Single-chip microcomputers can be easy to program

by BILL HUSTON
Motorola Inc.
Austin, Texas

ABSTRACT

Most single-chip microcomputers (MCUs) use the split-memory Harvard architecture. A few single-chips trace their architectural heritage to large computers due to the common-memory Von Neumann organization. The major differences are that a Harvard-based MCU costs less in its undistorted form, and a Von Neumann-based MCU is more expandable and easier to program.

Since the traits of Harvard-based single-chips are quite well known, though perhaps not by that name, the focus is placed on the programming benefits of a Von Neumann MCU. Programming costs can be lowered while increasing program reliability. Data organizations can be more flexible in both RAM and ROM. Program changes can be incorporated more quickly. The generalized instruction set is easier to understand. The M6805 family of MCUs is used to illustrate these benefits.
ARCHITECTURAL COMPARISONS

Like most major products, the single-chip microcomputer has evolved in a series of stages rather than being the inspired creation of a genius. All of the popular 4-bit single-chip microcomputers (MCUs) and many of the 8-bit MCUs are derived from the evolution of the calculator. Some 8-bit MCUs have instead evolved down from larger computers. These two diverse evolutionary paths are identified by comparing the two architectures that have resulted.

Harvard Architecture

The unique trait of the architecture shown in Figure 1 is the separate memory organization for programs (ROM) and data (RAM). Each type of memory has a dedicated address register. The ROM address register is the program counter, but the RAM address register has various names. Separate address registers permit register lengths and interconnections to be optimized. For example, a 6-bit RAM address can be used with a 10-bit ROM address.

With the separate memory architecture, data read from ROM are fed directly to the instruction decoder. Similarly, the RAM output goes only to the ALU. Thus, data widths of 4 bits in the RAM and ALU are not incompatible with an 8-bit ROM instruction size.

The split program and data memory architecture is sometimes called the Harvard architecture (or Aiken architecture). This designation contrasts it to the Von Neumann (or Princeton) architecture of all large computers today. The Harvard architecture was used in some of the very first electromechanical and electronic computers, built under the direction of Professor Howard Aiken at Harvard. Memory technology was of course very rudimentary in the 1940s. Since separate storage techniques were used for programs (paper tape) and data (telephone 10-step relays), the separate memory architecture was a natural. As with MCUs today, the hardware components and the interconnections are fewer with split dedicated memories.

The Harvard Mark I computer was used for over 10 years as a high-precision calculator of mathematical reference data such as navigation and ballistics tables. When the processor usage is straightforward, the Harvard architecture is fine, even superior. The problems arise when the needs become more complex.

For example, to allow a subroutine, a program counter save register is placed beside the PC. This does not dramatically disturb the interconnect efficiency of the Harvard architecture. The Harvard benefits dissipate quickly when three or more PC save registers are cascaded together into a costly amount of silicon. Sometimes the program is permitted to read and write into the PC save register, which adds more dedicated interconnects to Figure 1, as well as encountering the problem of unequal word sizes. Sometimes an MCU includes a stack pointer and saves the PC in RAM, which doubles the RAM read/write paths in Figure 1.

Address calculations are another example of Harvard architecture difficulties. Figure 1 shows that all MCU implementations have a path from the RAM address register to the ALU to permit calculations. RAM data structure sizes are limited when a 4-bit ALU is used with a 6-bit RAM address. Harvard MCUs use one or more instructions to calculate the content of the RAM address register. Then one or more instructions are used to obtain and operate on the RAM content. There are no single instructions that calculate the RAM address and then operate on the RAM content.

Some MCUs have no provision for calculating ROM addresses. The ROM address register is not available to the ALU, so relative addressing is not possible. In such cases it is not possible to read the content of a data table in ROM. Thus, a straightforward BCD-to-7-segment conversion has to be implemented in an I/O PLA. In some cases the Harvard architecture is further distorted to allow a program to read and write to a ROM address register. In such cases there are now two inputs to the ROM decoder in Figure 1, the program counter and a program-accessible ROM address register.

As the computer pioneers of the late 1940s and early 1950s discovered, the Harvard architecture has severe limits when it comes to generalized uses. Thus the Harvard architecture in today's more advanced single-chip MCUs includes numerous distortions. As a result, the economic motivation for the Harvard architecture in a calculator is lost in a general-purpose MCU. Extra dedicated registers and ALU data paths are added to the silicon area of an MCU, which increases the price. The Harvard architecture is also more difficult (expensive) to program.

It has been successfully shown with the M6805 family that a Von Neumann architecture MCU can be both lower in cost (less silicon die area) and easier to program.
Von Neumann Architecture

Figure 2 shows the fundamental architectural difference to be a common addressable area for RAM and ROM, and I/O as well. Rather than use point-to-point interconnecting as in Figure 1, Figure 2 shows common data and address busses. The program registers are also more generalized.

Figure 2. Von Neumann architecture single-chip MCU

Professor John Von Neumann at Princeton first documented the concept of a program stored in a common memory space with data. The chief benefit is the inherent ability to operate upon addresses as easily as data. Program and data table pointers can be saved in RAM. Indexing the other address calculations can be included.

The Von Neumann architecture has some shortcomings. The common bus saves interconnect area only when there are enough points tapping onto the bus to justify the three-state control needed to manage the use of the bidirectional bus. All address and data elements must be standardized to the bus width.

In current implementations, 8-bit busses, registers, and ALU are used, which means that some elements are larger than in 4-bit MCUs. Elements larger than the bus-addresses, for example—occupy more than 1 bus cycle. With an 8-bit bus, expansion to 16 bits of addressability is as easy as handling a 10-bit address.

The remainder of this paper focuses on the program benefits of the Von Neumann architecture, particularly as applied to the M6805 MCU family.

PROGRAM AND PROGRAMMER EFFICIENCY

It was once considered sufficient simply to have a very low-cost programmable IC. The programs written were short, and the programming effort was to be amortized over a large number of units. This view is obsolete today in many applications. The applications are more complex than the microwave ovens of a few years ago. Programs are not just written once and forgotten; they are changed, in some cases many times. Program changeability costs should also be considered when amortizing program costs.

The Von Neumann type of MCU architecture also permits greater program design flexibility. Memory use tradeoffs are more easily made. System hardware functions can be taken over by the program. The tools are available to allow programs to be more reliable. The most important efficiency factor for MCU programs is efficiency of ROM use—fitting the most features into a given ROM size.

Program Changeability

Only unsuccessful programs are never changed. Since a project is seldom started that is planned to be unsuccessful, all projects need to plan for program changeability. Field testing of a prototype points up faults in the original program as well as desirable improvements. The sources of program requirements (customers and marketers, for example) frequently conclude that what they asked for is not exactly what is needed. Similarly, the managers, marketers, and customers always come up with new features that would be desirable. These are just some of the sources of changes to the original product.

There are also changes to the program that generate derivative products. It is difficult to hide the fact that the single-chip is programmable. Everyone wants to take advantage of the programmable IC to suggest derivative products. Changeability must be designed in from the beginning.

Programming costs thus include the cost of incorporating program changes as well as the initial programming effort. Frequently the changes are incorporated by a different programmer. Program changeability costs thus also include the time it takes a new programmer to figure out what the original programmer did.

The MCU architecture can limit future extensions of the program to include additional functions. In such cases the program changeability costs include reprogramming for a new MCU. Specialized programming techniques that take advantage of odd MCU features or use unused memory in odd ways also limit future changeability. Major reprogramming costs can be avoided by using generalized MCU architectures, which do not tempt the programmer to use odd quirks in the inevitable attempts to get seven pounds of functions into a five-pound ROM sack. The features of the end product can be so tightly interwoven with each other and with the given memory organization that changes, even some apparently simple ones, can send the programmer back to Square 1.

The architecture of a single-chip MCU has more impact on the cost of program changes than at first suspected. The Von Neumann architecture allows programs to be written faster initially, understood more quickly by a different programmer, and changed more rapidly.

Fewer Lines of Code

"The programming time is directly proportional to the number of program statements."

This axiom has been widely accepted for programming projects, from compiler-language business-data-processing programs to assembly-language microprocessor applications. The axiom is also applicable to single-chips.

The functional definition, functional flow chart, and user documentation effort are rather independent of the MCU chosen. However, the detail flow charts, coding, program
checkout, and program documentation phases are proportional to the number of lines of code. In typical projects, coding and checkout represent the bulk of the programming effort.

If an MCU architecture permits the program to be written with fewer lines of code, it saves programming expense. Benchmarks have shown that M6805 family programs need about half as many lines of code to accomplish a given task as a typical 4-bit MCU. The benchmarks include full applications as well as typical comparison subroutines. Thus 50% of the program coding and checkout time can be saved.

More details of the M6805 family architecture are included later, but a few of the features that contribute to the program savings are listed here. Address calculations, including table look-up indexing, are a part of the instruction, not separate instructions that must precede the operation. In two-operand instructions such as add, AND, and compare, one operand is an addressable memory byte, which saves frequent register loading. Memory bits and bytes can be modified directly, without disturbing any registers, in a single instruction such as set a bit and increment a byte. All I/O pins may be set, cleared, or tested with one instruction. Interrupts automatically save and restore all registers.

As applications become more complex, programming time is becoming a larger part of the end product cost. A larger benefit in many cases is that the end product will be available sooner. Many products using MCUs go into competitive markets where saving a few months can measurably increase market share. When changes can be incorporated faster, the new product variations can also reach the market ahead of the competition.

ROM Versus RAM Tradeoffs

MCU programmers frequently get caught with not enough memory. Product cost targets can block switching to an MCU with more memory. So effort must be expended in redesigning the program until it fits.

When only ROM or RAM is overloaded, tradeoff techniques can be used to decrease the use of one at the expense of the other. The common memory field of the Von Neumann architecture is again shown to be an advantage. ROM and RAM are equally accessible, so functions can more easily be moved back and forth.

The flexibility of having any number of subroutine levels gives the user considerable control over the mix of ROM and RAM used. The more subroutine levels needed, the more RAM used for subroutine return addresses. So when spare RAM is available, the code can be shortened with more subroutines. When RAM is overfilled, fewer subroutine levels can be used by increasing ROM usage.

Efficient bit and byte handling instructions, such as that of the M6805 family processors, allow RAM data to be packed, multiple elements per byte.

I/O Versus ROM Tradeoffs

The increased instruction and addressing mode sophistication of a Von Neumann MCU sometimes allows previous hardware functions to be taken over by the software. Since hardware-versus-software tradeoffs are application-dependent, only generalized examples are cited.

Some MCU applications use an off-chip A/D converter. There are a series of alternative approaches that can be considered. One approach is to use an MCU that includes an on-chip A/D. Second, the analog value can also be converted to a variable frequency or pulse width, which is measured either with a timer on the MCU or with a program. A third method is to use an interrupt program to count the cycles it takes for an external ramp to match on an external comparator. Perhaps money can also be saved in the analog sensor or in the accuracy of the A/D conversion. A lower-cost sensor might produce nonlinear outputs, but the program could compensate for the nonlinearity by using an indexed conversion table or a smoothing formula.

The goal is the lowest total system cost, not the lowest MCU cost. There are frequently opportunities to consider doing by program functions that require external hardware with other MCUs.

Program Errors

Program reliability should be considered in relation to single-chip MCUs. It may seem improbable for an error to go undetected that is serious enough to require scrapping end products, but it has occurred. Such scrappage is part of the cost of programming. Software costs are treated as amortizable costs. The exception is program errors that turn into recurring costs. Program errors occur as a result of insufficient program checkout, which frequently is due to hurriedly incorporated changes.

Rather than initiating end product scrappage, program errors more often cause a quirk to show up in the end product. Such errors cause a series of recurring costs (costs proportional to the quantities in use, not one-time costs). Instruction manuals are expanded to explain the quirk. The service people are trained not to interpret the quirk as a failure. Time is taken to explain the quirk to complaining customers. These are direct, measurable costs of program errors.

An indirect cost of program errors is loss of good will. Customers who have to live with a recognized quirk are irritated. Some will take their business to a competitor the next time. These are not one-time programming costs.

Program unreliabilities also bring in the risk of legal liability. Some program errors could be construed as causing loss of life, limb, or property.

The use of sound programming techniques is clearly the best way to reduce the risk of program unreliabilities. The architecture of the MCU can contribute to encouraging good programming techniques.

Errors are inclined to be proportional to the number of lines of code it takes to write a given program. A processor that uses fewer statements to perform a function, is also easier to keep clear in the mind of the programmer. As implied earlier, orderly change incorporation presents the best opportunity to reduce the error risk. In this case, the otherwise unmeasurable factors of an easy-to-understand, consistent instruction set with few oddities has major value. When the application functions are tightly interlinked with memory and I/O traits, changes can be extensive and thus error-prone.
The watchword is to be sensitive to program reliability and to put some value on an MCU architecture that encourages better programming.

**ROM Usage Efficiency**

Using the least ROM area is one of the more important criteria used to select single-chip MCUs. The number of single-byte instructions in the repertoire is not a good measure of ROM efficiency. The question is not whether one thousand instructions fit into a 1K ROM, but rather the number of system functions that can be programmed into a 1K ROM. This brings up the subject of benchmarks.

It is tempting to gather or devise half a dozen routines that are felt to be typical of the intended application and implement them in two or three competing instruction sets. Such a tradeoff is vulnerable to human bias, perhaps unintentional, on two major fronts. First, the programmer is likely to be more experienced in one processor and thus less likely to produce optimal code on the alternate processors. Second, the choice of the benchmark routines is clearly a simplification of the application and likely to be slanted to the programming techniques used on one or a few processors.

In spite of the risks, comparisons obviously need to be made. Steps can be taken to reduce, as far as possible, these biases. But why not go one more step?

The initial writing of an MCU program tends to be short compared to programs on larger computers. Many single-chips have been programmed in a month or two. So if two MCUs are in contention, program them both for the complete application. Then the comparison benchmark is not just a few isolated routines, but also all the overhead that it takes to use those routines in a practical application. Small benchmarks can serve to evaluate speed-critical program paths in response-time-sensitive applications. But MCU users are usually more concerned with ROM efficiency than with throughput. ROM usage efficiency is not as easily judged from small benchmarks.

**THE M6805 FAMILY ARCHITECTURE**

In covering the benefits and shortcomings of Von Neumann-based single-chip microcomputer architectures, some of the architectural traits of the M6805 family of MCUs have been alluded to. This report is thus concluded with some details of the M6805 family architecture. How well have these MCUs capitalized on the shortcomings of the popular Harvard architecture MCUs? Is the M6805 family really easier to program, and does programming ease have monetary value? The result is an MCU architecture which is more economic (has a smaller die area) than the popular 8-bit Harvard architecture MCUs and at the same time includes the big-computer features that are usable in a single-chip.

Such programming tools as indexed look-up tables, many subroutine nesting levels, single-instruction memory modification, single-instruction bit test and modify, and common access methods for all addressable locations, are direct user benefits of the computer heritage as opposed to the calculator heritage. With these tools, programs are written easier and faster and are easier to modify and more reliable.

**One Address Map**

A striking feature of a Von Neumann architecture is the common memory space for the ROM and RAM. The M6805 family extends the advantage by allocating space in the address map for I/O registers. The common address map is shown in Figure 3. The instructions include short addressing modes for more ROM-efficient access to the first 256 addressable locations. The most frequently accessed data elements are thus concentrated in the quick-access 256-byte page zero. Present implementations include 64 bytes and 112 bytes of RAM in various versions, but future versions could easily include more or less RAM.

**ROM Areas**

A portion of the user ROM is included in the first 256 locations to allow quick access to frequently used subroutines and to allow quick access to look-up tables.

In addition to the user ROM, all M6805 family ROM-based MCUs include self-check ROM. A small program is included for factory wafer-level testing and is available for user testing if desired. The self-check ROM area is not counted as user ROM and does not in any way reduce factory final testing to data sheet specifications. Some users are using the callable self-check subroutines implemented in most versions for functional confirmation when coming out of reset. Some are using a low-cost self-check tester for functional screening of parts before PC board assembly. The EPROM versions do not use the small mask ROM for self-checking, but rather for bootstrap self-programming of the user EPROM.

The highest memory addresses are user ROM for the interrupt and reset vectors. The vectors are 16-bits (2 ROM bytes) designating the interrupt program starting address. Separate vectors are included for the external interrupt; the timer interrupt; the software interrupt; the power-up reset program; and, in the CMOS versions, the stand-by recovery (Wait mode) program.

**Addressable I/O**

The first 16 addressable locations are reserved for the on-chip I/O registers. I/O is thus accessible to all instructions using the ROM efficient short addressing modes. I/O data
may be read or written (load and store) as bits or bytes. But I/O bytes may also be operated upon (AND, add, compare, etc.).

Current MCUs include up to four 8-bit ports. Each port read/write register occupies 1 memory byte. The ports include a second byte, the data direction register, which determines whether each I/O pin is an input or a driven output.

The 4 ports thus occupy 8 addressable bytes. The timer accounts for 2 more bytes, one for the 8-bit counter and the other for timer control. The second external interrupt available on some versions occupies 1 byte. The A/D converter on some versions uses 1 byte for the digitized result and 1 byte for A/D control. The EPROM versions include a register to control the self-programming of the EPROM. One family version includes an on-chip phase-locked loop for frequency synthesis that uses 2 I/O bytes for the variable divider.

The Register Set

Figure 4 shows that when a generalized address map is used, only five program registers are needed to provide a powerful instruction set. The specialized registers of the Harvard-type architecture are not needed.

<table>
<thead>
<tr>
<th>8 BITS</th>
<th>A ACCUMULATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 BITS</td>
<td>X INDEX REGISTER AND ADDITIONAL ACCUMULATOR</td>
</tr>
<tr>
<td>5 or 6 BITS</td>
<td>SP STACK REGISTER</td>
</tr>
<tr>
<td>11 TO 13 BITS</td>
<td>PC PROGRAM COUNTER</td>
</tr>
<tr>
<td>5 BITS</td>
<td>CC CONDITION CODE BITS</td>
</tr>
</tbody>
</table>

S AND PC LENGTHS VARY WITH THE AMOUNT OF MEMORY IMPLEMENTED

Figure 4. Register

The accumulator is used for arithmetic and logical operations. The program counter is from 11 to 13 bits long, depending on the amount of memory implemented.

The index register has two uses. The three indexed addressing modes use X to contain a variable that is added to a value provided within the instruction. The X register is also an auxiliary accumulator. Many of the register manipulation instructions that operate on A also are used with X.

Additional general-purpose registers are not needed, since instructions are available to modify RAM locations directly without disturbing A or X. Examples are increment a byte, set or clear a bit, and test a bit or byte.

The stack pointer is initialized to the highest RAM address. The variable portion is 5 or 6 bits to limit the maximum stack length to 31 or 63 bytes. A subroutine call uses 2 stack bytes to save the return address. The automatic interrupts use 5 stack bytes to save the A, X, PC, and CC registers. The 5-bit stack pointer thus permits up to 13 nested subroutines, assuming 1 interrupt level, (31 - 5)/2 = 13. The 6-bit stack pointer allows for 29 subroutine levels. Both subroutine nesting levels are safely beyond that which could normally be used in a single-chip program. It is convenient, however, to let the programmer determine the needed subroutine levels rather than have the limit established by the architecture.

The condition code register is five individual status bits that are treated as a register when an interrupt save occurs. Four of the CC bits represent the results of the last data byte accessed or register operation performed. These permit subsequent testing with conditional branch instructions. The four result conditions are carry (or borrow), half carry (for BCD adds), all zeros byte, and negative (bit 7 set). The fifth CC bit is the interrupt mask, which enables all on-chip interrupts.

Future Expandability

A frequent restriction of Harvard architecture MCUs is a limit on expanding the memory or I/O size in future versions. In most cases the maximum RAM size is limited within the op-code field of instructions that load the RAM address register. There are a number of popular architectures that cannot use more than 64 bytes of addressable RAM.

A Von Neumann architecture has few restrictions on the mix of ROM and RAM. The only address limit imposed by the M6805 family architecture is that the maximum addressability is 64K, though no current versions include a full 16-bit address. The program counter, all the long addressing mode instructions, and the subroutine and interrupt save space all accommodate a 16-bit address field with no architectural changes.

Numerous System Configurations

A major benefit of architectural expandability is that many family versions can be introduced in a short time. Eleven versions of the M6805 family are already available, and more are on the way.

Three technologies are presently represented: HMOS, CMOS, and EPROM. ROM sizes range from 1K to 4K, with RAMs from 64 to 112 bytes. The 28- and 40-pin packages typically permit 20 and 32 I/O pins respectively. For evaluation, prototyping, and smaller production runs, both EPROM and ROM-less versions are offered. Some versions include an on-chip 8-bit A/D converter. Another includes a frequency synthesizer for RF applications. Standby RAM capability is included in some versions. Most include high-current output drivers.

Automatic Interrupts

Interrupts are the primary tool allowing a program to synchronize to real-time I/O events. Single-chip MCU applications have become I/O-intensive. Inputs and outputs of diverse natures must be accepted and generated. Frequently, tight timing relationships must be measured or maintained. Multiple timing relationships must be coordinated, sometimes at higher speeds.

Some Harvard-architecture-based MCUs have no interrupt facilities because there is no place to store the return address. The modernized Harvard MCUs have added an interrupt, which is frequently only a fixed subroutine call. Fully automatic interrupts save all program registers, not just the pro-
gram counter. The interrupt program thus need not waste ROM bytes and time storing all of the registers.

Efficient interrupt tools make complex real-time MCU interfaces possible.

Ten Addressing Modes

Another benefit of the Von Neumann architecture is that the common address map allows the instruction set to be enhanced by providing more addressing modes.

Figure 5 shows that the M6805 family has added four addressing modes to the M6800 instruction set while dropping only one 16-bit mode. The new bit manipulation capability is particularly appropriate to the controller environments that use single-chip MCUs. The extra indexing modes ease the table look-up task, the most useful indexing function in controllers, as well as permitting better ROM use.

<table>
<thead>
<tr>
<th>ADDRESSING MODE</th>
<th>M6800</th>
<th>M6801</th>
<th>M6805 FAMILY</th>
</tr>
</thead>
<tbody>
<tr>
<td>INHERENT (OPERAND IN OPCODE)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMMEDIATE (OPERAND FOLLOWS OPCODE)</td>
<td>6 BITS</td>
<td>16 BITS</td>
<td></td>
</tr>
<tr>
<td>ABSOLUTE (OPERAND ADDRESS FOLLOWS OPCODE)</td>
<td>256 LOCATIONS (DIRECT)</td>
<td>64K LOCATIONS (EXTENDED)</td>
<td></td>
</tr>
<tr>
<td>RELATIVE</td>
<td>PC + 128 (BRANCHES)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexed (FOR TABLE ACCESSES)</td>
<td>EA = X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EA = x + 8-BIT VALUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EA = x + 16-BIT VALUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit Manipulation</td>
<td>BIT SET CLEAR</td>
<td>BRANCH ON BIT</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5. Ten addressing modes

The inherent addressing mode includes the single-byte register reference and control instructions, which do not reference memory. Immediate addressing is the inclusion of an 8-bit data value in the second byte of a 2-byte instruction.

Short and long absolute addressing, called the direct and extended modes, includes the memory address in the instruction. The first 256 most frequently accessed bytes, the RAM, I/O, and part of the ROM, are accessed with a 2-byte instruction. A 3-byte extended instruction accesses any byte in the address map.

Relative addressing allows the conditional branch instructions to reach a program within the range of -127 to +129 of the instruction. An absolute jump can then reach anywhere else in memory.

The three indexed addressing modes add flexibility in the organization of the data in memory. In a single-byte indexed instruction, the effective address is the contents of the index register. The index register thus contains an 8-bit pointer to the data byte to be accessed. As such, the X pointer can reference any RAM byte, any I/O byte, or a portion of the ROM. This no-offset indexing is similar to the only available RAM access method on typical Harvard-architecture-based MCUs. The program calculates an address, puts it in a RAM address register, and then accesses the data. No-offset indexing is most frequently used in the M6805 family processor to scan down a data table looking at each entry.

The second and third indexed addressing modes are short and long table look-up indexing. The 8-bit contents of the index register are added to an 8-bit or a 16-bit value contained in the instruction to determine the effective address of the data to be accessed. In table look-up use, the instruction contains the address of the beginning of the table, and X contains a displacement into the table. Short offset indexing includes an 8-bit address within a 2-byte instruction; long indexing uses a 3-byte instruction to include a 16-bit table address. With short indexing the table must begin in the first 256 locations, but the displacement may create an effective address up to 255 locations beyond page zero.

Most microprocessors and 8-bit single-chip microcomputers have been good at byte manipulation. To be controller efficient, the M6805 family has added single-instruction bit manipulation and test capability. Any bit of any byte within the first 256 addressable bytes may be set or cleared. All the I/O pin and all the on-chip RAM bits may thus be individually changed. The addressed byte is read, the designated bit is changed, and the modified byte is written back into memory, all in one instruction. The two addresses—the direct (page zero) byte address and the bit address—are both contained in a 2-byte instruction. The read-modify-write cycle does not disturb the A or X program registers.

The second bit addressing mode is the single-instruction bit test capability. These are 3-byte instructions that include three addresses. First is the 8-bit direct address of any byte within the first 256 bytes. Second is a 3-bit address of the byte within the byte that is to be tested. Third is an 8-bit relative conditional branch displacement. One instruction is used to branch anywhere within the range of -126 to +130 locations of the instruction, depending on whether the designated bit is set or clear.

Instruction Set

The 10 addressing modes presented above bring much of the power to the M6805 family instruction set. The addressing mode flexibility allows many specialized instructions to be avoided. The instructions themselves are generalized; this feature, when combined with the addressing modes, produces a remarkably powerful processor in a small silicon area.

Except for a few miscellaneous instructions, all instructions are combined with one of the addressing modes to access memory. The 10 addressing modes combine with 59 basic instructions (61 instructions in the CMOS versions) to produce 207 total instructions (209 in CMOS). The programmer gets the power of 207 (209) instructions while having to learn only 59 (61) instructions plus 10 addressing modes.

The most frequently used M6805 family instructions are the memory reference instructions. Included are four move instructions, four arithmetic instructions, three logical instructions, three compare instructions, and two jump instructions. Except for the jumps, these are all two-operand instructions. One operand is taken from memory via the addressing mode, and the other operand is the A or X register. The result of the arithmetic and logical instructions is put into the A accumulator. The compare instructions perform a subtract (for magnitude compare) or an AND (bit compare) of the two values without modifying the registers or memory. Six of the major
addressing modes apply to each of the 16 memory reference instructions. Both short and long absolute addressing allows the memory operand (or jump address) to be anywhere in the address map and to be more efficiently accessed if within the first 256 locations. All three indexing modes are applied to all 16 instructions. An indexed table retrieval need not simply load a byte; it may also add, AND, compare, etc., a table byte with A. Immediate addressing is also usable with all the memory reference instructions, except the jumps.

Programming time is saved in several ways. Operations are performed during the same instruction as a memory retrieval (load). Magnitude and logical compares are accomplished without first saving the state of a register. Diverse memory data organizations can be used, since retrievals can use absolute addressing, register pointer indexing, or table look-up indexing.

The next class of instructions are the register and memory modification instructions. Included are the typical register manipulation functions of increment, decrement, complement, clear, shift, and rotate. A test without modifying is also included in this set. The unusual thing about these instructions is that they may be used to operate on memory data as well as both the A and X registers. An instruction like the memory increment can displace up to five instructions in another processor: Save the content of A, load memory byte, increment A, store incremented byte, restore previously saved content of A. All three short addressing modes are applicable to the memory modification instructions. Short absolute and both short indexing methods are included. Since ROM bytes are not modifiable, the long addressing modes have little use with these instructions.

The bit manipulation and test capability has already been covered. The four instructions are bit set, bit clear, branch on bit set, and branch on bit clear.

Ten of the 14 conditional branches test the condition code bits for the result of the last data operation. This set includes tests for zero, negative, carry, half carry, and above zero. The states of the interrupt mask bit and the interrupt pin are also testable. All these conditional branches allow branching on the true or false state. It is convenient that the branch is a relative arithmetic displacement (+ or −128 nominally), which has no page boundaries. In many MCUs the branch is permitted only within a fixed page.

The list of 13 miscellaneous instructions is short so that few specialized instructions need be learned. A regular (generalized, not specialized) register set and instruction set leave very few specialized functions to be performed. Six instructions are register reference functions: interregister transfers and the CC bit manipulations. There are four stack manipulation instructions associated with the interrupts and subroutines: return from subroutine and interrupt, call software interrupt, and reset stack pointer. The M6805 family versions implemented in CMOS include the Stop and Wait instructions.

Since CMOS ICs use dramatically less power when not operating, two program-initiated standby modes are included. The differences in the two modes are the conditions that cause the processor to resume execution. In the Stop mode the external interrupt pin causes the processor to restart. In the Wait mode either the external interrupt or the timer interrupt causes execution to restart. The timer interrupt permits the processor to be restarted at regular intervals. The timer interrupt can initiate a cycle consisting of scanning all inputs, processing the inputs, saving needed results, and generating needed outputs. When this cycle is complete, the processor can be put back into the Wait state. The battery drain is thus the average of the operating current and the stand-by current for the operating-to-stand-by duty cycle.

FULL PROGRAM PERFORMANCE

As single-chip microcomputer applications are becoming more complex, the real-time program needs typical of larger computers are becoming necessary.

Program costs must be kept down. The programs must be capable of being easily changed for future products, and easily documented to allow a different programmer to incorporate changes. MCU architectures can permit efficient ROM use. The classic computer types of architectures offer more tools for memory optimization. RAM usage and I/O features can be traded off with ROM use.

Generalized instructions with many addressing modes allow large-computer performance for an 8-bit MCU. Single instruction table manipulations are included in the M6805 family of MCUs. Single instruction memory bit and byte manipulations are included. Memory bits and bytes can be tested without disturbing the program registers. A common address map is used to allow ROM and I/O space to be accessed with as much flexibility and ease as RAM. The address map is designed for instruction-efficient access to the most frequently used data elements without making any memory inaccessible. There are no architectural restrictions on the amount of memory or on the implemented mixture of ROM and RAM.

The programmer’s single-chips are Von Neumann architectures like the M6805 family.