MEETING NOTICE

The next meeting will be Monday, December 19, at 7:30. The meeting location is Alpha Audio's third floor conference room, at 2049 West Broad Street. The night-time phone number there is 358-3853. The front door has a touch-pad combination lock, and the combination for the night will be 7603 (seven six zero three).

The program will be a presentation on the C language, by Carlos Chafin.

Everyone is welcome!

MINUTES (Meeting of November 21, 1983)

Present: Carlos Chafin, Charlie Gaines, David Harrington, Harold Lanna, John Purcell, Jim Scott, Ron Stauffer, Jerry Tiller, Nelson Trinkle and Parks Watson. Also present were John Parker and Bill Sprinkle, to whom we extend a cordial invitation to join our group.

The first order of business was the matter of a proposed dues increase, sufficient to cover the cost of duplicating and mailing the GAZETTE. It was estimated that, depending on the number of pages in each issue, whether condensed or full-size type is used, if and when first-class postage is increased and the number of copies of each issue, the annual cost could range from $9.00 to $15.00 per member. In view of the number of indeterminate variables involved, the concensus was that a median figure would be appropriate. It was moved, seconded and the motion carried, that dues for the calendar year 1984 be $12.00 per member.

Nomination of officers for 1984 was next on the agenda. After much passionate oratory by the respective candidates, the following slate was adopted:
Election of officers will take place at the December meeting. Additional nominations from the floor will be accepted at that time.

It was noted that REMark plans to publish a directory of local HUG groups in the January issue. In view of the November 15, deadline (see note below) for submitting updated information, it was decided that we would move up our election of officers to October, with nominations in September, in order to meet the above deadline. Officers would still take office on the following January.

The program for December will be a presentation on C language programming by Carlos Chafin. Carlos’ talk was scheduled for the October meeting but had to be postponed due to unforeseen circumstances.

The meeting was adjourned at 9:15 PM.

Parks Watson
Secretary/Treasurer

[NOTE: Reference to REMark Iss. 45, page 53, confirmed my recollection that the deadline for submission of data for the January issue was December 1, instead of November 15. This would allow us to have election of officers in November and still meet this later deadline. We might want to reconsider the action noted in the minutes above. - Parks Watson]

******************************************************************************

NEWS

RHUG ELECTIONS IN DECEMBER

At our December meeting we will elect officers for the upcoming year. See the minutes (above) for details.

+++++++

RHUG DUES DUE IN DECEMBER

Our dues are now $12.00 per calendar year. This amount is intended to cover the cost of duplicating and mailing the Gazette for 1984. To make sure you stay on the mailing list, get square with RHUG before the next issue is mailed in early January.

******************************************************************************

TRIONYX T-H90 MOTHERBOARD
by Jerry Tiller

Well, I’m back again. You didn’t know I’d left? Yes I did. I am here to tell you about the new Trionyx motherboard for the H-8.
As I said in my last article I ordered a Trionyx motherboard. Well, it arrived today (Aug. 3). (Notice how I spelled "Trionyx"? It's correct this time.)

First, let me tell you, it is not cheap. The motherboard is sold in two assembled versions. An H-8 version for 175.00 which exactly duplicates the connector layout of the Heath motherboard and a Trionyx version 250.00 which adds 7 new buss slots plus expands the original buss to 90 pins. The kit prices are cheaper for the brave one. The H-8 equivalent is 136.00 and the Trionyx version is 205.00. Not a whole lot of savings.

For those of you who have an H/Z-89/90 or who have a late vintage H-8 with gold connectors let me explain the reason for changing the old motherboard in the H-8. The early versions of the H-8 motherboards were equipped with tin plated connectors. So what, you say. Well, these tin connectors are the source of many unexplained and mysterious crashes in the middle of a program. It became a ritual to periodically remove all of the plug-in boards and clean the connectors. The tin connectors would build up an oxide film which would not allow the contacts on the plug-in boards to make good contact with the motherboard. Hence, intermittent failures.

This problem became widely known and much talked-about among H-8 owners. Heath Co.'s answer was to make new motherboards for all new computers they were selling. They did not offer the gold connectors to old H-8 owners nor would they even acknowledge the problem. So Trionyx developed the T-H90 for the H-8.

Trionyx had other motives for making a new motherboard. They were developing a Z-80 board for the H-8 which will run at 4MHZ. The existing buss structure of the H-8 and the ground plane would not operate at the higher clock speeds. The Z-80 board is a reality now with a 16-bit 8086 CPU board coming right behind it. (Look out, Z-100.)

Enough of this, on with the show.

I purchased the fully assembled version of the motherboard so I would not have to go back later and expand it to accommodate the new boards being developed.

A close examination of the board after unwrapping it revealed a very well-constructed product. The foil traces and the solder work were top quality. A large package of documentation accompanied the motherboard detailing its design and operating specifications. There weren't any assembly instructions to be found except for a sheet of paper saying the H-8 would have to be extensively disassembled to install the new board. This was not an oversight since the instructions for assembly are already in the H-8 construction manual.

Disassembling the H-8 proved to be a fairly easy task. They were right, you really have to take it apart. You must remove all of the circuit boards including the front panel board. Once all of this is done you remove the protective cover over the filter capacitor in the back. The next step is to remove the screws holding the old motherboard in place and turn the board over to get to the solder connections on the bottom. Unsolder the 5 leads attached to the old motherboard and move them back out of the way.
Remove the 4 screws holding the side panel in place and set the rest of the H-8 aside. Remove the 6 threaded hex spacers from the side panel and put them away. You must do this because the new motherboard will not fit properly in the cabinet. Replace the spacers with new nylon spacers. The spacers are not included. I had to purchase them separately. With the new spacers installed, remount the side panel to the cabinet. Install the wires in their proper place (same locations as on the old board) and solder them in place.

Mount the new motherboard with the screws and fiber washers supplied in the package. Install the filter cover in its original place. Plug the H-8 in and check for the proper voltages on the buss. Once this is done you can install the circuit boards back in their original connector locations. Slots 11 through 17 are used for some of the new boards being developed and the old circuit boards will not work in them. A peek at the pin-out diagram of the new motherboard will show that not all of the buss lines are not connected on all of those plug-in slots.

Once everything is back together, plug it in, turn it on, and begin computing with the knowledge that there should be no more middle-of-the-program crashes. If you still have them then you probably have a bad circuit board somewhere else in the system.

Total assembly and disassembly time was two hours. Not bad when you consider how far you have to take it apart.

My H-8 has been running steady for the last 24 hours without a single beep or crash. It wouldn’t run any longer than an hour or two before going off into never-never land and destroying my program. Before I was having to back everything up every thirty minutes in order not to lose it all when it crashed. Now I don’t have to do that any more.

Well, you can’t say I didn’t warn you. I told you I’d be back. Who knows? Maybe I’ll have another piece of hardware to write about next month. (Hey Kay! Can I buy a Z-80 board?)

Hasta Luigi my Friends.

**********************************************************************

ASSEMBLY LANGUAGE PROGRAMMING - PART 6
by Jim Scott

INTRODUCTION

This is the sixth of a series of articles which parallel and summarize the discussions about assembly language at our meetings. The purpose of the discussions and the articles is to present enough information about assembly language programming so that someone who knows how to program in a higher-level language, and is willing to use the proper manuals for reference, will at least have some idea how to get started at programming in assembly language.

The previous three articles (August, September, and October issues of the Gazette) described the Data Transfer, Arithmetic, and Logical groups of instructions for the 8080 CPU. Instructions in these groups move data between registers and memory, and perform arithmetic and logical operations on data in registers and memory. This time we will discuss the Branch Group of instructions.
BRANCH GROUP

Instructions in this group alter the normal sequence of program flow.

What we mean by "the normal sequence of program flow" is that after the CPU executes an instruction, it will ordinarily then execute the very next instruction in memory. Article 2 in this series (see the July issue of the Gazette) put it like this: The Program Counter (PC) "contains the address of the memory location which contains the next instruction to be executed by the computer. Most instructions simply add their instruction length (one, two, or three bytes) to the PC, so that the instructions are executed in sequence. Instructions such as JMP, CALL, RET, and PCHL [i.e., instructions in the Branch Group] can put other values into PC, to alter the order with which the instructions are executed."

When an instruction alters the PC so that the normal sequence of program flow is not followed, we say that the program "branches," or "takes a branch." An instruction which does this, or may do it depending on some condition, is called a "branch instruction". It may branch "backward" (to a lower address than that of the branch instruction), "forward" (to a higher address), or even to the branch instruction itself (which would presumably cause a "tight loop" that you could only get out of by resetting the computer). If the location branched to does not contain a valid instruction, of course, there will be problems.

The branch instructions allow us to have program loops, IF statements, and subroutines and functions. They allow our programs to make decision and to react differently to different inputs. Without the branch instructions, our computers would be pitiful tools; each program would execute straight through and quit immediately. If you wanted to write a program to open a disk file, read and process 100 records, and close the file, you would have to write the read-and-process part of the program 100 times.

None of the branch instructions affect any condition flags. However, some of the branch instructions are conditional, meaning that they may or may not change the PC to make the program take a branch, depending on the value of one of the condition flags. The mnemonic operation code for a conditional branch instruction includes an abbreviation (represented by "cc" in the descriptions of the individual instructions) of the condition being tested for. The abbreviations are as follows:

- NZ not zero (Z = 0)
- Z zero (Z = 1)
- NC no carry (Cy = 0)
- C carry (Cy = 1)
- PO parity odd (P = 0)
- PE parity even (P = 1)
- P plus (S = 0)
- M minus (S = 1)

where Z, Cy, P, and S are the Zero, Carry, Parity, and Sign flags, respectively, in the Flag Word register. As you may remember from the previous two articles, these flags are usually set by instructions in the Arithmetic and Logical groups.
For example, JZ means Jump If Zero, i.e., take a branch if the Zero flag has been set to 1. Generally, a conditional branch instruction immediately follows an instruction, typically a compare instruction, which will set the flags.

The individual instructions are as follows.

**JMP (Jump)**

**Format:** JMP addr

where addr is a two-byte address of a memory location. The address specified is put into the PC, thus causing the instruction at that address to be the next one executed.

**Example:** JMP 331CH

This stores the hex address 331C into the PC. The next instruction executed will be the one at 331C.

This instruction does a simple unconditional branch. It is the equivalent of the GOTO instruction in many high-level languages.

**Jcc (Conditional Jump)**

**Format:** Jcc addr

where addr is a two-byte address of a memory location, and cc is one of the conditions listed above. If the specified condition is true, the address specified is put into the PC, thus causing the instruction at that address to be the next one executed; otherwise, control continues sequentially (3 is added to the PC).

**Example:** JNC SKIP

If the Carry flag is 0 (i.e., if the latest instruction that affected the Carry flag did not result in a carry), this instruction stores the address equivalent to the assembler label SKIP into the PC; the next instruction executed will be the one at SKIP. If the Carry flag is 1, the next instruction executed will be the one immediately after the JNC SKIP instruction.

This instruction does a conditional branch. It is usually immediately preceded by an arithmetic or logical instruction, which will determine the value of the flags. Then the Jcc instruction either branches or does not branch, depending on the value of one of the flags. This is the equivalent of a simple IF...THEN GOTO... statement. Here are some examples:

```
  CPI 48H ;DOES REGISTER A CONTAIN A HEX 48?
  JZ RCKA ;IF SO, JUMP TO RCKA.
```

(Remember that a compare instruction sets the Zero flag to 1 if the two operands being compared are equal.)

```
  SUI 1 ;SUBTRACT 1 FROM THE VALUE IN REGISTER A.
  STA NUMSPAC ;SAVE RESULT IN MEMORY LOCATION NUMSPAC.
  JNZ OUTSPAC ;IF RESULT OF SUBTRACTION WAS NOT ZERO, JUMP TO OUTSPAC.
```

(Remember that a STA instruction, like all of the instructions in the Data Transfer Group, does not affect the flags.)

```
  CPI 81H ;IS VALUE IN REGISTER A LESS THAN A HEX 81?
  JC NOEXP ;IF SO, JUMP TO NOEXP.
```
(Remember that a compare instruction sets the Carry flag to 1 if the contents of register A are less than the contents of the other operand.)

ANI 0FFH ;AND THE REGISTER A WITH HEX FF (WOULD NOT ALTER REG A).
JPO ODDA ;IF RESULT (= ORIGINAL VALUE) HAS ODD PARITY, JUMP.

CALL (Call)

Format: CALL addr
where addr is a two-byte address of a memory location. The address of the next sequential instruction (the one after the CALL instruction) is pushed onto the stack, and the address specified (by the addr operand) is put into the PC, thus causing the instruction at that address to be the next one executed. (For details of what it means to push something onto the stack, see the PUSH instruction in the next article in this series.)

Example: CALL MULT
This pushes onto the stack the address of the instruction immediately following the CALL instruction, then branches to the instruction at MULT.

This instruction is used to invoke a subprogram (i.e., a subroutine or a function - the distinction is not important here) in such a way that the subprogram, when it is finished, can branch back to the instruction following the CALL instruction. If we used JMP rather than CALL, the subprogram would have no way of knowing the whereabouts of the instruction that branched to it, so it would also not know where to branch back to when it was finished. (Typically, a subprogram is invoked from more than one point in a program.) The difference here is that the address of the instruction to branch back to is on the stack, where the subprogram can find it. The subprogram can take advantage of this information by using the RET or Rcc instruction (see below) to return (branch back to) the instruction following the CALL (or Ccc) instruction.

Ccc (Conditional Call)

Format: Ccc addr
where addr is a two-byte address of a memory location, and cc is one of the conditions listed above. If the specified condition is true, the address of the next sequential instruction (the one after the Ccc instruction) is pushed onto the stack, and the address specified (by the addr operand) is put into the PC, thus causing the instruction at that address to be the next one executed; otherwise, control continues sequentially (3 is added to the PC), and the stack is not affected. (For details of what it means to push something onto the stack, see the PUSH instruction in the next article in this series.)

Example: CZ CLEAR
If the Zero flag is 1 (i.e., if the latest instruction that affected the Zero flag had a zero result, or compared two equal operands), this instruction pushes the address of the following instruction onto the stack, and stores the address equivalent to the assembler label CLEAR into the PC; the next instruction executed will be the one at CLEAR. If the Zero flag is 0, the next instruction executed will be the one immediately after the CZ CLEAR instruction, and the stack will not be affected.
This instruction does a conditional CALL. It is usually immediately preceded by an arithmetic or logical instruction, which will determine the value of the flags. Then the Ccc instruction either does or does not perform a CALL, depending on the value of one of the flags. This instruction is used to invoke a subprogram in such a way that the subprogram, when it is finished, can branch back to the instruction following the Ccc instruction. If we used Jcc rather than Ccc, the subprogram would have no way of knowing the whereabouts of the instruction that branched to it, so it would also not know where to branch back to when it was finished. (Typically, a subprogram is invoked from more than one point in a program.) The difference here is that the address of the instruction to branch back to is on the stack, where the subprogram can find it. The subprogram can take advantage of this information by using the RET or Rcc instruction (see below) to return (branch back to) the instruction following the Ccc (or CALL) instruction.

RET (Return)

Format: RET
This instruction pops an address off of the stack and puts it into the PC, thus causing the instruction at that address to be the next one executed. (For details of what it means to pop something off of the stack, see the POP instruction in the next article in this series.)

Example: RET
This branches to the address popped off of the stack.

This instruction is used to return from a subprogram or system routine that was invoked by a CALL, Ccc, or RST instruction. Thus, it branches back to the instruction following the CALL, Ccc, or RST instruction.

Rcc (Conditional Return)

Format: Rcc
where cc is one of the conditions listed above. If the specified condition is true, this instruction pops an address off of the stack and puts it into the PC, thus causing the instruction at that address to be the next one executed. (For details of what it means to pop something off of the stack, see the POP instruction in the next article in this series.)

Example: RP
If the Sign flag is 0 (i.e., if the latest instruction that affected the Sign flag had a positive result), this instruction pops an address off of the stack, and stores it into the PC, thus taking a branch to that address. If the Sign flag is 1, the next instruction executed will be the one immediately after the RP instruction, and the stack will not be affected.

This instruction is used to return, conditionally, from a subprogram or system routine that was invoked by a CALL, Ccc, or RST instruction. Thus, it conditionally branches back to the instruction following the CALL, Ccc, or RST instruction.

RST (Restart)

Format: RST n
where n is a number from 0 to 7. The address of the next sequential instruction (the one after the RST instruction) is pushed onto the
stack, and the value 8*n is put into the PC, thus causing the instruction at that address to be the next one executed. (For details of what it means to push something onto the stack, see the PUSH instruction in the next article in this series.)

Example: RST Ø5H
This pushes onto the stack the address of the instruction immediately following the RST instruction, then branches to the instruction at memory location Ø*5 = 40 (= 28 hex).

This instruction is used to invoke a special system routine in such a way that the routine, when it is finished, can branch back to the instruction following the RST instruction. If we used JMP rather than RST, the system routine would have no way of knowing the whereabouts of the instruction that branched to it, so it would also not know where to branch back to when it was finished. The difference here is that the address of the instruction to branch back to is on the stack, where the system routine can find it. The system routine can take advantage of this information by using the RET or Rcc instruction (see below) to return (branch back to) the instruction following the RST instruction.

The RST instruction can obviously only be used to invoke eight different system routines, because it can only branch to absolute memory location Ø, 8, 16, 24, 32, 40, 48, and 56 (hex Ø, 8, 10, 18, 20, 28, 30, and 38). In practice, these eight locations in low memory usually contain JMP instructions which branch to the real system routines elsewhere in memory. Furthermore, values stored in other registers, before the RST instruction is executed, can specify more details of what processing is being requested.

If you use the MDOS assembler, an SCALL instruction will be assembled as if it were a RST 7 instruction followed by a one-byte number. This is the approved method of making an MDOS system call; the system routine will return to the instruction following the one-byte number (by adding 1 to the address popped off of the stack). With CP/M, the assembler will not recognize the SCALL instruction; this is just as well, because the contents of low memory will be different. CP/M system calls are done with the CALL instruction.

PCHL (Jump HL Indirect)

Format: PCHL
The address in the H-L register pair is put into the PC, thus causing the instruction at that address to be the next one executed.

Example: PCHL
This branches to the instruction whose address is in the H-L register pair.

This instruction is very similar to a JMP instruction, except that the address to be branched to is in HL rather than in the second and third bytes of the instruction. It is not used frequently, but it is the way to go when the address to be branched to is not known at assembly time (e.g., it must be computed, or it must be loaded from a register pair or from a pair of memory locations).

CONCLUSION

The next article will complete the 8080 instruction set, with the Stack, I/O, and Machine Control Group of instructions.