The 6809
Part 1: Design Philosophy

Terry Ritter
Joel Boney
Motorola, Inc.
3501 Ed Blustein Blvd.
Austin, TX 78721

This is a story. It is a story of computers in general, specifically microcomputers, and of one particular microprocessor - with revolutionary social change lurking in the background. The story could well be imaginary, but it happens to be true. In this 3 part series we will describer the design of what we feel is the best 8 bit machine so far made by human: the Motorola M6809.

Philosophy

A new day is breaking; after a long slow twilight of design the sun is beginning to rise on the microprocessor revolution. For the first time we have mass production computers; expensive custom, cottage industry designs take on less importance.

Microprocessors are real computers. The first and second generation devices are not very sophisticated as processors go, but the are general-purpose logic machines. Any microprocessor can eventually be made to solve the same problems as any large scale computer, although this may be an easier or harder task depending on the microprocessor. (Naturally, some jobs require doing processing fast, in real time. We are not discussing those right now. We are discussing getting a big job done sometime.) What differentiates the classes is a hierarchy of technology, size performance, and curiously, philosophy of use.

A processor of given capability has a fixed general complexity in terms of digital logic elements. Consider the computers that were built using the first solid state technology. In short they consisted of many thousands of individual transistors and other parts on hundreds of different printed circuit boards using thousands of connections and miles of connecting wire. A big computer was a big project and a very big expense. This simple economic fact fossilized a whole generation of technology into the “big computer philosophy.”

Because the big computer was so expensive, time on the computer was regarded as a limited and therefore valuable resource. Certainly the time was valuable to researchers who could now look more deeply into their equations than ever before. Computer time was valuable to business people who became at least marginally capable of analyzing the performance of an unwieldy bureaucratic organization. And the computer makers clearly thought that processor time was valuable too; or was a severely limited resource, worth as much as the market would bear.

Processor time was a limited resource. But some of us, a few small groups of technologists, are about to change that situation. And we hope we will also change how people look at computers, and how professionals see them too. Computer time should be cheap; people time is 70 years and counting down.

The large computer, being a very expensive resource, quickly justified the capital required to investigate optimum use of that resource. Among the principal results of these projects was the development of batch mode multiprocessing. The computer itself would save up the various tasks it had to do, then change from one to the other at computer speeds. This minimized the wasted time between jobs and spawned the concept of an operating system.

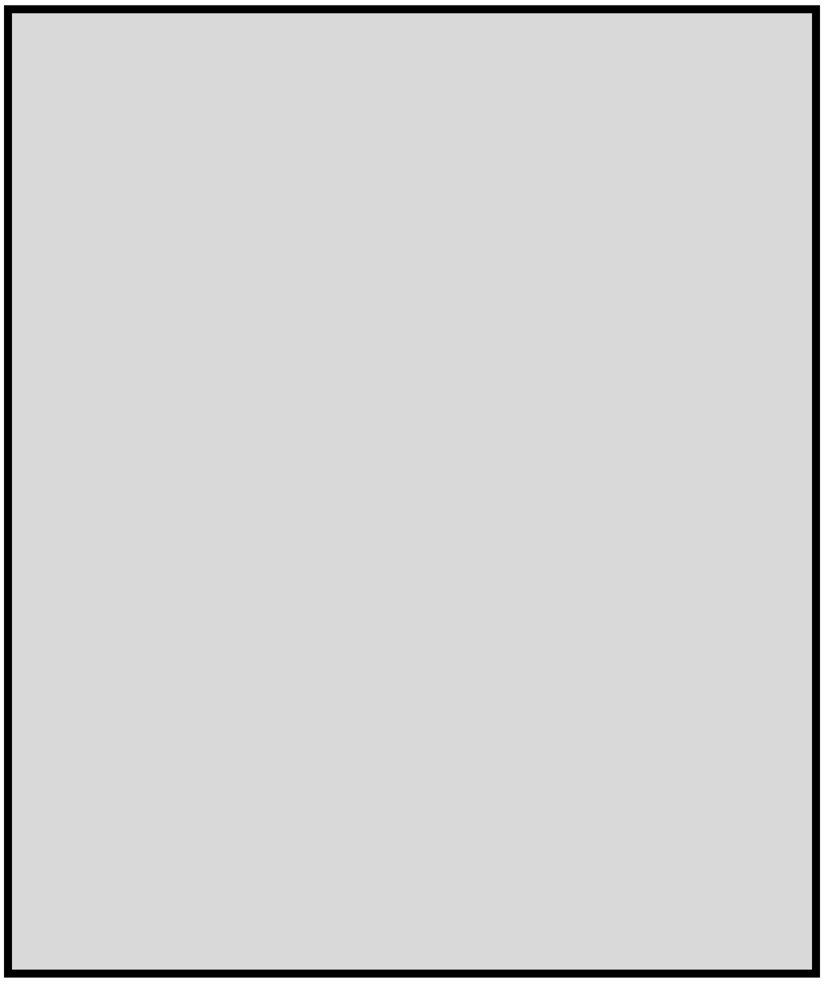


Photo 1: Systems architects Ritter (right) and Boney review some of the 6809 design documents. This work results in a complete description of the desired part in a 200 page design specification. The specification is then used by logic designers to develop flowcharts of internal operations on a cycle by cycle basis.

Photo 2: 6809 logic design. Design engineer Wayne Harrington inspects a portion of the 6809's processor logic blueprint at the Motorola Austin plant. The print is colored by systems engineers to partition the logic for the logic-equivalent TTL "breadboard."

People were in the position of waiting for the computer, not because they were less important than the machine, but precisely because it was a limited resource (the problems it solved were not).

Electronics know-how continued to develop, producing second generation solid state technology: families of digital logic integrated circuits replaces discrete transistors designs. This new technology was exploited in two main thrusts: big computers could be made conceptually bigger (or faster, or better) for the same expense, or computers could be made physically smaller and less expensive. These new, smaller computers (minicomputers) filled market segments which could afford a sizable but not huge investment in both equipment and expertise. But most people, including scientists and engineers, still used only the very large central machines. Rarely were minicomputers placed in schools; few computer science or electrical engineering departments (who might have been at the leading edge of new generation technology) used them for general instruction.

And so the semiconductor technologists began a third generation technology: the ability to build a complete computer on a single chip of silicon. The question then became, "How do we use this new technology (to make money)?"

The semiconductor producer's problem with third generation technology wa that an unbelievably large development expense was (and is) required to produce just one large scale integration (LSI) chip. The best road to profit was unclear; for a while, customer interconnection of gate array integrated circuits was tried, then dropped. Complete custom designs were (and are) found to be profitable only in vary large volumes.

Another road to profit was to produce a few programmable large scale integration devices which could satisfy the market needs (in terms of large quantities of different systems) and the factory;s needs (in terms of volume production of exactly the dame device). Naturally, the general-purpose computer was seen as a possible answer.

About the Authors

Joel Boney and Terry Ritter are with the Motorola 6800 Microprocessor Design Group in Austin TX. Joel is responsible for the software inputs into the design of the 6800 family processors and peripheral parts and was a co-architect of the M6809. Terry Ritter is a microcomponent architect, responsible for the specification of the 6809 advanced microprocessor. While with Motorola, Terry has been co-Architect of the 6809, and co-architect as well of the 6847 and 68047 video display generator integrated circuits. He holds a BSES from the University of Texas as Austin and Joel Boney has a BSE from the University of South Florida.

So what was the market for a general-purpose computer? The first thought was to enter the old second generation markets; ie: replacement of the complex logic of small or medium scale integration. Control systems, instruments and special designs could all use a simular processor, but this designer was the key. Designers (or design managers)had to be converted from their heavy first and second generation logic design backgrounds to the new third generation technology. In so doing, some early marketing strategists overlooked the principal microprocessor markets.

Random logic replacement was by no means a quick and sufficient market for microprocessors. In particular, the design cycle was quite long, users we often unsophisticated in their use of computers, and the unit volumes was somewhat small. Only when microprocessors entered high volume markets (hobby, games, etc) did the manufactures begin to make money and thus provide a credible reason (and funds) for designing future microprocessors. Naturally, the users who wanted more features were surprised that it was taking so long to get new designs - they knew what was needed.

Thus semiconductor makers began to realize that their market was more oriented to hobby applications that to logic replacement, and was more generalized than they had thought. But even the hobby market was saturable.

Meanwhile companies continued to improve production and reduce costs, and competition drove process down into the ground. Where could they sell enough computers for real volume production, the wondered. One answer was the personal computer!

Design of Large Scale Integration Parts

The design of a complex large scale integration (LSI) part may be conveniently broken into thee phases: the architectural design, the logic and the layout software and hardware (breadboard) simulations. Each phase ha its own requirements.

The architect/systems designers represent the use of the device, the need of the marketplace and the future needs of all customers. They propose what a specific customer should have that could also be used by other customers, possible in different ways. They advocate what the customers will really want, even when if no customers can be identified who know that they will want it. that it is possible or that they will want it. The attitude that “I know what is best for you” and be irritating to most people, but it is necessary in order to make maximum use of a limited resource (in this case, a single LSI design). The architect eventually generates the design specification used in subsequent phases of the design.

Logic design consists of the production of a cycle by cycle flowchart and the derivation of the equations and logic circuitry necessary to implement the specified design. This is a job of immense complexity and detail, but it is absolutely crucial to the entire project. Throughout this phase, the specification may be iterated toward a local optimum of maximum features at minimum logic (and thus cost). The architectural design continues, and techniques are developed to cross-check on the logical correctness of the architecture.

The third phase is the most hectic in terms of demands and involvement. By this time, many people know what the product is and see the resulting part merely as the turning of an implementation “crank.” It seems to those who are not involved in this phase that more effort could case that crank to turn faster. Since the product could be sold immediately, delay is seen as a real loss of income. In actual practice, more effort will sometimes “break the crank.”

The other major device needed for home computers–the video display generator color TV interface–is presently in volume production. Several versions are available, many derived from the original Motorola architecture

A medium scale integration logic implementation (usually transistor-transistor logic, for speed) is required to verify the logic design. A processor emulation may require ten different boards of 80 medium scale integrated circuits each and hundreds of board to board interconnections. Each board will likely require separate testing, and only then will the emulation represent the processor to come. Extensive test programs are required to check out each facet of the part, each instruction, and each addressing mode. This testing may

Photo 3: 6809 emulator board. Software and systems engineers implement a functional equivalent of the 6809 as a 6800 program. A 6800 to 6809 cross assembler allows 6809 programs to be assembled and then executed as a check of the architectural design.

detect logic design errors that will have to be fixed at all levels of design.

Circuit design, in the context of the semiconductor industry, depends upon running computer simulation (which require sophisticated device models) of signals at various nodes to verify that they will meet the necessary speed requirement. Transistors are sized and polysilicon lines changed to provide reliable worst case operation.

Layout is the actual task of arranging transistors and interconnections to implement the logic diagram. Circuit design results will indicate appropriate transistor sizes and polysilicon widths; these must now be arranged for minimum area. Every attempt is made to make general logic "cells" which can be used in many places across the integrated circuit, but minimization is the principal concern.

The layout for the chip eventually exists only as a computer data base. Each cell is individually digitized into the computer, where is can be arbitrarily positioned, modified or replicated as desired. Large 2 by 3 m (6.5 by 10 feet) plots of various areas of the chip are hand checked to the logic diagram by layout and circuit designers as final checks of the implemented circuit.

Photo 4: Circuit design. Detailed computer simulations of the circuit under design yield predictions of on chip waveforms. Tulley Peters and Bryant Wilder decide to enhance a particular critical transistor.

When layout is complete, the computer data base that represents the chip design is sent to the mask shop (the mask is a photographic stencil of the part used in the manufacturing process). At the mask shop precision plotting and photographic step and repeat techniques are used to produce glass plates for each mask layer. Each mask covers an entire wafer with etched nickel or chrome layouts at real chip size. (A typical LSI device will be between 5 by 5 and 7.6 by 7.4 mm (0.2 by 0.2 and 0.3 by 0.3 inches). These masks are used to expose photosensitive etch resist the will protect some areas of the wafer from the chemical processes which selectively add the impurities that create transistors.

Actual processing steps are quite simular for each part. But the processing itself is a variable, and it will not be known until final testing exactly how many parts will turn out to be saleable. Therefore, a best estimate is taken, and the required numbers of wafers (of a particular device) is started and processed. The whole industry revolves around highly trained production engineers, chemists and others who process wafers to highly secret recipes. Some recipes work, some don't. You find out which ones do by testing.

Each die (ie: individual large scale integration circuit) is tested while still on the wafer; failing devices are marked with a blob of ink. The wafer is sawed into individual dies and the good devices placed into a plastic or ceramic package base. The connection pads are "die bonded" to the exposed internal lead frame with very tiny wire. The package is then sealed and tested again.

Testing a device having only 40 pins but which has up to 40,000 internal transistors is no mean trick nor a minor expense. Furthermore, the device must execute all operations properly at the worst case system conditions (which may be high or low extremes of temperature, voltage and loading) and work with other devices on a common bus. Thus, the device is not specified to its own maximum operating speed, but rather the speed of a worst case system. Motorola microprocessors can usually be made to run much faster (and much slower) than their guaranteed worst case specifications.

Project Goals

The 6809 project started life with a number of (mostly unformalized) goals. The principle public goal was to upgrade the 6800 processor to be definitely superior to the 8 bit competition. (The Motorola 68000 project will address the 16 bit market with what we believe will be another superior processor.) Many people, including many customers, felt that all that had to be done was to add another index register (Y), a few supporting instructions (LDY, STY) and correct some of the past omissions (PSHX, PULX, PSHU,m PULY). Since this would mean a rather complete redesign anyway, it made little sense to stop there.

A more philosophical goal — thus one much less useful in discussions with engineers and managers (who had their own opinions of what the project should be) — was to minimize software cost. This led to an extensive, and thus hard to explain sequence of logic that went somewhat like this:

Q: How do we reduce software costs?

A: 1. Write code is a block structured high level language.

2. Distribute the code in mass production read only memories.

Q: Why aren't many read only memories being used now?

A: 1. The great opportunities for error in assemble language allow many mistakes which incur sever read only memory costs.

2. The present architecture is not suitable for read only memories.

Q: In what way are the second generation processors unsuitable?

A: It is very difficult to use a read only memory in any other context than that for which it was originally developed. It is hard to use the same read only memory on systems built by different vendors. Simply having different input and output (IO) or using a different memory location is usually enough to make the read only product useless.

Q: What is needed?

A: 1. Position independent code.

2. Temporary variables on the stack.

3. Indirect operations through the stack for input and output.

4. Absolute indirect operation for system branch tables.

And so it went. How could we make a device that would answer the software problems of two generations of processors? How indeed!

Design Decisions

Usually an engineering project may be pursued in many ways, but only one way at a time. The ever present hope is that this one time will be the only time necessary. Furthermore, it would be nice to get the project over with as soon as possible to get on with selling some products. (A rapid return on investment is especially important in a time of rapid inflation.) To these honorable ends certain decisions are made which delineate the

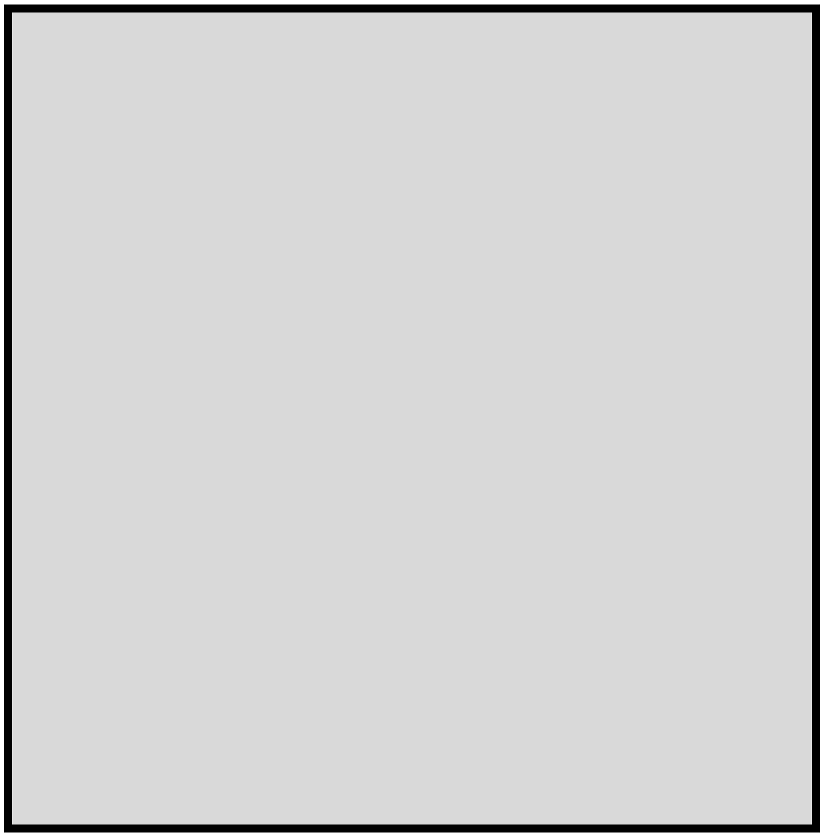


Photo 5: Checking the flowcharts. Logic and circuit designer Bryant Wilder compares the specification to one of the flowcharts. The flowcharts are used to develop Boolean equations for the required logic; those equations are then used to generate a logic diagram.

Instruction Class	Percent Usage
Loads	23.4
Stores	15.3
Subroutine calls and returns	13.0
Conditional branches	11.0
Unconditional branches and jumps	6.5
Compares and tests	6.2
Increments and decrements	6.1
Clear	4.4
Adds and subtracts	2.8
All others	11.3

Table 1: 6800 instruction types based on static analysis of 25,000 lines of 6800 source code. In static analysis the actual number of occurrences of each instruction is tallied from program listings. In the alternate technique, called dynamic analysis, the numbers of occurrences of an instruction is tallied while the program is running. An instruction inside a program loop would therefore be counted more than once.

investment and risk undertaken in an attempt to achieve a new product.

The 6809 project was no exception. To minimize project risk it was decided that the 6809 would be built on the same technological base as the recently completed 6800 depletion load redesign. In particular, the machine would be a random logic computer with essentially dynamic internal operation. It would use the reliable 6800 type of storage register. Functions would be compatible with the defined 6800 bus and 6800 peripherals. This decision would extend the like of parts already in production and minimize testing peripheral devices for a particular processor (6800 versus 6809). Buss compatibility doesn't have to mean identity — the new device could have considerably improved specifications but could not do worse than the specifications for the existing device. This mandate was a little tricky when you consider that we were dealing with a more complex device using exactly the same technology, but there was a slight edge: the advancing very large scale integration (VLSI) learning curve.

One wide range decision wa that the new device would be an improved 6800 part. The widely known 6800 architecture would be iterated and improved, but no radical departure would be considered. In fact, the new devise should be code compatible with the 6800 at some level.

Compatibility was the basis for the 6809 architecture design. It implied that the 6809 could capitalize on the existing familiarity with the 6800. 6800 programmers could be programming for the 6809 almost immediately and could learn and use new addressing mode and features as they were needed. This decision also ended any consideration of radically new architecture for the machine before it was begun.

A corporation selling into a given market is necessarily limited to moderate innovation. Any vast product change requires reeducation of both the internal marketing organization and the customer base before mass sales can proceed. Consequently, designers have to restrict their creativity to conform to the market desires. The amount of change actually implemented, produced and seen by society is the true meaning of a computer "generation." In the end, society itself defines the limits of a new generation, and a design years ahead of its time may well fail in the marketplace.

M6800 Data Analysis

Once the initial philosophical and marketing trade-offs were made, construction of the final form of the M6809 began. By this time a large numbers of M6800 programs had been written by both Motorola and our customers, so it was felt that a good place to start design of the 6809 was to analyze large amounts of existing 6800 source code. Surprisingly, the data gathered about 6800 usage of instructions and addressing modes agreed substantially with simular data previously compiled for minicomputers and maxicomputers. By far the most common instructions were load and stores, which accounted for over 38 percent of all 6800 instructions. Next were the subroutine calls (Direct, Extended, Immediate, Indexed, Relative, Accumulator) had nearly equal usage, which indicated that programmers actually took advantage of the bytes to be saved by direct (page zero) addressing and indexed addressing. Furthermore the offsets for indexed instructions were either 0 or less than 32 (see table 2).

This information was used to greatly expand the addressing modes (as discussed later) with out making the 6800 programs require more code when converted to run on the 6809. Also the number of increment or decrement index register instructions in loops indicated that autoincrementing and autodecrementing would be beneficial. Auto decrementing and autoincrementing are simular to indexing except the index register used is decremented before, or decremented

Table 2: Size of offsets used in 6800 indexed addressing, based on static analysis of 25,000 lines of 6800 source code.

Index Offset	Percent Usage
0	40.0
1-31	53.0
32-63	1.0
64-255	6.0

after, the addressing operation takes place.

As all programmers and even architects like ourselves eventually learn, consistent and uniform instruction sets are used more effectively than instruction sets that treat similar resource (IO, registers or data) in dissimilar ways. For example, the least used instructions on the 6800 were those that dealt with the A accumulator in specific ways that did not apply to the B accumulator (eg: ABA: add B to A, CBA: compare B to A). It's not that these instructions are not useful, it's just that programmers will not use inconsistent instructions or addressing modes. Consistency became the battle cry of the M6809 designers!

Customer Inputs

At the completion of the 6800 analysis stage, the first preliminary design specification for the 6809 was generated. This preliminary specification was then taken to about 30 customers who represented a cross section of current 6800 users, as well as some customers and consultants known to be hostile to the 6800. With these customers visits we hoped to resolve two major questions about the 6809's architecture:

1) Which architecture was more desirable 8 bit or 16 bit?

2) Did 6809 compatibility with the 6800 need to occur at the object level or at the source level.

Most customers felt that an 8 bit architecture was adequate for their upcoming applications, and they did not want to pay the price penalty for 16 bit as long as the 6809 included the most common 16 bit operations such as add, subtract, load, store, compare and multiply. Many were interested though, in Motorola's advanced 16 bit processor (68000) for future 16 bit applications. From the very inception of the6809 project it was a requirement that the 6809 would be compatible with the 6800. Wether this compatibility needed to occur at the object level or at the assembly language (source code) level was a question we felt our customers should help us answer. Virtually every customer indicated that source compatibility was sufficient because they would not try to use 6800 read only memories in 6809 systems. Most customers indicated that they would take advantage of the 6800 compatibility in order to initially convert running 6800 programs into running 6809 programs, and then modify the 6800 code to take advantage of the 6809's features.

The decision not to be object code compatible was an easy one for us since it meant that we

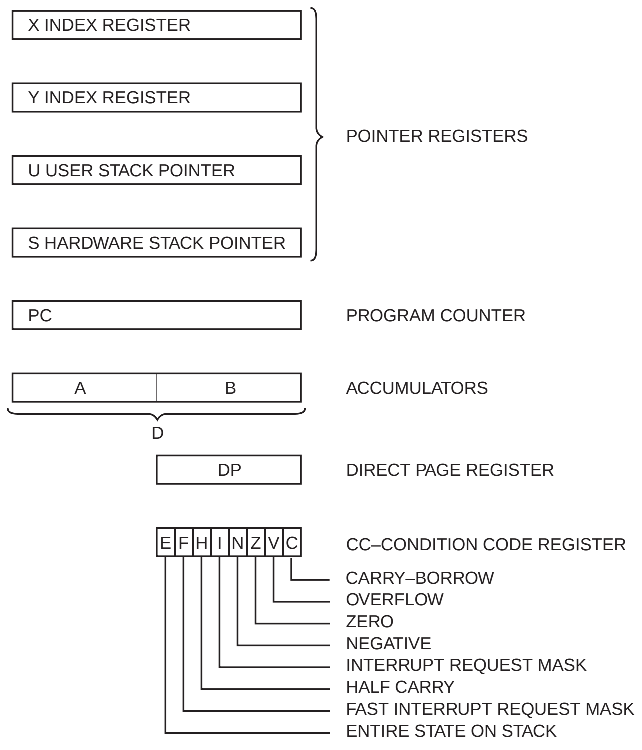


Figure 1: 6809 programming model.

could remap the 6800 op codes in a manner guaranteed to produce more byte efficient and faster 6809 programs. The remapping of op codes was greatly affected by the 6800 data analysis. Some low occurrence 6800 instruction were combined into consistent 2 byte instructions, allowing the more useful instruction to take fewer bytes and execute faster. Also, some 6800 instructions were eliminated completely in favor of 2 instruction sequences. These sequences are generated automatically by our assembler when the 6800 mnemonic is recognized. This remapping in favor of more often used functions results in 6809 programs that require only one half to two thirds as much memory as 6800 programs, and run faster.

M6809 Registers

What, then, are the pertinent features that make the 6809 a next generation processor? In the following paragraphs we will attempt to highlight the improvements made to the 6800. The programming model for the 6809 (figure 1) consists of four 8 bit registers and five 16 bit registers.

The A and B accumulators are the same as those of the 6800 except that they can also be catenated into the A:B pair, called the D register, for 16 bit operations.

The condition codes are simular to the 6800, with the inclusion of two new bits. The F bit is the interrupt mask bit for the new fast interrupt. The

		Nonindirect				Indirect			
Type	Forms	Source	Post Byte	$\overset{+}{\sim}$	$\overset{+}{\#}$	Source	Post Byte	$\overset{+}{\sim}$	$\overset{+}{\#}$
Constant offset from R	no offset	,R	1RR00100	0	0	[,R]	1RR10100	3	0
	5 bit offset	n,R	0RRnnnnn	1	0		defaults to 8-bit		
	8 bit offset	n,R	1RR01000	1	1	[n,R]	1RR11000	4	1
	16 bit offset	n,R	1RR01001	4	2	[n,R]	1RR11001	7	2
Accumulator offset from R	A register offset	A,R	1RR00110	1	0	[A,R]	1RR10110	4	0
	B register offset	B,R	1RR00101	1	0	[B,R]	1RR10101	4	0
	D register offset	D,R	1RR01011	4	0	[D,R]	1RR11011	7	0
Autoincrement/ –decrement R	increment by 1	,R+	1R000000	2	0		not allowed		
	increment by 2	,R++	1RR00001	3	0	[,R++]	1RR10001	6	0
	decrement by 1	,-R	1RR00010	2	0		not allowed		
	decrement by 2	,--R	1RR00011	3	0	[,--R]	1RR10011	6	0
Constant offset from program counter	8 bit offset	n,PCR	1XX01100	1	1	[n,PCR]	1XX11100	4	1
	16 bit offset	n,PCR	1XX01101	5	2	[n,PCR]	1XX11101	8	2
Extended		use nonindexed				[n]	10011111	5	2

Table 3: Indexed addressing modes. All instructions with indexed addressing have a base size and number of cycles. The $\overset{+}{\sim}$ and $\overset{+}{\#}$ columns indicate the number of additional cycles and bytes for the particular variation. The post byte op code is the byte that immediately follows the normal op code.

fast interrupt (FIRQ) only stacks the program counter and condition code register when an interrupt occurs. The interrupt routine is then responsible for stacking any registers it uses. The E bit is set when the registers are stacked during interrupts if the entire register set was saved (as in nonmaskable and maskable interrupts) or cleared if the short register set was saved (for a fast interrupt).

On the 6800, an instruction with direct mode (or page zero) addressing consisted of an op code followed by an 8 bit value that defined the lower eight bits of an address, The upper eight bits were always assumed to be zero. Thus, direct addressing could only address locations in the lowest 256 bytes of memory. The 6809 adds versatility to this addressing mode by defining an 8 bit direct page register that defines the upper eight bits of address for all direct addressing instructions. This allows direct mode addressing to be used throughout the entire address space of the machine. To maintain 6800 compatibility, the direct page register is set to 0 on reset.

Four 16 bit indexable register are included in the 6809. They are the X, Y, U and S registers. The X register is the familiar 6800 index register, and the S register is the hardware stack pointer. The Y register is a second index register; the U register is the user stack pointer. All four registers can be used in all indexing operations and the U and S resisters are also stack pointers, The S register is used during interrupts and subroutine calls by the hardware to stack return addresses and machine states.

The last 16 bit register is the program counter. In certain 6809 addressing modes, the program counter can also be used as an index register to achieve position independent code.

Addressing Modes

It was out opinion that the best way to improve an existing architecture and maintain source compatibility was to add powerful addressing modes. In out view, the 6809 has the most powerful addressing modes available on any microprocessor. Powerful addressing modes helped us achieve out goals of position independence, reentrancy, recursion, consistency and easy implementation of block structured high level languages.

All the 6800 addressing modes (immediate, Extended, Direct, Indexed, Accumulator, Relative, and inherent) are supported on the 6809 with the direct mode of addressing made more useful by the inclusion of the direct page register (DPR).

The direct page register usage and direct addressing need some explanation, since they can be very effective when used correctly. For example, since global variables are referenced frequently in high level language execution, the direct page register can be used to point to a page containing the global variables while the stack contains the local variables, which are also referenced frequently. This creates very efficient code which is

```
                         NAM     AUTOEX
                         OPT     LLEN=80
                         *
                         ************************************************************
                         *   COMPARE STRINGS SUB
                         *
                         *    FIND AN INPUT ASCII STRING POINTED TO BY THE
                         *    X-REGISTER IN A TEXT BUFFER POINTED TO BY THE
                         *    Y-REGISTER. THE BUFFER IS TERMINATED BY A
                         *    BYTE CONTAINING A NEGATIVE VALUE. ON ENTRY
                         *    A CONTAINS THE LENGTH OF THE INPUT STRING. ON
                         *    EXIT Y CONTAINS THE POINTER TO THE START
                         *    OF THE MATCHED STRING + 1 IFF Z IS SET. IFF Z
                         *    IS NOT SER THE INPUT STRING WAS NOT FOUND
                         *
                         *    ENTRY:
                         *        X POINTS TO INPUT STRING
                         *        Y POINTS TO TEXT BUFFER
                         *        A LENGTH OF INPUT STRING
                         *     EXIT:
                         *        IFF Z=1 THEN Y POINTS TO MATCHED STRING + 1
                         *        IFF Z = 0 THE NO MATCH
                         *        X IS DESTROYED
                         *        B IS DESTROYED
                         *
                         ************************************************************
                         *
0100         6           ORG     $100
0100  E6 A0  6   CMPSTR  LDB     ,Y+       GET BUFFER CHARACTER
0102  2A 01  3           BPL     CMP1      BRANCH IS NOT AT BUFFER END
0104  39     5           RTS               NO MATCH, Z=0
0105  E1 84  4   CMP1    CMPB    ,X        COMPARE TO FIRST STRING CHAR.
0107  26 F7  3           BNE     CMPSTR    BRANCH ON NO COMPARE
                 *SAVE STATE SO SEARCH CAN BE RESUMED IF IT FAILS
0109  34 32  9           PSHS    A,X,Y
010B  30 01  5           LEAX    1,X       POINT X TO NEXT CHAR
010D  4A     2   CMP2    DECA              ALL CHARS COMPARE?
010E  27 0C  3           BEQ     CMPOUT    IF SO, IT’S A MATCH, Z=1
0110  E6 A0  6           LDB     ,Y+       GET NEXT BUFFER CHAR
0112  2B 08  3           BMI     CMPOUT    BRANCH IS BUFFER END, Z=0
0114  E1 80  6           CMPB    ,X+       DOES IT MATCH STRING CHAR?
0116  27 F5  3           BEQ     CMP2      BRANCH IF SO
0118  35 32  9           PULS    A,X,Y     SEARCH FAILED, RESTART SEARCH
011A  20 E4  3           BRA     CMPSTR
011C  35 B2  11  CMPOUT  PULS    A,X,Y,PC  FIX STACK, RETURN WITH Z
                 *
      0000               END
```

Listing 1: 6809 autoincrementing example. This subroutine searches a text buffer for the occurrence of an input string. In autoincrement mode, the value pointed to by the index register is used as the effective address and the index register is then incremented.

safe since the compiler keeps track of the direct page register. The direct page register can also be used effectively and safely in a multitasking environment where the real time operating system allocates a different base page register for each task.

On the other hand, it would be quite dangerous to indiscriminately reallocate the direct page register frequently, such as within subroutines or loops, since it might become very easy to lose track of the current direct page register value. Therefore, even though the direct page register is unstructured, we included it because, when used correctly, the byte savings are significant. Also, to make direct addressing more useful, the read modify write instruction on the 6809 now have all memory addressing modes: Direct, Extended and Indexed.

The major improvements in the 6809’s addressing mode were made by greatly expanding the indexed addressing modes as well as making all indexable instructions applicable to the X, Y, U and S registers (see table 3).

Indexed addressing with an offset is familiar to 6800 users, but the 6809 allows the offset to be any of four possible lengths: 0, 5, 8 or 16 bits, and the offsets are signed two’s complements values. This allows greater flexibility in addressing while achieving maximum byte efficiency. The inclusion of the 16 bit offset allows the role of index register and offset to be reversed if desired. A further enhancement allows all of the above modes to include an additional level of indirection. Even extended addressing can be indirected (as a special indexed addressing mode). Since either stack pointer can be specified as a base address in indexed addressing, the indirect mode allows addresses of data to be passed to a subroutine. The subroutine can then reference the data pointed to with one instruction. This increases the efficiency of high level language calls that pass arguments by reference.

M6800 data indicated that quite often the index register was being used in a loop and incremented or decremented each time. This moved the pointer though tables or was used to move data from one area of memory to another (block moves). Therefore, we implemented autoincrement and autodecrement indexed addressing in the M6809. In autoincrement mode the value pointed to by the index register is used as the effective address, and then the index register is incremented. Autodecrement is similar except that the index register is first decremented and then used to obtain the effective address. Listing 1 is an example of a subroutine that searches a text; buffer for the occurrence of an input string. It makes heavy use of autoincrementing.

Since the 6809 supports 8 and 16 bit operations, the size of the increment or decrement can be selected by the programmer to be 1 or 2. The post increment, predecrement nature of the addressing mode makes it equivalent in operation to a push and pull from a stack. This allows the X and Y registers to also be used as software stack pointers if the programmer needs more than two stacks. All indexed addressing modes can also contain an extra level of post indirection. Autoincrement and autodecrement are more versatile than the block moves and string commands available on other processors.

```
00010 0100                ORG  $100
00011 0100  108E 1000  4  LDY  #CAT   LOAD BASE ADDRESS OF ARRAY
00012 0104  96   32    4  LDA  SUB1   GET FIRST SUBSCRIPT
00013 0106  C6   64    2  LDB  #100   MULTIPLY BY FIRST DIMENSION
00014 0108  3D        11  MUL
00015 0109  D3   33    6  ADDD SUB2   ADD SECOND SUBSCRIPT
00016 010B  EC   AB    9  LDD  D,Y    FETCH VALUE
```

Listing 2: Array subscript calculations. This 6809 program fetches a 16 bit value from a two-dimensional array called CAT, with dimensions: CAT (100,30).

Quite often the programmer needs to calculate the offset used by an indexed instruction during program execution, so we included an index mode that allows the A, B, or D accumulator to be used as an offset to any indexable register. For example, consider fetching a 16 bit value from a two dimensional array called CAT with dimensions: CAT (100,30). Listing 2 shows the 6809 code to accomplish this fetch. These addressing modes can also be indirected.

Implementation of position independent code was one the highest priority design goals. The 6800 had limited position independent code capabilities for small programs, but we felt the 6809 must make this type of code so easy to write that most programmers would make all their programs position independent. To do this a additional long relative (16 bit offset) branch mode was added to all 6800 branches as well as adding program relative addressing. Program relative addressing uses the program counter much as indexing uses on of the indexable registers. This allows all instructions that reference memory to reference data relative to the current program counter (which is inherently position independent). Of course, program relative addressing can be indirected.

The addressing modes of the 6809 have created a processor that has been termed a “programmer’s dream machine.” To date all the benchmarks we have written for the 6809 are position independent, modular, reentrant and much smaller than comparable programs on other microprocessors. It is easier to write good programs on the 6809 than bad ones!

New or Innovative Instructions

The 6809 does not contain dozens of new innovative instructions, and we planned it that way. What we wanted to do was clean up the 6800 instruction set and make it more consistent and versatile. We do not feel a processor with 500 different assembler mnemonics for instructions is better than on with 59 powerful instructions that operate on different data in the manner, for example, the 6809 contains a transfer instruction of the form TFR R1, R2 that allows transfer of any like-sized registers. There are 42 such valid combinations on the 6809, and clearly one TFR instruction is easier to remember than 42 mnemonics o the form: TAB, TBA, TAP, TXY, etc. Also an exchange instruction (EXG) exists that has identical syntax to the TFR instruction and has 21 valid forms. In the time it took to read three sentences you just learned 63 new 6809 instructions! As another example, we combined the instructions that set and cleared condition code bits on the 6800 into two 6809 instructions that AND or OR immediate data into the condition code register.

Other significant new instructions include the new 16 bit operations, The D register can be loaded, stored, added to subtracted from, compared, transferred, exchanged, pushed and pulled. All the indexable registers (16 bits) and be loaded, stored and compared. The load effective address instruction can also be used to perform 8 or 16 bit arithmetic on the indexable registers as described later.

Two significant new instructions are the multiple push and multiple instructions on the 6809. With one 2 byte instruction any register or set of registers can be pushed or pulled from wither stack. These instructions greatly decrease the overhead associated with subroutine calls in both assembly and high level language programs. In conjunction with instructions using autoincrement and autodecrement, the 6809 can efficiently emulate a stack computer architecture, which means it should e efficient for Pascal p-code interpreters and the like.

The orders in which the registers are pushed or pulled from the stacked is given in figure 2. Note that not all registers need to be pushed or pulled, but that the order is retained if a subset is pushed. This stacking order is also identical to the order used by all hardware and software interrupts.

One new instruction in the 6809 is a sleeper. The load effective address to indexable register (LEA) instruction calculates the effective address from the indexed addressing mode and deposits that address in an indexable register, rather than loading the data pointed to by the effective address as in a normal load. This instruction was originally created because we wanted a way to let the addressing mode hardware already present in the processor calculate the address of a data object so that it could be passed to a subroutine. After the index addressing modes were completed it was realized the LEA instruction had many more uses, and once again, allowed us to combine other instructions into one powerful instruction. For example to add the D accumulator to the Y index register, the instruction is: LEAY D, Y; to add 500 to the U register: LEAU 500, U; and to add 5 to the value is the S register and transfer the sum to the U register: LEAU 5, S.

In writing position independent read only memory programs it is sometimes necessary to reference data in a table within the same read only memory. This is generally a tedious process even in computers that claim to support position independent code because the register that points to the table must eventually contain an absolute address.

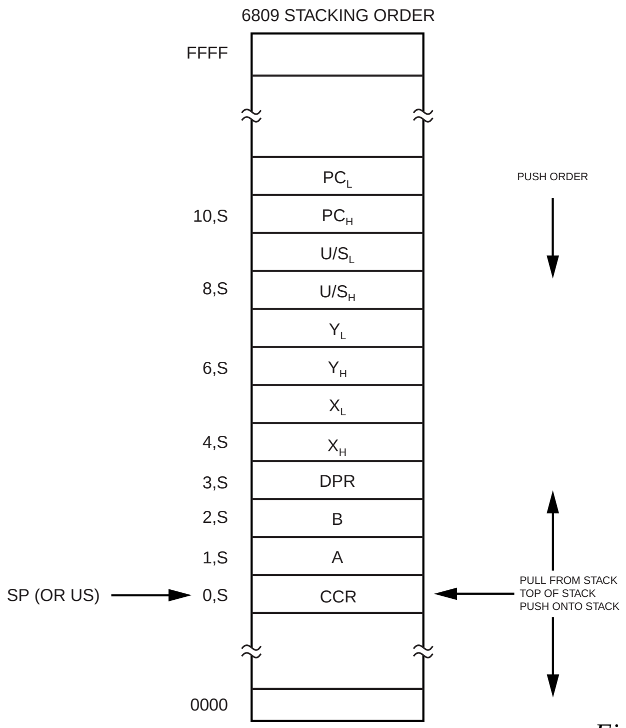


Figure 2: 6809 push/pull and interrupt stacking order.

The LEA instruction, in conjunction with program counter relative addressing, makes this possible with one instruction on the 6809. For example, to put the address of a table DG located in a relative read only memory into indexable register U: LEAU DG, PCR; or to find out where a position independent read only memory is located: LEAY *, PCR (or TFR PC, Y). Our benchmarks show the LEA instruction to be the most used new 6809 instruction by far.

An unsigned 8 bit by 8 bit to 16 bit multiply was provided for the 6809. The A accumulator contains one argument and the B the other. The result is put back onto the A:B (D) accumulator. A multiply was added because multiplied are used for calculating array subscripts, interpolating values and shifting, as well as for more conventional arithmetic calculations. An unsigned multiply was selected because it can be used to form multiprecision multiplies.

Another facet of good programming practice that we wanted to encourage was the use of operating system calls or software interrupts (SWI). The 6800 SWI has been effectively used by 6800 support software for breakpoints and disk operating system calls. That’s nice, but unfortunately there was only one software interrupt, and since Motorola’s software used that one the customer found it difficult to share. The 6809 provides three software interrupts, one of which Motorola promises never to use. It is available for user systems.

One new instruction on the 6809, SYNC, allows external hardware to be synchronized to the

```
00008  0100                         ORG      $100
00009  0100  B6  F002  5            LDA      PIABC      LOAD PIA CONTROL REG. - SIDE B
00010  0103  84  F7    2            ANDA     #$F7       TURN OFF B-SIDE INTERRUPTS
00011  0105  B7  F002  5            STA      PIABC
00012  0108  8E  3000  3            LDX      #BUFFER    GET POINTER TO BUFFER
00013  010B  C6  80    2            LDB      #128       GET SIZE OF TRANSFER
00014  010D  1A  50    3            ORCC     #$50       DISABLE INTERRUPTS
00015                         * WAIT FOR ANY INTERRUPT LINE TO GO LOW
00016  010F  13        2      LOOP  SYNC                SYNCHRONIZE WITH I/O
00017  0110  B6  F000  5            LDA      PIAAD      LOAD A-SIDE DATA; CLEAR INTERRUPT
00018  0113  A7  80    6            STA      ,X+        STORE IN BUFFER
00019  0115  5A        2            DECB                DONE?
00020  0116  26  F7    3            BNE      LOOP       BRANCH IS NOT
00021  0118  B6  F002  5            LDA      PIABC      TURN B-SIDE INTERRUPTS BACK ON
00022  011B  8A  08    2            ORA      #$08
00023  011D  B7  F002  5            STA      PIABC
```

Listing 3: Hardware synchronization using SYNC, a new instruction in the 6809 processor that allows external hardware to be synchronized to the software by using one of the interrupt lines. Very fast instruction sequences can be created using SYNC when it is necessary to process data from very fast input and output devices.

software by using one of their interrupt lines. Using this instruction, very tight, fast instruction sequences can be created when it is necessary to process data from very fast input and output devices. Listing 4 gives an example of the use of SYNC. It is assumed that the A side of the peripheral interface adapter (PIA) is connected to a high speed device that transfers 128 bytes of data to a memory buffer. When the device is ready to send a piece of data, it generates a fast interrupt (FIRQ) from the A side of the peripheral interface adapter. Program lines 12 and 13 set up the transfer; lines 16 through 20 are the synchronization loop. On each pass through the loop, the program waits at the SYNC instruction until any interrupt line is pulled low. When the interrupt line goes low, the processor executed the next instruction. In order to use SYNC, all other devices tied to any of the interrupt line must be disabled. For this example it was assumed that the B side of the peripheral interface adapter also had interrupts enabled; program lines 9 though 11 disable the interrupts and line 21 through 23 reenable it. Line 14 is included to keep the interrupt by the A side of the peripheral interface adapter from going to the interrupt routine. Note that interrupts do not need to be enabled for SYNC to work, and in fact are normally disabled.

Another improvement to the instruction set was brought about by the inclusion of the hardware signal BUSY. BUSY is high during read/modify/write types of instructions to indicate to shared memory multiprocessors that and indivisible operation is in progress. As shown in figure 3 this fact can be used to turn existing instructions into the LOCK and UNLOCK necessary for mutual exclusion of critical sections of the program, or for allocation of resources.

And lastly, never let it be said the 6809 has no SEX appeal—sign extend, that is. The SEX instruction takes an 8 bit two's complement value in the B accumulator and converts it to a 16 bit two's complements value in the D accumulator by extending the most significant bit (sign bit) of B into A.

Table 4 is a convenient way to look to look a all the instructions available on the 6809. The notation first page/second page/third page op codes have the following meaning: first page op codes have only one byte of op code. For example: load A immediate has an op code of hexadecimal 68. All second page op code are preceded by a page op code of 10. For example, the op code for CMPD immediate is hexadecimal 1083 (two bytes). Similarly third page op codes are preceded by a hexadecimal 11. A CMPU immediate is 1183. Some instructions are given two mnemonics as a programmer convenience. For example, ASL and LSL are equivalent. Notice that the long branch op codes LBRA and LBSR were brought onto the first page for increased code efficiency.

Stacks

As mentioned previously, the 6809 has many features that support stack usage. Most modern block structured high level languages make extensive use of stacks. Even though stacks are useful in the typical textbook example of expression evaluation, their major usage in languages such as Pascal is to implement control structures. Microprocessor users already realize the advantage of a stack in nesting interrupts and subroutine calls. Most high level languages also pass data on the stack and allocate temporary local variables

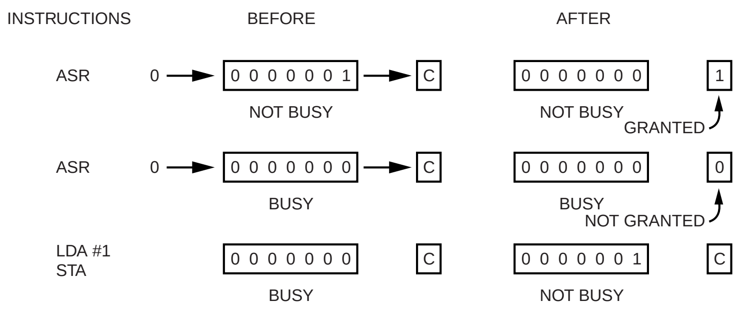


Figure 3: The ASR (arithmetic shift right) instruction is used as a "test and clear" and ST (store) is used for "unbusy." These primitive operations are used for implementing critical section exclusion on the 6809.

from the stack.

Listing 4 and figure 4 show an example of a high level language subroutine linkage. Before calling the subroutine the caller pushed and addresses of two arguments and the answer on the stack and then executed the jump to subroutine which puts the return program counter on the stack. The subroutine then saves the old stack mark pointer on the stack as well as reserving space on the stack for the local variables for the subroutine. In this example, size locations are used but the subroutine body during calculation. At this point the stack mark pointer is set to a new value for this subroutine. The stack mark pointer is used because the S register may very during execution of the subroutine body due to local subroutines, etc. It is much more convenient for the compiler to generate offsets to the parameters is the U is used for this purpose instead of the S.

Once U is set it is used to fetch the two arguments using indexed indirect addressing. The subroutine body presumable does something with the arguments and finishes with an answer in the D register. The subroutine exit saved this value. It then puts the return address in X and restores the previous stack mark pointer. The whole stack is then cleaned up (deleted) and return is made to the caller.

Motorola 6800 users should note that the stack pointers on the 6809 point to the last value pushed on the stack rather than the next free location, as on the 6800. This was done so that autoincrement and autodecrement would be equivalent to pulls and pushes. For example: STA ,-S is

Table 4: 6809 op code map and cycle counts. The numbers by each op code indicate the number of machine cycles required to execute each instruction. When the number contains an I (eg: 4+I), and additional number of machine cycles equaling I may be required (see table 3). The presence of two numbers, with the second on in parentheses, indicate that the instruction involves a branch. The larger number applies if the branch is taken. The notation first page/second page/third page has the following meaning: first page op codes have only one bye of op code (eg: load A immediate has an op code of hexadecimal 86). All page 2 op codes are preceded by a page op code hexadecimal 10 (eg: the op code for CMPD immediate is hexadecimal 1083 – two bytes). Similarly third page op codes are preceded by a hexadecimal 11. A CMPU immediate is 1183. Some instructions are given two mnemonics as a programmer convenience (eg ASL and LSL are equivalent). Notice that the long branch op codes LBRA and LBSR were brought onto the first page to increased code efficiency.

	Most Significant Four Bits																
	DIR		REL		ACCA	ACCB	IND	EXT	IMM	DIR	IND	EXT	IMM	DIR	IND	EXT	
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	
Least Significant Four Bits	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	F	
0000 0	6 NEG	PAGE 2	3 BRA	4+I LEAX	2 NEG	2	6+I	7	2 SUBA	4	4+I	5	2 SUBB	4	4+I	5	0
0001 1	—	PAGE 3	3 BRN/ 5 LBRN	4+I LEAY	—	—	—	—	2 CMPA	4	4+I	5	2 CMPB	4	4+I	5	1
0010 2	—	2 NOP	3 BHI/ 5(6)LBHI	4+I LEAS	—	—	—	—	2 SBCA	4	4+I	5	2 SBCB	4	4+I	5	2
0011 3	6 COM	2 SYNC	3 BLS/ 5(6) LBHS	4+I LEAU	2 COM	2	6+I	7	4,6,6+I,7 SUBD /	5,7,7+I,8 CMPD /		5,7,7+I,8 CMPU	2 ADDD	4	4+I	5	3
0100 4	6 LSR	—	3 BHS 5(6) BCC	5+1/by PSHS	2 LSR	2	6+I	7	2 ANDA	4	4+I	5	2 ANDB	4	4+I	5	4
0101 5	—	—	3 BLO/ 5(6) (BCS)	5+1/by PULS	—	—	—	—	2 BITA	4	4+I	5	2 BITB	4	4+I	5	5
0110 6	6 ROR	5 LBRA	3 BNE/ 5(6) LBNE	5+1/by PSHU	2 ROR	2	6+I	7	2 LDA	4	4+I	5	2 LDB	4	4+I	5	6
0111 7	6 ASR	9 LBSR	3 BEQ/ 5(6) LBEQ	5+1/by PULU	2 ASR	2	6+I	7	—	4 STA	4+I	5	—	4 STB	4+I	5	7
1000 8	6 ASL (LSL)	—	3 BVC/ 5(6) LBVC	—	2 ASL (LSL)	2	6+I	7	2 EORA	4	4+I	5	2 EORB	4	4+I	5	8
1001 9	6 ROL	2 DAA	3 BVS/ 5(6) LBVS	5 RTS	2 ROL	2	6+I	7	2 ADCA	4	4+I	5	2 ADCB	4	4+I	5	9
1010 A	6 DEC	3 ORCC	3 BPL/ 5(6) LBPL	3 ABX	2 DEC	2	6+I	7	2 ORA	4	4+I	5	2 ORB	4	4+I	5	A
1011 B	—	—	3 BMI/ 5(6) LBMI	6/15 RTI	—	—	—	—	2 ADDA	4	4+I	5	2 ADDB	4	4+I	5	B
1100 C	6 INC	3 ANDCC	3 BGE/ 5(6) LBGE	20 CWAI	2 INC	2	6+I	7	4,6,6+I,7 CMPX /	5,7,7+I,8 CMPY /		5,7,7+I,8 CMPS	3 LDD	5	5+I	6	C
1101 D	6 TST	2 SEX	3 BLT/ 5(6) LBLT	11 MUL	2 TST	2	6+I	7	7 BSR	7 JSR	7+I	8	—	5 STD	5+I	6	D
1110 E	3 JMP	8 EXG	3 BGT/ 5(6) LBGT	—	—	—	3+I JMP	4	3,5,5+I,6 LDX /			4,6,6+I,7 LDY	3,5,5+I,6 LDU /			4,6,6+I,7 LDS	E
1111 F	6 CLR	7 TFR	3 BLE/ 5(6) LBLE	19/20/20 SWI/2/3	2 CLR	2	6+I	7	—	5,5+I,6 STX /		6,6+I,7 STY	—	5,5+I,6 STU /		6,6+I,7 STS	F

Table 5: Hexadecimal addresses of the 6809 restart and interrupt vectors.

FFFF	*Restart*
FFFC	*NMI*
FFFA	*SWI*
FFF8	*IRQ*
FFF6	*FIRQ*
FFF4	*SWI2*
FFF2	*SWI3*
FFF0	*Reserved*

equivalent to PSHS A; and LDA ,S+ is equivalent to PULS S. This also means the X and Y registers can be used as stack pointers if the programmer desires. For example: STA ,-X is a push on a stack defined by X. The possible ambiguity between where the stack pointer points on the 6800 and the 6809 may be less of a problem than it seems, since of 6800's TSX becomes the 6809's TFR S, X without adding 1 and TXS becomes a TFR X, S without subtracting 1 – think about it. The only danger is in programs that used the stack pointer as an index register. In these programs the stack pointer may point on location away from where it did previously.

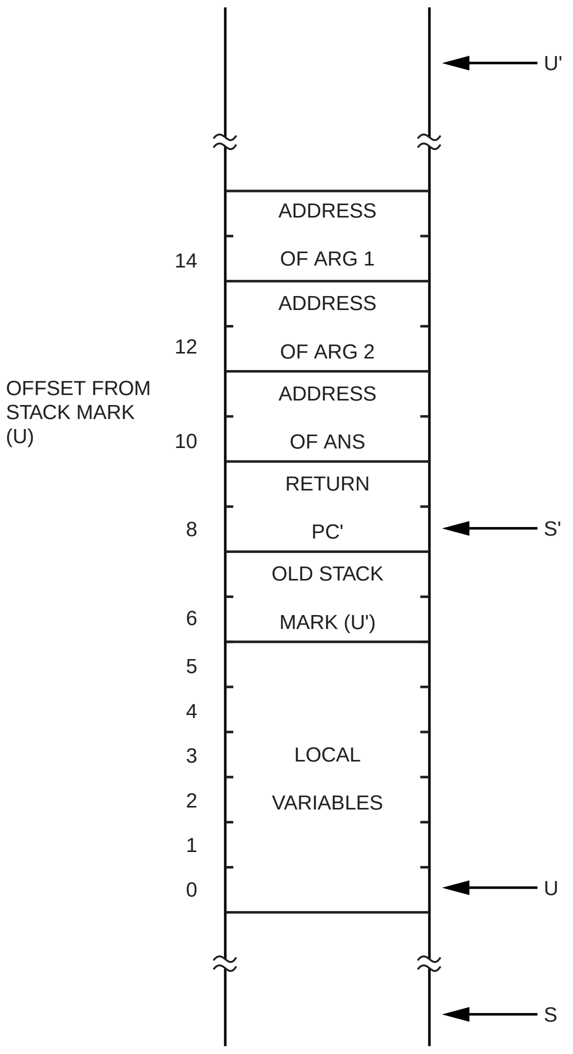
Interrupts

The 6809 has three fully vectored hardware interrupts. The nonmaskable interrupt (NMI) and maskable interrupt (IRQ) are the same as the 6800's NMI and IRQ. The new interrupt is the fast maskable interrupt, or FIRQ, that stacks the program counter and condition code register only on interrupt. Table 5 gives the addresses of the interrupt vectors for the 6809.

A new signal (IACK) has been added that is available anytime an interrupt vector is fetched. This signal together with address bus lined A1 through A3 can be used to implement in interrupt scheme in which each device supplies its own interrupt vector.

The interrupt control and prioritization logic of the 6809 have been defined very carefully – not redundant or indeterminate conditions can exist when several interrupts occur simultaneously. The details of the interrupt structure are precisely defined in Motorola documentation for the 6809.

Part 2, entitled "instruction Set Dead-Ends, Old Trails and Apologies," will be a question and answer discussion about the design philosophy that went into the 6809.■



```
00006   0500  34 40        6    SUBR   PSHS    U         SAVE OLD STACK MARKER
00007   0502  32 66        5           LEAS    6,S       RESERVE LOCAL STORAGE
00008   0504  1F 43        6           TFR     S,U       GET NEW STACK MARKER
00009   0506  EC D8  0E 10             LDD     [14,U]    GET ARGUMENT 1
00010   0509  AE DE  0C 10             LDX     [12,U]    GET ARGUMENT 2
00011                                  *
00012                                  *  SUBROUTINE BODY
00013                                  *
00014   050C  ED D3  0A 10             STD     [10,U]    SAVE ANSWER
00015   050F  AE 48        6           LDX     8,U       GET RETURN ADDRESS
00016   0511  EE 46        6           LDU     6,U       RESTORE U'
00017   0513  32 E8     10 6           LEAS    16,S      POP EVERYTHING OFF STACK
00018   0516  6E 84        3           JMP     ,X        RETURN
```

Listing 4: Use of stacks on the 6809 processor. In this typical high level language subroutine example, U' and S' are the mark stack pointer and the hardware stack pointer, respectively, just prior to the call. U and S are the same registers during execution of the subroutine body. Before calling the subroutine the caller pushes the address of two arguments and the answer on the stack and then executes the jump to subroutine which puts the return program counter on the stack. The subroutine then saves the old stack mark pointer on the stack as well as reserving space on the stack for the local variables for the subroutine (see figure 4).

A Microprocessor for the Revolution: The 6809

Part 2: Instruction Set Dead Ends, Old Trails and Apologies

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

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?

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 simular 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

Photo 1: Layout. Layout designer Tony Riccio adds a line in a large layout cell. Their 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.)

than before. And the macrosequenced PUSH or PULL instructions are *more efficient* than 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 we frequently used in 6800 code because they were more convenient than any other alternatives. 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 64K address space.

Answer 2:

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

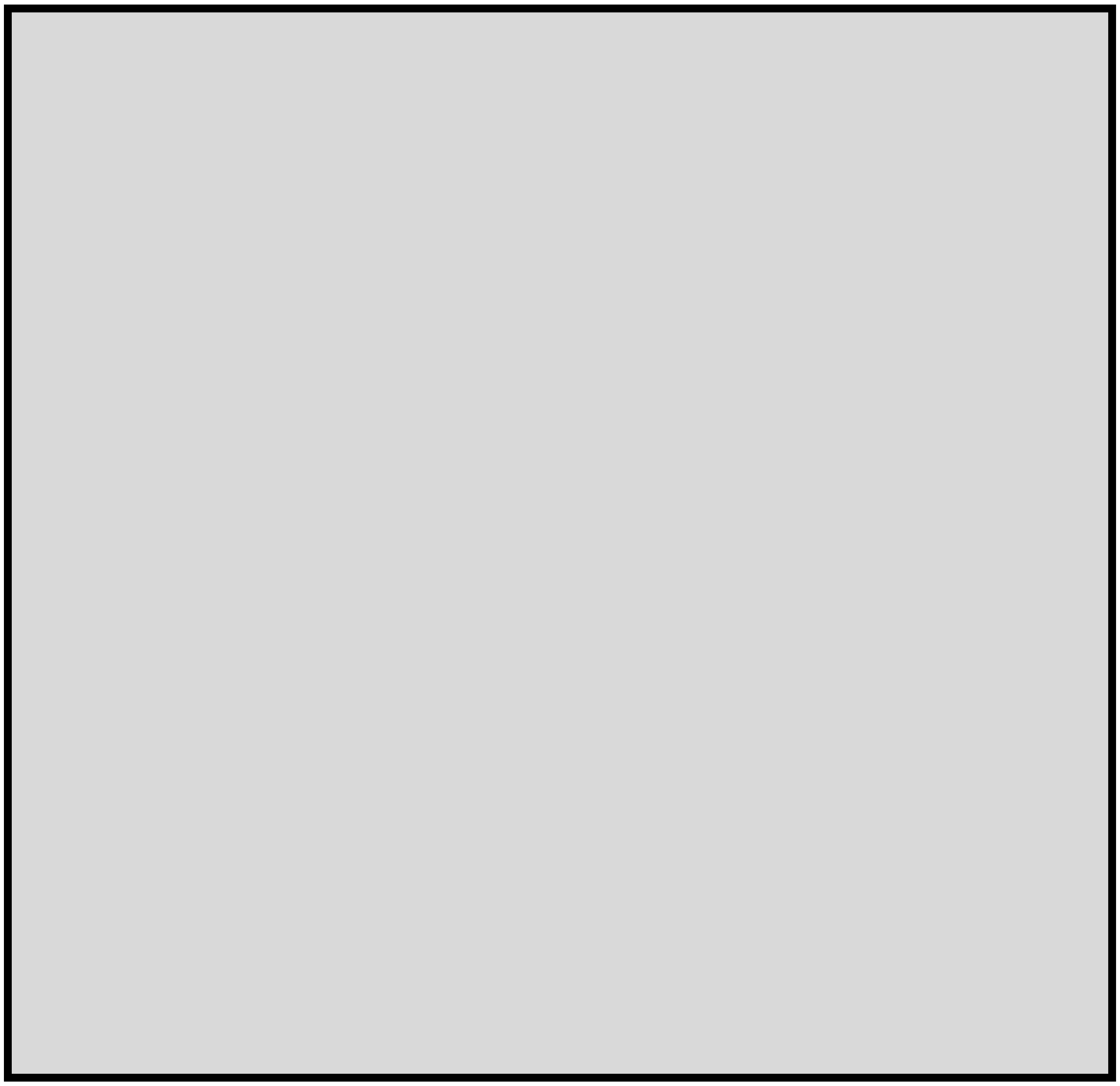


Photo 2: Breadboard design. After partitioning the logic, the mos (metal oxide semiconductor) diagram is translated to TTL. The required ten boards are then designed and built. Meanwhile, Bill Keshlear validates the logic changes on the master copy of the logic diagrams, since they will imply changes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

stood 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, sorting 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 (16K 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, moved, 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 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 there fore necessary to arbitrate among them, switching exactly one processor onto the common bus, and allowing it to proceed with its memory access while the other 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 index 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 little longer, but is more general, and probably is located is 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 is 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 couse. 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 included 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 singed 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 useable pointers plus some others. This basic dichotomy of data and pointers to data exists in practice, and is therefore rarely a problem with out implementation. But the EXG instruction allows convenient manipulation between these groups in any unusual circumstances.

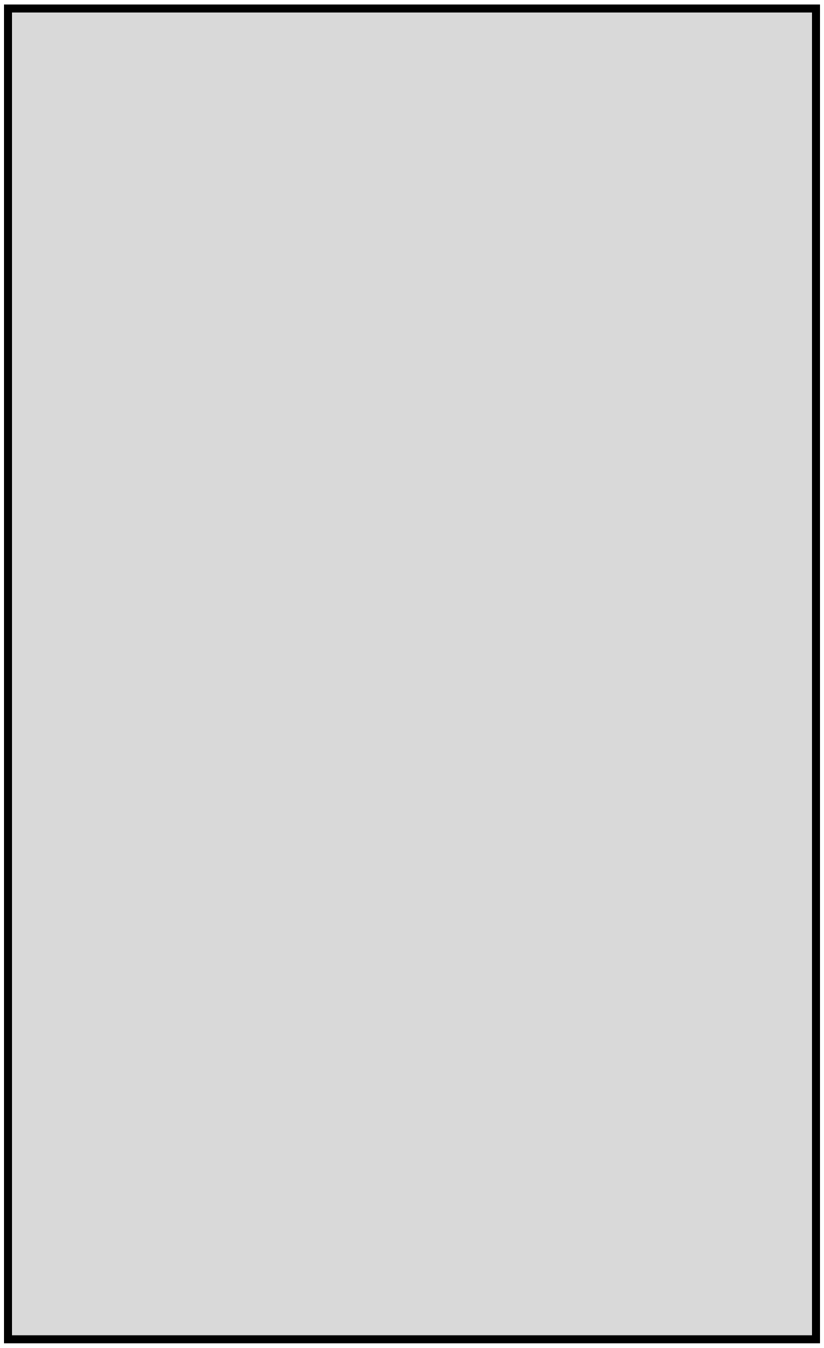


Photo 3: Visual inspection. Some of the gross processing errors or problems that occur with probing equipment can be detected visually. Here, lead production operator Mary Celedon checks a 6802 wafer.

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 addressing so the different constant offsets 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.

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

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 interrupt as SWI does, thus allowing use even in interrupt derived 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 for 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 a 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 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 the 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 change 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 result 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 of 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:

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.

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

Answer 14:

The direct page register is one of those possible 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 address 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 direct page register. All forms of absolute addressing for 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: subroutine) 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 -

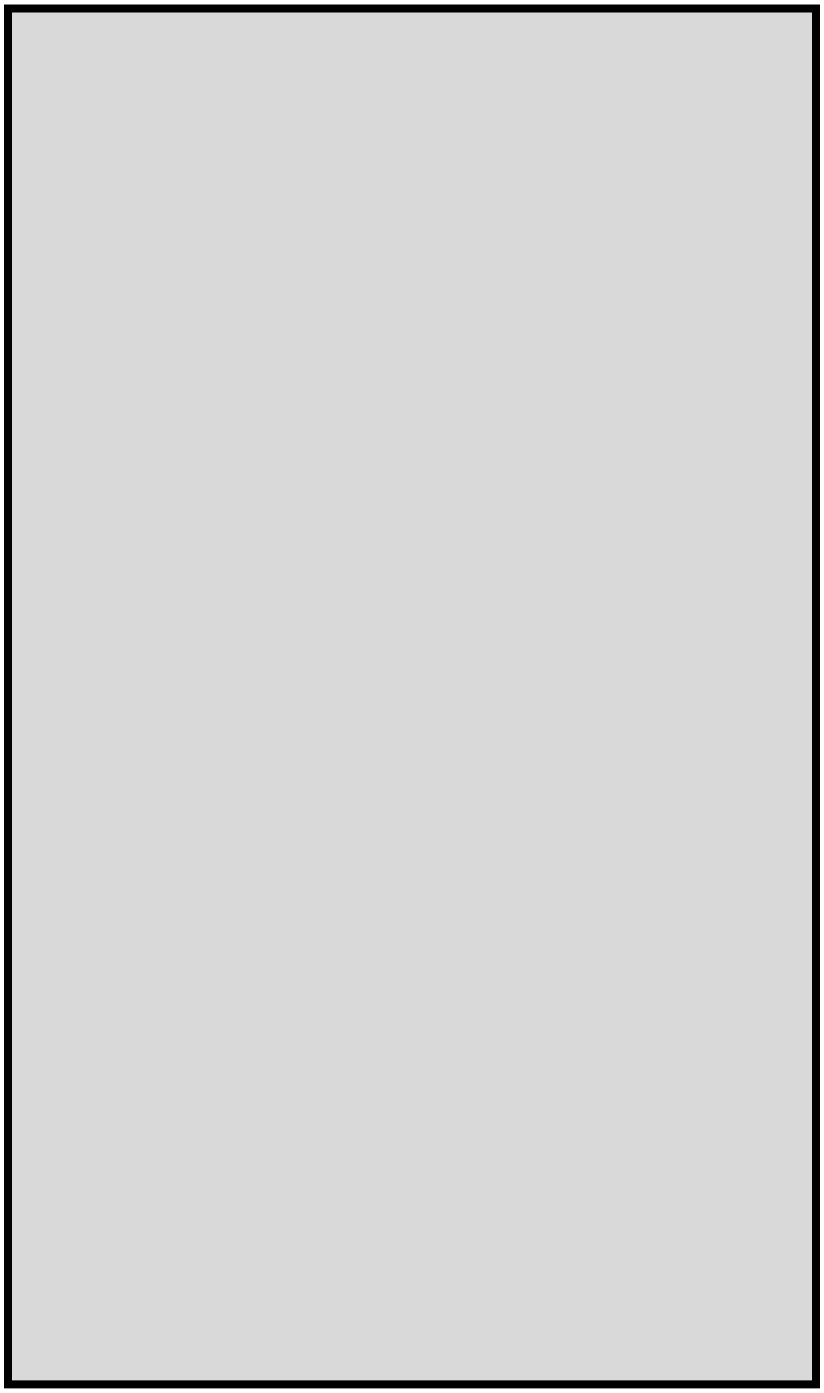


Photo 5: First silicon engineering analysis. Logic and circuit design engineer Bob Thompson tracks down a weak node in the first batch of 6801 chips. The 6801 die is packaged, but not sealed, so that internal nodes may be probed while in operation. Viewing through a microscope, a probe can be placed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

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 b 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.

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 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 16K 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 cause interrupt execution of FCB $1063 times, then RTI), or can support hardware refresh (a direct memory

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 cary.
ANDA, ANDB	And memory with accumulator.
ANDCC	And memory with condition code register.
ASLA, ASLB, ASL	Arithmetic shift left accumulator or memory.
ASRA, ASRB, ASR	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 bits).
NEGA, NEGB, NEG	Negate accumulator or memory.
ORA, ORB	Or memory with accumulator.
ORCC	Or immediate with condition code register.
PSHS (register)$^{8}_{0}$	Push register(s) on hardware stack.
PSHU (register)$^{8}_{0}$	Push register(s) on user stack.
PULS (register)$^{8}_{0}$	Pull register(s) on hardware stack.
PULU (register)$^{8}_{0}$	Pull register(s) on 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 barrow.
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**
ADD	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 pointer register.
CMPX, CMPY, CMPU, CMPS	Compare pointer register.
LEAX, LEAY, LEAS, LEAU	Load effective address into pointer register.
SEX	Sign extend.
TFR register, register	Transfer register to register.
EXG register, register	Exchange register to register.
PSHS (register)$^{8}_{0}$	Push register(s) on hardware stack.
PSHU (register)$^{8}_{0}$	Push register(s) on user stack.
PULS (register)$^{8}_{0}$	Pull register(s) on hardware stack.
PULU (register)$^{8}_{0}$	Pull register(s) on user stack.

Table 1: continued:

INDEXED ADDRESSING MODES

Mnemonic	**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 clear.
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 if not equal.
BPL, BPL	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 interrupt/
SYNC	Syncrhonize with interrupt line.

access [DMA] sequence, or isolatec 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 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 deign philosophy.■