INTERSIL

ROM BASED SUBROUTINE CALLS WITH THE IM6100

INTERSIL CMOS LSI

APPLICATION BULLETIN
M008

INTRODUCTION

Frequently the same or similar sequence of instructions must be executed in different parts of a program. There are obvious advantages to writing a program in which the identical piece of code is written only once and each time it is used in the main part of the program, the program flow is changed to execute the code. The piece of code is called a "subroutine" since it is a subsidiary part of a larger routine or program. After the subroutine has been executed, a transfer of control is made back to the instruction following the transfer to the subroutine. This immediately poses the problem of how the subroutine knows which location to return to since many different parts of the main program make "calls" to the same subroutine.

IM6100 SUBROUTINE CALL

In the IM6100, the JMS, Jump to Subroutine, instruction is used to eliminate the need for writing the complete set of instructions each time an intermediate task must be performed, be it finding a square root or typing a character on the Teletype. Since the IM6100 is designed to be program compatible with the DEC PDP-8/E™ it uses the same convention as the PDP-8 for subroutine linkage which is to store the "return" address in the first location of the called subroutine. After the subroutine code has been executed, a return transfer is made by jumping back "indirectly" through the first location of the subroutine. Thus, the programmer has a simple means of exiting and for returning to the correct location of the program upon completion of the task.

This convention, though extremely simple and straightforward, has two drawbacks, the first drawback being when the user program is stored in read-only memory, ROM, the JMS instruction cannot be used to call a ROM based subroutine since one cannot write into a read-only location to establish the return link. The second drawback is associated with "recursive" subroutine calls. It is quite possible that one subroutine may call another. The IM6100 linkage mechanism is applicable in this case. However, there are instances, when a subroutine may call itself over and over, recursively. Obviously, the simple linkage mechanism will not work since a call to itself will destroy the return address associated with the call immediately preceding it. Although it is possible to design around recursive techniques, recursion is important, in some cases, since it permits a better structured program with less memory when compared with iterative designs.

LINKAGE THROUGH RAM

If one is not interested in recursion, which is true in most instances, ROM based subroutines may be called by providing a RAM entry point for each subroutine. For example, a subroutine in ROM location 6600_8 may be called from location 5013_8 with the linkage mechanism, shown below:

```
                      /CALLING A SUBROUTINE BY LINKING THRU RAM
                      /SUBROUTINE IN LOCATION 6600 (ROM)
                      /EXAMPLE OF BEING CALLED FROM 5013

                      *5013
5013   4170           JMS 0170
                      *0170
0170   0000           0000                /RETURN ADDRESS
0171   5572           JMP I .+1           /ENTER SUBROUTINE THRU
0172   6600           6600                /RAM LOCATION 0170

                              /LOCATIONS 171 & 172 MUST
                              /BE INITIALISED AT POWER ON

                      /EXIT FROM SUBROUTINE
                      *6676               /LAST INSTRUCTION
6676   5570           JMP I 0170          /RETURN VIA 0170
```

Execution times:

	4 MHz	8 MHz
CALL	13μs	6.5μs
RETURN	7.5μ	3.75μs

Memory Overhead for each subroutine in the program:

3 RAM locations in Page Zero, two of which must be initialized at power-on.

6 ROM locations to initialize the two locations in RAM.

™ Trademark of Digital Equipment Corporation, Maynard, MA.

INTERSIL

RETURN STACK

ROM based subroutines, as well as recursion, can be handled through the medium of a pushdown stack or LIFO (Last-in-first-out). Most of the currently available microprocessors put the subroutine return addresses into a stack memory which may be part of the CPU chip or part of the external memory.

When return addresses are stored in an on-chip pushdown stack, there is a natural limit to the number of dynamic subroutines active at any given time. For example, if there are eight stack positions, then, generally, only seven subroutine calls may be active at one time since the real used stack size must be kept smaller to allow some stack depth for interrupt service routines, if any. This, of course, assumes that no processor state information other than the Program Counter need be saved when calling subroutines. If the Accumulator or other status information must be saved, the number of subroutines that may be "simultaneously" active is significantly reduced. The on-chip stack does allow for faster subroutine calls since external memory accesses are kept to a minimum.

Another approach is to maintain a stack pointer in the CPU and to store return addresses in the external read-write memory. When a subroutine is called, the return address is pushed into the RAM stack and the pointer is updated. Stacks in RAM are of potentially huge depth and this allows certain kinds of algorithms to be easily programmed. If the on-chip stack is accessible to the programmer, the depth of the stack can be extended by software. Most on-chip stack manipulations are cumbersome and time consuming, and this imposes a rigid limit on the allowed depth of the subroutine calls. In view of the fact that most microprocessor applications involve some amount of external RAM, the external RAM stack solution is achieving wider acceptance. The microprocessor chip area is also reduced by providing the stack memory externally.

SOFTWARE STACK

The IM6100 architecture provides for the simulation of a stack in software. In the following section we discuss a specific software implementation of a stack oriented subroutine linkage mechanism.

PROGRAM DESCRIPTION

A subroutine is "called" by invoking a supervisory routine, CALL, followed by the entry address of the subroutine. CALL leaves the Program Counter, PC, on a stack, starting at a user defined base. A return from the subroutine is executed with another supervisory routine, RETURN, which implements the linkage back to the main program. The "entry address" which follows CALL is skipped over when returning from the subroutine.

AC, LINK and MQ are not affected. The supervisory routines do not check for stack overflow or underflow. The program is easily modified to save AC or any other processor state information on the stack and since the stack pointer itself is maintained in memory, one can also check for overflow and underflow conditions.

The supervisory routines may be assembled any place in the user program. For illustration purposes, we have assigned arbitrary locations. The user memory is expected to be organized as RAM in the lower pages and ROM in the higher pages. The CALL and RETURN routines use six locations in page zero. Since page zero is directly accessible from any other page, the supervisory routines may be called from any location in memory.

Four of the page zero locations used by the supervisory routines must be initialized when power is turned on. The IM6100 Program Counter is set to 7777_8 when the RESET line is active. The power-on routine, starting at 7777_8, is executed to initialize the user system.

Execution times:

	4 MHz	8 MHz
CALL	70μs	35μs
RETURN	54μs	27μs

Fixed memory overhead for CALL and RETURN:

6 RAM locations in Page Zero, four of which must be initialized at power-on.

29 ROM locations, 17 for routines and 12 for power-on initializing.

Memory overhead for each active call:

1 RAM location for the stack to grow.

PAL III convention:

The symbols CALL and RETURN must be defined in the user program, as shown below:

```
CALL = JMS CALLX
RETURN = JMP I RETX
```

Program listing:

```
                    /SOFTWARE STACK ROUTINES FOR IM6100

                    /RAM LOCATIONS IN PAGE ZERO

                    *162

0162 0000 CALLX,    0000             /ENTRY POINT FOR "CALL" ROUTINE
0163 5564           JMP I .+1        /GO TO "CALL" IN ROM
0164 7400           CALLY            /START OF "CALL" IN ROM

0165 7411 RETX,     RETY             /POINTER TO "RETURN" ROUTINE IN ROM

0166 0170 STACK,    .+2              /CURRENT STACK POINTER. INIT TO
                                     /0170 BY POWER-ON ROUTINE
0167 0000 AC,       0000             /TEMPORARY LOC FOR AC

                    /THE LOCATIONS CALLX+1,CALLX+2,RETX AND
                    /STACK MUST BE INITIALISED AT POWER-ON.


                    /ROM LOCATIONS

                    *7400

7400 3167 CALLY,    DCA AC           /SAVE AC
7401 2166           ISZ STACK        /UPDATE STACK POINTER

7402 1162           TAD CALLX        /CALLX HAS RETURN ADDRESS
7403 7001           IAC              /INCREMENT BY 1 TO SKIP OVER
7404 3566           DCA I STACK      /ENTRY ADDRESS OF USER SUBROTINE
                                     /AND SAVE ON STACK
7405 1562           TAD I CALLX      /GET USER ROUTINE ENTRY ADDRESS
7406 3162           DCA CALLX        /AND PUT IT IN CALLX

7407 1167           TAD AC           /RESTORE AC
7410 5562           JMP I CALLX      /GO TO USER  SUBROUTINE


7411 3167 RETY,     DCA AC  /SAVE AC
7412 1566           TAD I STACK      /GET RETURN ADDRESS FROM STACK
7413 3162           DCA CALLX        /AND PUT IT IN CALLX

7414 7060           CMA CML          /AC=7777; COMPLEMENT LINK
7415 1166           TAD STACK        /STACK POINTER-1; RESTORE LINK
7416 3166           DCA STACK        /UPDATE STACK POINTER

7417 1167           TAD AC           /RESTORE AC
7420 5562           JMP I CALLX      /RETURN


                    *7600
7600 1372 INIT,     TAD JMPI
7601 3163           DCA CALLX+1
7602 1373           TAD KCALLY
7603 3164           DCA CALLX+2
7604 1374           TAD KRETY
7605 3165           DCA RETX
7606 1375           TAD BASE
7607 3166           DCA STACK
                             /CONTINUE WITH REST OF SYSTEM POWER-ON
                             /INITIALISE
                    *7772
7772 5564 JMPI,     JMP I CALLX+2
7773 7400 KCALLY,   CALLY
7774 7411 KRETY,    RETY
7775 0170 BASE,     STACK+2

                    *7776
7776 7600           7600             /START OF INIT ROUTINES
7777 5776           JMP I 7776       /RESET STARTING


                    /EXAMPLE OF USER PROGRAM CALLING A SUBROUTINE
                    /IN LOCATION 6600 FROM LOCATION 5013

                    CALL= JMS CALLX
                    *5013

5013 4162           CALL
5014 6600           6600             /SUBROUTINE STARTS AT 6600

                    /EXAMPLE OF A SUBROUTINE EXIT AT LOCATION 6676

                    RETURN= JMP I RETX
                    *6676

6676 5565           RETURN
```

CONCLUSION

The two different approaches for ROM based subroutine calls are summarized in Table 1.

TABLE 1

	Fixed Overhead		Overhead for Each Active Call	Overhead for Each Subroutine in the Program		Execution Time at 4 MHz	
	RAM	ROM	RAM	RAM	ROM	CALL	RETURN
ALL RAM SYSTEM	0	0	0	1	0	5.5/8.0*	7.5
LINKAGE THRU RAM	0	0	0	3	6	13.0	7.5
SOFTWARE STACK	6	29	1	0	0	70.0	54.0

*8.0μs if the subroutine is not in the Current Page

If the program has more than four subroutines, the memory overhead requirements for the RAM linkage technique exceeds the fixed overhead for the software stack. However, directly linking through RAM is six times faster than what could be achieved with the software stack, and it is only slightly slower than the optimum. The software stack is completely general purpose and the memory overhead is small. The performance penalty is not significant if subtask execution times exceed 1 ms which is the typical IM6100 execution time for a software multiply or divide at 4 MHz. The user must, of course, choose the appropriate method, depending on the speed and memory requirements for a specific task.