNUM[1] := 44; { div } 
else ERROR := 20; 
end; 
Popc(CS); 
Operad(4); 
end; {if} 
end; {else} 

end; 

if ERROR = 0 then begin 
  if DOLINE then Next_Line; 
  DOLINE := true; 
end; { if for blank line} 

ReadUpCs; 
{ END OF FILE check needed } 
end; { while} 

if ERROR <> 0 then Errorp(ERROR) 
else begin 
L I N E := LINE -1; 
for i := 1 to LINE do begin 
  writeln( ' ' ); 
  write(i,': ' ); 
  for j := 1 to ka do 
    write(OPA[i,j],' ' ); 
  end; {i} 

i := 1; 
while i <= LINE do begin 
  for e := 1 to KA do 
    OPNUM[e] := OPA[i, e]; 
  ADDR := OPNUM[6]; 
end;
case OPM(1) of
 4:  Print_Mem;
 5:  Print_Reg;
 6:  ShowVidi;
 7:  rotate := true;
14:  if DATV[OPM(2)] <> DATV[OPM(3)] then INC_i := false;
15:  if DATV[OPM(2)] <= DATV[OPM(3)] then INC_i := false;
16:  if DATV[OPM(2)] >= DATV[OPM(3)] then INC_i := false;
17:  if DATV[OPM(2)] = DATV[OPM(3)] then INC_i := false;
18:  if DATV[OPM(2)] > DATV[OPM(3)] then INC_i := false;
19:  if DATV[OPM(2)] < DATV[OPM(3)] then INC_i := false;
24:  if DATV[OPM(2)] <> RAM[ OPM(3), OPM(4)]
        then INC_i := false;
25:  if DATV[OPM(2)] <= RAM[ OPM(3), OPM(4)]
        then INC_i := false;
26:  if DATV[OPM(2)] >= RAM[ OPM(3), OPM(4)]
        then INC_i := false;
27:  if DATV[OPM(2)] = RAM[ OPM(3), OPM(4)]
        then INC_i := false;
28:  if DATV[OPM(2)] > RAM[ OPM(3), OPM(4)]
        then INC_i := false;
29:  if DATV[OPM(2)] < RAM[ OPM(3), OPM(4)]
        then INC_i := false;
34:  if DATV[OPM(2)] <> REG[ OPM(3), OPM(4)]
        then INC_i := false;
35:  if DATV[OPM(2)] <= REG[ OPM(3), OPM(4)]
        then INC_i := false;
36:  if DATV[OPM(2)] >= REG[ OPM(3), OPM(4)]
        then INC_i := false;
37:  if DATV[OPM(2)] = REG[ OPM(3), OPM(4)]
        then INC_i := false;
38:  if DATV[OPM(2)] > REG[ OPM(3), OPM(4)]
        then INC_i := false;
39:  if DATV[OPM(2)] < REG[ OPM(3), OPM(4)]
        then INC_i := false;
40:  DATV[OPM(2)] := DATV[OPM(3)];
41:  DATV[OPM(2)] := DATV[OPM(3)] + DATV[OPM(4)];
42:  DATV[OPM(2)] := DATV[OPM(3)] - DATV[OPM(4)];
43:  DATV[OPM(2)] := DATV[OPM(3)] * DATV[OPM(4)];
44:  DATV[OPM(2)] := DATV[OPM(3)] div DATV[OPM(4)];
50:  INC_i := false;
      { goto }
52:  OPA[i-1, 6] := OPA[i-1,6] + 1;
      { inc mem }
53:  OPA[i-1, 6] := OPA[i-1,6] - 1;
      { dec mem }
else begin
    { CM = } case OPM(1) of
      {RAM} 1:  ReadR(1);
      {CMS} 2:  South(1);
      { 0 } 3:  Zero(1);
    end;
{ NS = }
case OPNUM[2] of
  {RAM} 1: ReadR(2);
  {N } 2: begin
    REG[2, 1] := 0;
  end;
  {S } 3: South(2);
  {EW } 4: Equal(2,3);
  {C } 5: Equal(2,4);
  {O } 6: Zero(2);
end; { case }
{ EW = }
case OPNUM[3] of
  {RAM} 1: ReadR(3);
  {E } 2: East;
  {W } 3: West;
  {NS } 4: Equal(3,2);
  {C } 5: Equal(3,4);
  {O } 6: Zero(3);
end;
{ C = }
case OPNUM[4] of
  {RAM} 1: ReadR(4);
  {NS } 2: Equal(4,2);
  {EW } 3: Equal(4,3);
  {CY } 4: Equal(4,6);
  {BW } 5: Equal(4,7);
  {O } 6: Zero(4);
  {1 } 7: for j := 0 to 11 do REG[4, j] := 4095;
end;
{ RAM = reg }
k := OPNUM[5];
if k <> 0 then { not a read }
  for j := 0 to 11 do RAM[addr, j] := REG[v8[k], j];
for j := 0 to 11 do begin { do value for SM, CY, BW}
  NS := REG[2, j];
  EW := REG[3, j];
  CR := REG[4, j];
  NNS := NS xor $fff; {not ns}
  NEW := EW xor $fff; {not ew}
  {SM} REG[5,j] :=(NNS and (CR xor EW)) or (NS and (CR xor NEW));
  {CY} REG[6,j] := (NS and EW) or (CR and (EW or NS));
  {BW} REG[7,j] := (NNS and (EW or CR)) or (EW and CR);
end;
end; { else of case}
end; { case }
if INC_i then i := i + 1
  else i := ADDR;
INC_i := true;
end; { while i}
end; { else for error}
end.
; assembler code for display of GAPP memory for display on
; screen
;
code segment
assume cs:code

graph proc near
    mov cs:ssbp, bp
    mov bp, sp
    mov bx, [bp+2] ; get parameter value
    mov cs:ses, es
    mov cs:sssi, si
    mov cs:ssdi, di
    mov ax, 0B828h ; screen
    mov es, ax ; screen
    sub di, di ; es:di screen
    mov si, 90

    17: mov cx, 12
    19: mov ax, [bx] ; get RAM value
        jmp kk

    kf: add bx, 2
        loop 19
        dec si
        jnz 17

    ;
    mov sp, bp
    mov es, cs:ses
    mov bp, cs:ssbp
    mov si, cs:sssi
    mov di, cs:ssdi
    ret 2

    ;
    ses dw ?
    ssdi dw ?
    sssi dw ?
    ssbp dw ?

    kk: sub dx, dx
        rcr ah, 1
        jnc yy1
        add dx, 0f00h

    yy1: rcr ah, 1
        jnc yy2
        add dx, 0f000h

    yy2: rcr ah, 1
        jnc yy3
        add dx, 0fh

    yy3: rcr ah, 1
        jnc yy4
        add dx, 0fh
yy4:  mov   es:[di],dx
    add   di,2
;
   sub   dx,dx
   rcl   al,1
   jnc   ytl
   add   dx,0f0h
ytl:  rcl   al,1
   jnc   yt2
   add   dx,0fh
yt2:  rcl   al,1
   jnc   yt3
   add   dx,0f000h
yt3:  rcl   al,1
   jnc   yt4
   add   dx,0f000h
yt4:  mov   es:[di],dx
   add   di,2
;
   sub   dx,dx
   rcl   al,1
   jnc   yh1
   add   dx,0f0h
yh1:  rcl   al,1
   jnc   yh2
   add   dx,0fh
yh2:  rcl   al,1
   jnc   yh3
   add   dx,0f000h
yh3:  rcl   al,1
   jnc   yh4
   add   dx,0f000h
yh4:  mov   es:[di],dx
   add   di,2
   jmp   kf
graph endp
code ends
end
;  asm,  link
;  exe2bin  graph.exe  graph.com
;  del  graph.exe
REFERENCES


