A Microprocessor

Part 2: Instruction Set Dead Ends,

Terry Ritter and Joel Boney
Motorola Inc
3501 Ed Bluestein Blvd
Austin TX 78721

Copyright 1978 by
Terry Ritter and Joel Boney

In part 1 of this series (see January 1979 BYTE, page 14) we discussed the instruction set and other details of the Motorola 6809 processor. Part 2 is a question and answer discussion of the design philosophy that went into the 6809.

Any change from old to new inevitably brings criticism from someone. Indeed, any failure to change to include someone's pet ideas brings its own criticisms. We have not been isolated from sometimes severe criticism, nor from its political implications.

However, a number of our decisions have been reasonably challenged, and here we hope to present illumination and defense. While we are aware of a number of improvements which might have been included, the whole point is to sell a reasonably sized (and thus reasonably priced) integrated circuit. We hope that architectural errors of commission, as they are found, will be seen in light of the complete design. We are not aware of any such errors at this time.

Point 1:

The replaced instructions (PSHA/PULA, TAB/TBA, INX/DEX) all take more cycles and bytes than before. Why did you do such a thing?

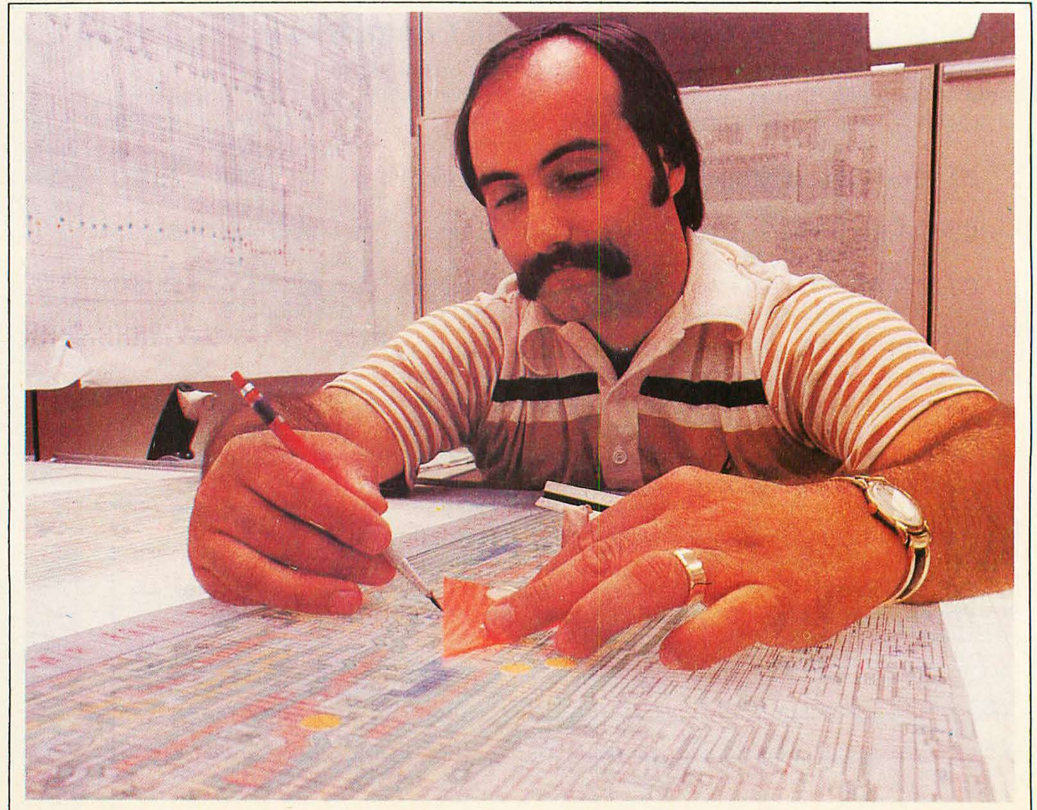

Photo 1: Layout. Layout designer Tony Riccio adds a line in a large layout cell. The various colored lines represent different types of conductors (metal, polysilicon, N+, etc) which will be formed on the integrated circuit. (The yellow dots represent problems to be corrected.)

for the Revolution: The 6809

Old Trails and Apologies

Answer 1:

Consider: the question is not just PSHA/PULA, but rather PSHA/PULA/PSHB/PULB/PSHX/PULX/PSHY/PULY/PSHU/PULU, *etc*, as well as similar op codes for the other stack. *There are only 256 1 byte op codes.* If the PUSHs and PULLs are made 1 byte, others must be made 2 byte, and *these* will take more cycles and bytes than before. And the macrosequenced PUSH or PULL instructions are *more efficient* than 1 byte op codes when more than one register is involved.

Similarly, as more registers are added, the number of possible transfer paths become combinatorially larger. Do you really want to give up that number of 1 byte op codes?

As for INX/DEX, we find that these were frequently used in 6800 code because they were more convenient than any other alternative. We now offer autoincrementing and autodecrementing indexing as a viable (ie: shorter, in cycles and bytes) alternative. We also allow arbitrary additions to X, Y, U and S.

Point 2:

I don't see any facility for expanding the 64 K address space.

Answer 2:

True. Memory expansion is possible, but consider this: microprocessors are products of a mass production technology –

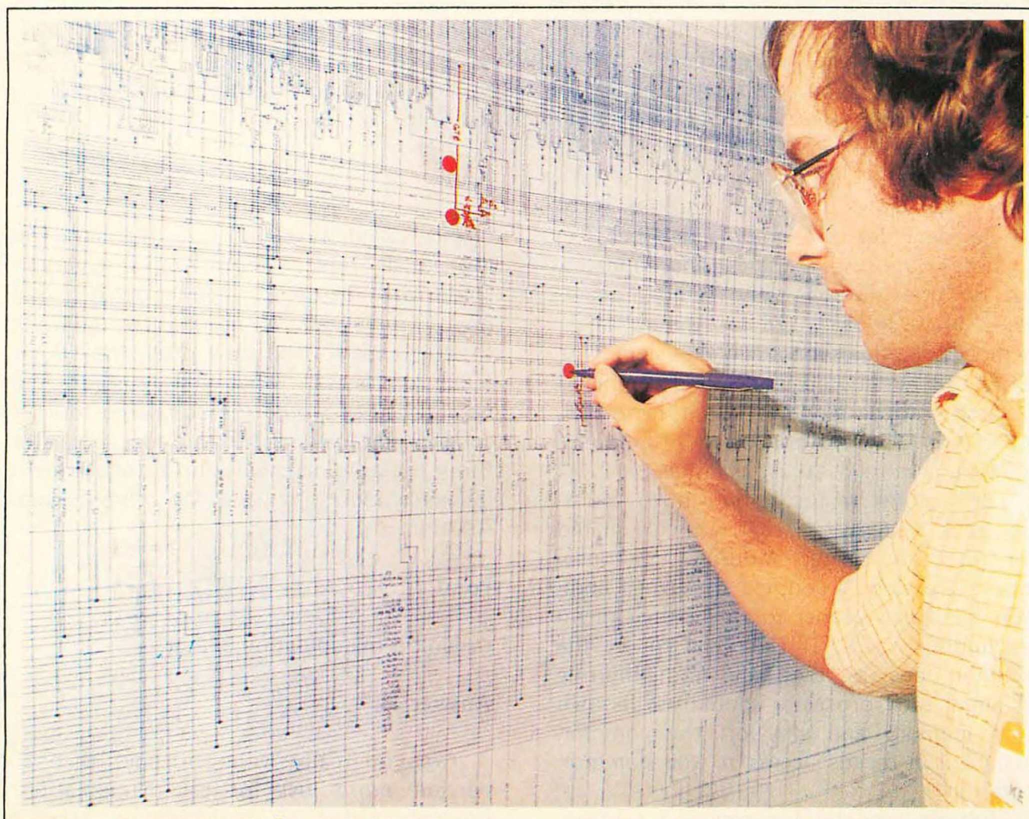

Photo 2: Breadboard design. After partitioning the logic, the MOS (metal oxide semiconductor) diagram is translated to TTL. The required ten boards are then designed and built. Meanwhile, Bill Keshlear updates the logic changes on the master copy of the logic diagrams, since they will imply changes on the boards.

processor cost is no longer a system limiting factor. It is generally inappropriate to use a single $20 processor to control $10,000 worth of memory; the single processor could use only a fraction of the bandwidth resource available in that much memory (here, bandwidth means the maximum possible rate of change of storage state under processor control). A far more reasonable approach is to place the same total store on ten processors and give yourself the possibility of major throughput improvement. Naturally you'll have to learn how to control all this power, but if you're an innovative systems designer, that's exactly your job.

There are two principal divisions of multiprocessor systems, depending on the degree of coupling between the processors. Closely coupled processors usually communicate through some common memory; loosely coupled processors communicate through input/output ports, serial lines, or other "slow" communications channels. Loosely coupled systems can usually be understood as networks of quasi-independent processors.

Now, let's consider a concept that we call "smart memory." One reason for wanting more address space on a processor is to randomly access a large store of on line data. Most of your processing is spent cataloging data, storing data, moving, searching and updating data. If you want to handle more data, you put on more memory and the system gets bigger and slower.

But suppose you put a processor on each reasonable piece of memory (16 K or whatever). Make the program for that processor really dumb — make it just take orders for data. Its whole purpose is to handle data for the command processor; it stores, moves, searches and updates. But for now, it does only memory operations. Now hook a lot of these "smart memory" modules onto your system (the IEEE 488 bus should work), and command a search. All the modules search in parallel, and if you grow and put on more modules, you handle more data just as fast as ever!

The second major approach to multiprocessor systems is what we call shared bus multiprocessing. Multiple microprocessors are closely coupled through a common bus and a proper subset of their memory address space. It is crucial to see the common bus as the bandwidth limiting resource; each processor should use its own local memory and stay off the common bus until it needs access to the common store.

Multiple requests for common memory access might be issued by various processors at exactly the same moment. It is therefore necessary to arbitrate among them, switching exactly one processor onto the common bus, and allowing it to proceed with its memory access while the others are held *not*-READY.

It should be clear that the same concept (a common bus arbitration and switching node) can be hierarchically extended. Further, the addressing capability can be expanded and possibly remapped at each node to allow fast random access to huge amounts of on line mass storage. Such obvious extension is left as an exercise for the serious student. Perhaps you are thinking that you *can* build it, but nobody can write the software to control it. We are not insensitive to the problem, just unhappy with the attitude. We worked hard to give you the tool; all you have to do is learn to use it. Every new technology is like this — our scientists still don't know how to fully control the atom, but that doesn't stop atomic fusion from being one of the most attractive "games" around since the payoffs are huge.

Nobody has a *chance* to develop complex multiprocessor software until she or he has a real multiprocessor system. Now, for $500 and a little work, you've got the hardware. It's time to start learning to control these systems. If it's hard one way, do it another. The power is there for use.

Point 3:

You still didn't include block operations, did you?

Answer 3:

No — and we could have. But have you looked at how often block instructions could really be used in your programs? And how much code is needed to duplicate them yourself? And how often they don't really do exactly what you wanted? And how fast they would run compared to your programmed version? Please do look. We think the autoincrement and autodecrement indexed addressing is a far more general solution.

Point 4:

No bit manipulation, either.

Answer 4:

Are you really willing to pay 10 to 20 percent more just for bit manipulation? Program coded bit manipulation takes a

Photo 3: Visual inspection. Some of the gross processing errors or problems that occur with probing equipment can be detected visually. Here, lead production operator Mary Celedon checks a 6802 wafer.

little longer, but is more general, and probably is located in a very lightly used portion of your program, thus having very little effect on your total throughput or program size.

Point 5:

Why no undefined op code trap?

Answer 5:

Because the machine is a random logic implementation. The unused op codes are used as "don't cares" in derivation of internal logic equations, thus allowing reduced logic and integrated circuit size. Failure to include the don't cares in the logic equations would result in a larger and more expensive circuit.

Point 6:

Some other processors allow both indexed before indirect (indexed indirect) operation and indirect before indexed (indirect indexed) operation, but yours does not. Why?

Answer 6:

First of all, we wanted our addressing modes to operate on all of our memory instructions. Secondly, indirect indexed addressing has much lower utility than our indexed indirect form. Thirdly, we didn't strip down our instruction set, so real features were getting a little precious. Everything has to fit on one chip, remember.

We had considered the possibility of including a sort of chained addressing, in which the memory data would be interpreted as a new indexed postbyte capable of specifying a complete new addressing operation. This sort of thing could continue to indefinite levels, of course. But such an instruction would then be executing data, which is usually a bad idea (self-modifying code) and is also the reason why we include no EXEcute instruction. (Naturally, EXEcute can be emulated if you really need it, but since EXEcute is usually used to make up for the lack of powerful addressing modes, it will not likely be missed from the 6809). Furthermore, this executed data would almost certainly be discontiguous in the memory space, making even the analysis of the simple case (read only memory) programs extremely difficult. Placing such an uncontrollable gimmick in a processor design would be like placing a glittering knife in front of a baby, and would be similarly irresponsible.

Point 7:

You have a MULtiply, but no DIVide.

Answer 7:

True enough. Multiply operations are required in high level language subscript array calculations, but how often do you really need divide? Do you really want to pay for something you will rarely use and can do easily with a program? Additionally, the unsigned multiply is easily capable of extension into multiple precision arithmetic. (Try that with a signed multiply!) Divide does not decompose as nicely. This, combined with the absence of similar instructions in the machine (divide needs

24 bits of parameters, both in and out) was enough to leave it out.

Point 8:

Your registers are all special purpose.

Answer 8:

Well, in a way, as we have 16 bits of accumulator and 64 bits of usable pointers plus some others. This basic dichotomy of data and pointers to data exists in practice, and is therefore rarely a problem with our implementation. But the EXG instruction allows convenient manipulation between these groups in any unusual circumstances.

Point 9:

Why did you include all those new addressing modes? I'll never use them.

Answer 9:

We expect that you *will* use the new addressing modes, and quite heavily. There are a lot of different indexed options. But notice that the large number of different modes is a result of including all permutations of a few basic ideas.

Fundamentally, you can index from any pointer register (x 4), use indexed indirect access (x 2), and have accumulator offsets (x 3) or constant offsets of up to 16 bits in three versions (x 3) (see box at lower right). But if you work in assembly language, you don't need to figure addresses, so the different constant offset modes may be ignored. And if you select an addressing mode which is not available, the assembler will politely inform you of your indiscretion.

Alternately, you can specify autoincrement or autodecrement operations (x 2), by either one or two (x 2), which may be indirected (x 1.5) (except there is no indexed autoincrement and autodecrement by one indirect – think about it). Finally, constant offsets are allowed from the program counter (x 3) and these may also be indirected (x 2).

There are a lot of modes, no doubt about it. But relatively few new ideas are required to gain full control over those powerful new features.

Point 10:

I would have liked an operating system call instruction which carried a parameter to the operating system.

Answer 10:

So would we. Unfortunately, the location I want to use for parameters may not (and probably will not) be what you want to use. It is desirable to allow both constant and variable parameters to the operating system. What you do get is two more trap-like software interrupt (SWI) instructions; the instructions SWI2 and SWI3 do not mask interrupts as SWI does, thus allowing use even in interrupt driven programs. Parameters may be passed in any register, or on the stack, or as the next byte of in line code. All of this will require some overhead, but the scheme is far more general than a trap that carries a parameter.

Point 11:

Tell me again about the stack pointers: why *two* stack pointers?

Answer 11:

Good point. The original reason for adding the user stack pointer was to facilitate the creation of a data stack in memory that is separate from the program stack. This avoids one of the serious problems of using a second generation processor in a modular programming environment – that of returning parameters to a calling routine. We want to pass parameters in a position independent manner, of course, but the return from subroutine (RTS) instruction uses the top element of the stack as a return address, and this address is placed on the stack *before* the subroutine is entered. On the 6800 there will be a lot of stack rearrangement going on to get around this problem. The user stack pointer was created as a new stack unencumbered with return addresses (or interrupt state information) to allow data to be passed between routines of different levels in a reasonable manner. And since the new stack works exactly like the old, there is relatively small silicon cost involved.

We do suspect, however, that many programmers will elect to accept the overhead involved with passing parameters on the hardware stack (note that the overhead problem is greatly reduced with the 6809). These programmers will be concerned with the access of parameters placed on the stack by higher level routines. Notice that, as more elements are added to the stack, these *same parameters* are referred to by *varying offsets* with respect to the stack pointer itself: this is bad, since it becomes difficult to analyze exactly which value is

*The notation (x n) means there are n ways to perform that particular operation. (x 1.5) means there are two ways to perform that operation but not every addressing mode is allowed. . . .*RGAC

Photo 4: Editing the layout. Drafting manager Wayne Busfield and senior layout designer Rick Secrist make changes indicated by engineering analysis. This iterative process improves performance and production yield, and thus lowers cost.

being accessed by any given subroutine. Thus many programmers will use the U register as a *stack mark* pointer, fixed at some previous location of the stack pointer. All lower level modules will then be able to refer to the same data by identical offsets from the U register.

Point 12:

Why do the 6809 stack pointers point to the last item on the stack rather than the next free location, as on the 6800?

Answer 12:

This architectural change was virtually mandated by following the chain of logic that resulted from extending the 6800 into double byte, autoincrement and stack indexable operations.

First, let us assume the above extensions with a 6800 style stack: the stack pointer thus points one byte below (lower in memory) the last byte deposited. Naturally the other pointers should work similarly (allowing their use as additional stacks, and requiring no new understanding). This means that the autoindex operations have to be preincrement and postdecrement. Now, suppose we have a stack or table of double byte data; the data pointer must be set up one byte *below* the data to prepare for autoincrement (or pull) operations. To access the first value, the expression LDD ,+S must be used, while succeeding operations appear to need LDD ,++S. This result is not great for loops. Alternately, the stack pointer could be made to point *two* bytes above the stack for double byte data only. This would require different offsets from the stack pointer (to access, say, the top of the stack) depending upon the size of the data being accessed. Different offsets would also be required, depending on whether the data was just being used,

or being pulled from the stack. This is workable, but not great conceptually. Another possibility is to form the effective address from the value of the pointer after only the *first* increment. This "kluge" solution would be hard to implement anyway, so we changed the stacks.

This chain of reasoning is an example of the difference between architectural design and just slapping instructions together.

Point 13:

Why not have more registers?

Answer 13:

Good designs are often the results of engineering compromises. To meet product size goals, only so many things can go on an integrated circuit. You can have registers, *or* features, or some combination. The 6809 does have approximately 20 addressing modes.

Registers for the sake of registers amount to little more than separate, very expensive and restricted memory areas. The register resource is always insufficient to hold temporary results in a large program, and must be reallocated in various routines. This allocation process is an error prone programming overhead. A separate register set for interrupt processing is suitable only for one interrupt level and, otherwise, is mostly wasted.

A few registers fully supported by features are better than just having a lot of registers.

Point 14:

Why no instructions to load or store the direct page register?

Answer 14:

The direct page register is one of those possibly dangerous features which was just too good to pass up (in terms of substantial benefits for minimum cost). The benefits include an operation length reduction of 33 percent for instructions using absolute addressing and a concurrent throughput increase of 20 percent. It now becomes possible to optimize code, perhaps allowing an oversized program to fit within discrete read only memory boundaries. The direct page register may also be used in a multitasking environment to allow single copies of routines to operate with multiple independent processes. However, providing a separate stack area and having each routine store local values on the stack may be a better solution,

Because a number of 6809 instructions (eg: INC/DEC, ASL/ASR/ROL/ROR/LSL, TST/COM/CLR/NEG) operate directly on memory, the direct page area may be used very much like a processor with 256 8 bit registers to hold counters, flags and serial information. So, perhaps most importantly, the direct page register relaxes the system requirement for programmable memory at a particular location (page 0) to use direct addressing; the cost is a single 8 bit register and no new instructions.

The programmer is cautioned to tread carefully when using the direct page register. All forms of absolute addressing for

Photo 5: First silicon engineering analysis. Logic and circuit design engineer Bob Thompson tracks down a weak node in the first batch of 6801 chips. The 6801 die is packaged, but not sealed, so that internal nodes may be probed while in operation. Viewing through the microscope, a probe can be placed at critical points equivalent to the layout plot. The chip itself is running a modified EXORcisor system, and the scope actually displayed an internal signal with excessively slow rise time.

Table 1: 6809 instruction set.

8 BIT OPERATIONS

Mnemonic	Description
ABX	Add B register to X register unsigned.
ADCA, ADCB	Add memory to accumulator with carry.
ADDA, ADDB	Add memory to accumulator.
ANDA, ANDB	And memory with accumulator.
ANDCC	And immediate with condition code register.
ASLA, ASLB, ASL	Arithmetic shift left accumulator or memory.
ASRA, ASRB, ARS	Arithmetic shift right accumulator or memory.
BITA, BITB	Bit test memory with accumulator.
CLRA, CLRB, CLR	Clear accumulator or memory.
CMPA, CMPB	Compare memory with accumulator.
COMA, COMB, COM	Complement accumulator or memory.
DAA	Decimal Adjust A accumulator.
DECA, DECB, DEC	Decrement accumulator or memory.
EORA, EORB	Exclusive or memory with accumulator.
EXG R1, R2	Exchange R1 with R2.
INCA, INCB, INC	Increment accumulator or memory.
LDA, LDB	Load accumulator from memory.
LSLA, LSLB, LSL	Logical shift left accumulator or memory.
LSRA, LSRB, LSR	Logical shift right accumulator or memory.
MUL	Unsigned multiply (8 bit by 8 bit = 16 bit).
NEGA, NEGB, NEG	Negate accumulator or memory.
ORA, ORB	Or memory with accumulator.
ORCC	Or immediate with condition code register.
PSHS (register)$_0^8$	Push register(s) on hardware stack.
PSHU (register)$_0^8$	Push register(s) on user stack.
PULS (register)$_0^8$	Pull register(s) from hardware stack.
PULU (register)$_0^8$	Pull register(s) from user stack.
ROLA, ROLB, ROL	Rotate accumulator or memory left.
RORA, RORB, ROR	Rotate accumulator or memory right.
SBCA, SBCB	Subtract memory from accumulator with borrow.
STA, STB	Store accumulator to memory.
SUBA, SUBB	Subtract memory from accumulator.
TSTA, TSTB, TST	Test accumulator or memory.
TFR R1, R2	Transfer register R1 to register R2.

16 BIT OPERATIONS

Mnemonic	Description
ADDD	Add to D accumulator.
SUBD	Subtract from D accumulator.
LDD	Load D accumulator.
STD	Store D accumulator.
CMPD	Compare D accumulator.
LDX, LDY, LDS, LDU	Load pointer register.
STX, STY, STS, STU	Store printer register.
CMPX, CMPY, CMPU, CMPS	Compare pointer register.
LEAX, LEAY, LEAS, LEAU	Load effective address into pointer register.
SEX	Sign extend
TRR register, register	Transfer register to register.
EXG register, register	Exchange register to register.
PSHS (register)$_0^8$	Push register(s) onto hardware stack.
PSHU (register)$_0^8$	Push register(s) onto user stack.
PULS (register)$_0^8$	Pull register(s) from hardware stack.
PULU (register)$_0^8$	Pull register(s) from user stack.

temporary values and parameters present problems in the development of large programs. Attempts to enlarge the number of direct locations by manipulating the direct page register may be tricky. And manipulation of the register by subroutines may lead to errors which switch the calling routines direct page in remote (ie: subroutined) unobvious code. Therefore, this register is made deliberately difficult to play with. Typically, it should be set up once and left there. To load the direct page register you can proceed as follows: EXG A,DP; LDA #NEWDP; EXG A,DP. Alternately, the direct page register is also available in PUSH/PULL instructions, but misuse is discouraged through lack of LDDP and STDP.

Point 15:

You preach consistency, yet you give us LEA, an instruction with different condition codes for different registers. Why is this so?

Answer 15:

The Z flag is unaffected by LEAS or LEAU, but conditionally set by LEAX or LEAY depending on the value loaded into the register. This provides 6800 compatibility with INX/DEX (implemented as LEAX 1,S or LEAX –1,X) and INS/DES (implemented as LEAS 1,S and LEAS –1,S), respectively.

Now clearly, if most 6800 programs are going to run on the 6809, the use of INX/DEX for event counts must be recognized. But in 6809 programs, releasing local stack area before executing RTS will be a very frequent action (LEAS –9,S; RTS) "cleaning up the stack." You do want to return a previous condition code value undamaged by the LEAS, so you get two types of LEA.

Point 16:

What about position independent code? Doesn't the 6800 allow it, too?

Answer 16:

Position independent code is one crucial factor in achieving low cost software. (Position independent temporary storage and input/output must also be available.) Only read only memories which may be used in arbitrary target systems are economically viable in the context of mass production. And only these read only memories can result in low cost firmware for us all.

Table 1, continued:

INDEXED ADDRESSING MODES

Mmemonic	Description
0, R	Indexed with zero offset.
[0, R]	Indexed with zero offset indirect.
, R+	Autoincrement by 1.
, R++	Autoincrement by 2.
[, R+++	Autoincrement by 2 indirect.
, -R	Autodecrement by 1.
, --R	Autodecrement by 2.
[, --R]	Autodecrement by 2 indirect.
n, P	Indexed with signed n as offset (n=5, 8, or 16 bits).
[n, P]	Indexed with signed n as offset indirect.
A, R	Indexed with accumulator A as offset.
[A, R]	Indexed with accumulator A as offset indirect.
B, R	Indexed with accumulator B as offset.
[B, R]	Indexed with accumulator B as offset indirect.
D, R	Indexed with accumulator D as offset.
[D, R]	Indexed with accumulator D as offset indirect.

Note: *R = X, Y, U, or S; P = PC, X, Y, U, or S. Brackets indicate indirection. D means use AB accumulator pair.*

6809 RELATIVE SHORT AND LONG BRANCHES

Mnemonic	Description
BCC, LBCC	Branch if carry clear.
BCS, LBCS	Branch if carry set.
BEQ, LBEQ	Branch if equal.
BGE, LBGE	Branch if greater than or equal (signed).
BGT, LBGT	Branch if greater (signed).
BHI, LBHI	Branch if higher (unsigned).
BHS, LBHS	Branch if higher or same (unsigned).
BLE, LBLE	Branch if less than or equal (signed).
BLO, LBLO	Branch if lower (unsigned).
BLS, LBLS	Branch if lower or same (unsigned).
BLT, LBLT	Branch if less than (signed).
BMI, LBMI	Branch if minus.
BNE, LBNE	Branch is not equal.
BPL, LBPL	Branch if plus.
BRA, LBRA	Branch always.
BRN, LBRN	Branch never.
BSR, LBSR	Branch to subroutine.
BVC, LBVC	Branch if overflow clear.
BVS, LBVS	Branch if overflow set.

6809 MISCELLANEOUS INSTRUCTIONS

Mnemonic	Description
CWAI	Clear condition code register bits and wait for interrupt.
NOP	No operation.
JMP	Jump.
JSR	Jump to subroutine.
RTI	Return from interrupt.
RTS	Return from subroutine.
SEX	Sign extend B register into A register.
SWI, SWI2, SWI3	Software interrupts.
SYNC	Syncrhonize with interrupt line.

The 6800 is capable of position independent code execution in relatively small programs. Somewhere around a 4 K byte limit, the program can no longer support all control-transfer paths using branch instructions (even allowing the use of intermediate branch "islands"). Either a long branch subroutine must be used (at a cost of 100+ cycles for each LBSR) or the program must be made position dependent.

Point 17:

What about dynamic memory?

Answer 17:

There are two problems associated with dynamic memories: address bus multiplexing and refresh. Address bus multiplexing is the most severe problem but requires multiplexing 6+6 address lines (for 4 K memories) or 7+7 lines (for 16 K memories); these values are particularly inconvenient for 8 bit processors (which usually multiplex address/data). Thus, we have yet to see a processor address this problem.

Microprocessors that automatically refresh memory during most unused bus cycles waste power on unnecessary refreshes and unnecessarily increase bus activity. The 6809 can easily refresh dynamic memory in software (a timer causes interrupt execution of of FCB $10 63 times, then RTI), or can support hardware refresh (a direct memory access [DMA] sequence, or isolated board automatic refresh) at minimal cost.

Point 18:

What about price?

Answer 18:

The 6809 will be more expensive than in-production second generation 8 bit designs. For one thing, it is bigger and also new — both reasons imply reduced yield compared to older parts. A moderately higher price should not be a problem, since the processor cost is a very minor part of the price of a whole system. The total 6809 system should be nearly as powerful and much less expensive than 16 bit designs. The cost of not using the 6809, on the other hand, will likely be severe in terms of increased programming error rates, larger read only memories and decreased throughput.

In "Part 3: Final Thoughts" (March 1979 BYTE), we will conclude this series with a discussion of clock speed, timing, condition codes and software design philosophy.■