Using the AT89C2051 Microcontroller as a Virtual Machine

AT89C2051 Flash Microcontroller

Application Note

It is often cited that what differentiates an embedded microcontroller from other general purpose computing devices is its integration into a larger electrical or electro-mechanical system. While this is generally true, the fact remains that processors of widely differing capability and architecture are employed in this regard.

Unfortunately, this broad explanation defines nothing; we are still left to contend with everything from full-blown embedded PCs to the smallest self-contained single-chip microcontrollers. Within this expansive realm, conventional wisdom may lead to the conclusion that the smallest microcontrollers are only appropriate for driving small-scale applications with very limited processing requirements. While this is unquestionably the case in many instances, a class of applications exists that mandates a relatively high level of program complexity within severely constrained space limitations. Faced with such a seeming paradox, engineers often feel they have no choice but to adopt a less than optimal design strategy using a larger microcontroller than originally intended.

The problem, of course, is one of limited resources. Functional complexity implies a non-trivial program, and the greater the functional complexity the larger the program. Even as the capability of small single-chip microcontrollers continuously inches upwards, application requirements seem to grow at a commensurate rate. Trying to hit such a moving target is difficult at best.

The economy of using a microcontroller with just enough processing power for a given application is a potent incentive to find just the right fit. Of course, this only works when the system requirements are thoroughly understood and clearly defined. Since such a design normally has little reserve capacity, it is usually hard pressed to handle features beyond those originally specified. Should additional capabilities eventually become a necessity, the result could be a system that runs out of steam and an engineer that runs out of options. Such are the perils of designing on the edge.

Atmel's AT89C2051 offers capabilities that far exceed those of competing devices of similar size. This opens up potential design opportunities that were simply unattainable with previously available parts. Housed in a 20-pin package, Atmel's miniature microcontroller retains all the major features of the 8051 architecture. Furthermore, the AT89C2051 includes all of the 8051's "special" pins including the external interrupts, UART transmit and receive lines, and the external timer controls. Even though the AT89C2051 significantly ups the processing ante, it would seem that there are limits to what you can accomplish with any single-chip microcontroller.

This dilemma is nothing new. The traditional way of dealing with such limitations has been to operate the microcontroller in external memory mode. Common sense would indicate the hopelessness of applying such an approach to the AT89C2051. After all, the AT89C2051 is truly a single-chip design that does not even possess an external bus structure. It turns out that the situation is not hopeless at all.

w

Processor Simulation

The concept of microprocessor simulation is widely used and well understood. Simulation is often used for development

purposes where a PC program models a specific processor's architecture and interprets and executes its binary instruction set. Using this technique enables one to develop, test, and debug algorithms that will ultimately be combined into a larger program. Such a program will eventually run on a standalone microprocessor or microcontroller. Using simulation early in the design cycle is attractive because it allows you to start developing code long before the actual target hardware is available.

Processor simulation has also been applied to simulate entire computing systems. In this context, existing application programs, in their native binary format, have been coerced to run on various computers powered by completely different processors. For obvious reasons, the performance resulting from such an approach often proves to be disappointing. This does not necessarily have to be the case if the implementation is designed for a specific purpose. Factors effecting performance efficiency include the host processor's strengths and limitations, the specific types of operations that are to be simulated, and, to an extent, the language the original program is written in.

Virtual Processor Simulation

Many developmental simulators have been produced that emulate the functions of popular processors and microcontrollers using standard desktop computers. The same principles can be utilized at the other end of the spectrum; there are cases where running a simulation on a small microcontroller can be put to an advantage. In this case, however, the benefit is not derived from simulating a known processor, but one that offers inherent advantages tailored to solving the specific problem at hand. The implication, of course, points to the design of a virtual processor. The idea is based on the premise of using a real processor to implement a virtual device specifically designed to suit the special needs of a particular application. In other words, designing the tool set for a particular job.

The fact is that adopting such a methodology can ultimately result in an architecture that can be pressed to serve as an efficient vehicle for a number of specialized tasks. Details including the fundamental architecture, instruction set, and memory model can be approached with total freedom. But, can such an approach provide the level of performance demanded by embedded applications?

Efficiency and Overhead

To illustrate that efficiency is a subjective matter, consider what happens when a typical C program is compiled to run on an 8051 processor. It's inconceivable that, on such an architecture, any C statement will effectively compile down to any corresponding 8051 instruction. A single C statement invariably results in the execution of multiple instruction steps. It follows that, given an efficient simulated instruction set, the simulation overhead might account for a very small percentage of the overall execution time.

The key behind making this premise work is to devise an instruction set and processor architecture that's conducive to performing the types of operations that a C compiler naturally generates. In such an implementation, the contrived instruction set essentially amounts to an intermediate language. The op codes merely serve as a vehicle for succinctly conveying the compiler's directives to the target processor for execution.

The target processor, while performing the functions of a simulator, interprets the intermediate instructions to perform the functions specified in the original high level language source statements. The resulting efficiency can be quite tolerable since the bulk of the instructions would execute regardless of whether they were emitted directly by the compiler or invoked by the simulation kernel.

It turns out the performance penalty of such an approach is, to a great extent, dependent on the way the program memory itself is implemented. Since the AT89C2051 has no external bus structure it makes sense to use a serial bus to access the program memory. Using I^2C for this purpose provides the required flexibility along with reasonable throughput.

Selecting I^2C as a memory bus presents the potential of choosing from a wide variety of EEPROM memory devices. The most favorable configuration is Atmel's AT24C64 that offers 8K bytes of storage in an 8-pin package. Utilizing extended 16-bit addressing, the AT24C64 provides linear access to the entire internal memory array. And although a lot of functionality can be crammed into a single chip, additional devices can easily be added in 8K increments to handle very complex applications. Up to eight AT24C64s can simultaneously reside on the I^2C bus providing a full 64K of storage while using just two wires.

Of course, serial memory access does come at a cost. In this case the expense comes in the form of access time. To an extent, this is moderated by the fact that the AT24C64 can operate at a 400 kHz clock rate (standard I^2C is specified at a maximum of 100 kHz). Remember however, that I^2C can exact a significant performance penalty because a substantial percentage of its bandwidth can be consumed for control functions.

The greatest overhead burden that I^2C imposes involves the transfer of addressing information. For every random read or write, a 16-bit address must be transmitted along with the extra overhead necessary to coordinate bus control for both the addressing phase and the data manipulation phase. Under such conditions, actual data movement could be swamped by the requisite overhead resulting in unacceptable performance degradation. Fortunately, I^2C provides a means of eliminating much of this wasteful activity.

The AT24C64, like all other I^2C memory devices, contains an internal auto-increment address generator. Using this feature, once addressability is established, data can be continually streamed in a sequential fashion. As each byte is read and acknowledged the internal address generator increments in preparation for the next byte transfer. The AT24C64 sets the maximum speed limit at 400 kHz but I^2C does not impose a lower limit. Effectively, the minimum frequency can drop all the way to DC. As a result, it's acceptable to suspend a sequential transfer for as long as necessary.

Utilizing these features, communications can be sped up considerably. The ramifications are particularly significant when the memory is used to store an executable program. For example, once an address is written into the AT24C64, data can be fetched in a continual stream until the program branches or, if multiple AT24C64's are used, until it becomes necessary to cross into the next chip. At these points it's necessary to explicitly reload the internal address generator. Normally, however, the majority of the accesses will be sequential, resulting in greatly reduced overhead.

Processor Simulators and Language Interpreters

It's important to note the distinction between language specific interpreters that implement a defined language such as BASIC, and a processor simulator that interprets a low level binary instruction set. A tokenized BASIC interpreter, while quite efficient in executing the commands that are explicitly implemented as part of the language, is strictly confined to what the language supports. The inherent efficiency of an interpreted language comes at the expense of flexibility.

In contrast, a processor simulator, that deals with a true binary instruction set, enjoys total freedom in combining these basic op codes into larger functional entities in almost limitless permutations. Just like a real processor, a simulated processor can utilize its instruction set for standard and custom C library functions, floating point libraries, device drivers, etc.

The Virtual Machine — An Imaginary Processor

The processor to be described is imaginary in the sense that its architecture and instruction set are original and unique. Realize, however, that this is not just a toy or an intellectual diversion—from an implementation standpoint it is quite real. The fundamental concept has been successfully ported to a variety of processor architectures. A version exists that runs on a personal computer that is suitable for demonstration and development purposes. The most promising small-system port has been to the AT89C2051 due to the microcontroller's standard processing core and integrated peripheral set. The basic 8K Virtual Machine is schematically depicted in Figure 1. The circuit's simplicity reveals that this is primarily a software implementation—the definitive soft machine.

This imaginary processor, the product of Dunfield Development Systems, has served in various applications providing reliable solutions to real world problems where a standard configuration was not necessary, optimal, or practical. That this Virtual Machine also goes by the name "C-FLEA" affirms its optimization for efficiently rendering the output of a C language code generator.

The prime currency of a processor is time. Viewed in this context, the expense of complexity can prove unacceptably burdensome. Taking this into consideration, the Virtual Machine, based on a simple 16-bit architecture that incorporates only four registers, is the epitome of simplicity. This register set comprises an accumulator, index register, stack pointer, and program counter. Appendix A provides detailed information about the Virtual Machine architecture and instruction set. Refer to Table 1 for a description of the fundamental resource set.

Although the Virtual Machine performs all operations to 16-bit precision, the needs of many embedded systems resolve to 8 bits. To facilitate working with this common denominator, the Virtual Machine stores data in little endian format (low byte first) which facilitates the use of a given variable's base address to refer to either an 8-bit or 16-bit quantity. Interestingly, the architecture provides no user accessible flags. When invoking a compare instruction, internal flags persist only long enough to accommodate the ensuing branch instruction or the intervening compare modifiers (which are described later).

This spartan register set is made workable by the inclusion of a variety of addressing modes that excel at the types of stack manipulations that are central to the canonical C implementation. The Virtual Machine's memory access instructions, detailed in Table 2, include the following addressing modes: immediate (8 or 16 bit), direct, indirect through index register (with or without offset), indirect through stack with offset, top of stack, and indirect through top of stack (remove or leave value on stack).

The bulk of the virtual instruction set is presented in Table 3. These instructions include memory access instructions, arithmetic instruction, and logical instructions. In keeping with the previously established proposition, most can return either bytes or words.

Since the compare instructions are designed to only determine equality, the instruction set is augmented by a set of special compare modifiers. Using these, nuances of relative (signed and unsigned) magnitude can be coerced from the basic compare instructions. These modifiers are described in Table 4.

Program branching is supported using the relatively conventional set of conditional and unconditional jump instructions shown in Table 5. Versions are provided for both near and far destination targets to enhance code efficiency. Note the inclusion of the SWITCH instruction which proves especially useful since the “normal” compare instructions destroy the contents of the accumulator when returning the result of the compare operation.

Table 6 presents the stack manipulation set. Included are common functions such as CALL, RETurn, and PUSH. Conspicuously absent is an explicit POP instruction. The corresponding functionality is provided by the various addressing modes that, by default, manipulate the top of the stack. For instance, POP A is synonymous with LD S+. Additional instructions are included to facilitate stack frame creation and destruction that is a necessary function of the C language implementation.

Finally, the virtual instruction set is rounded with a number of miscellaneous instructions shown in Table 7. For the most part, these perform standard functions that should be self explanatory. The input/output instructions are special in that they offer an implementation specific avenue for establishing certain peripheral functions as instructions. Remember that, even though, the virtual instruction set offers the programmer total freedom to construct any kind of computational sequence, all I/O operations are dependent on the support coded into the Virtual Machine kernel. Essentially, the simulation kernel is the software embodiment of a microprocessor architecture. Naturally, the goal is to provide a general purpose engine capable of serving in a wide variety of real embedded systems.

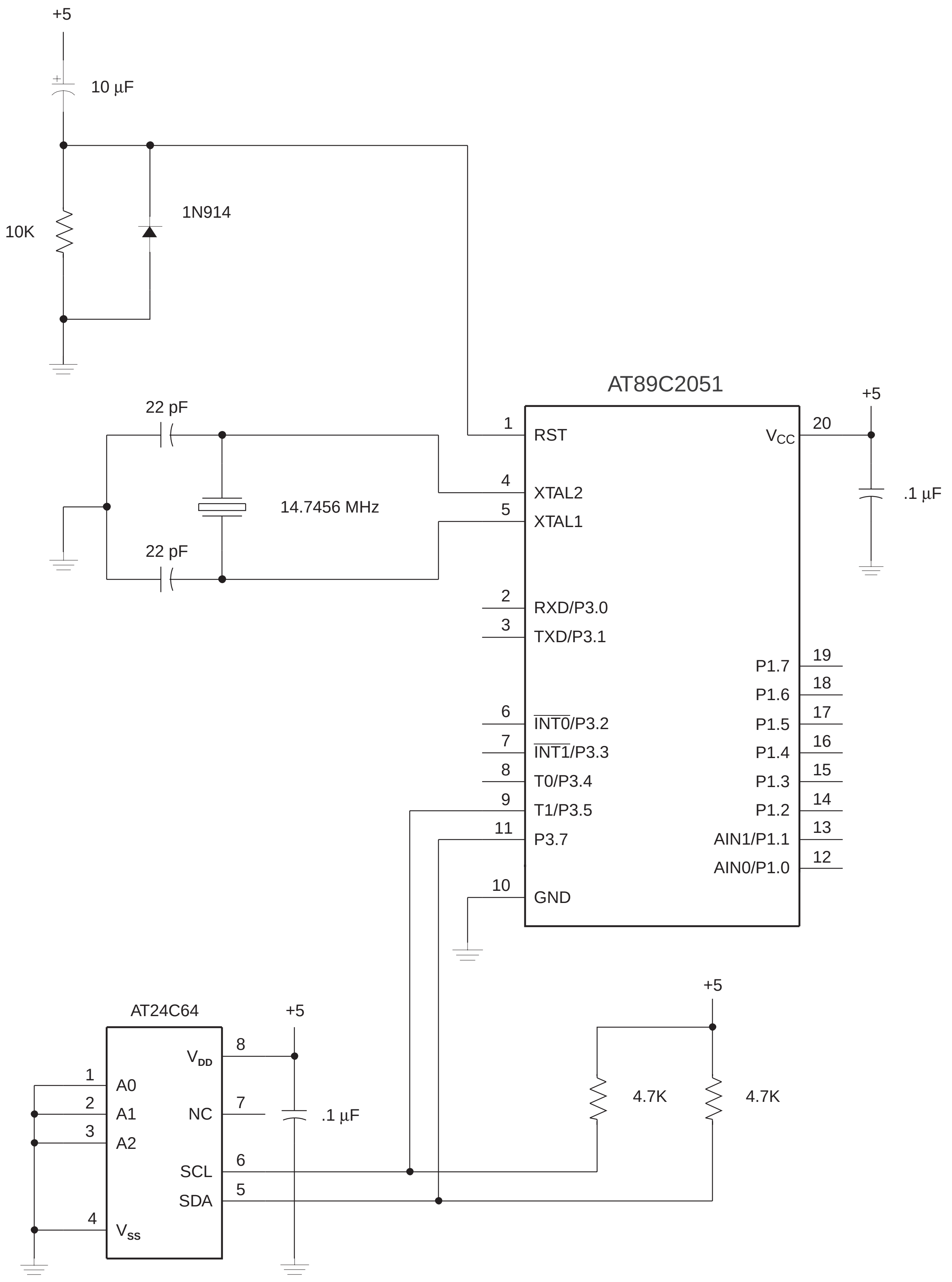
A significant number of op codes remain unassigned and are available for future use.

Initial Program Loader

While not actually part of the Virtual Machine, the simulation kernel contains a built-in program loader utility. This operates serially and is invoked following a system reset by a sequence of special commands from a utility program running on the host computer. In addition to transferring the load image to the Virtual Processor, the PC program provides a number of features which include a simulator (that can hook into the target’s logical and physical I/O subsystem) and a console window for performing user I/O to the target system. Since the Virtual Machine’s code generator emits a standard Intel HEX file format, the use of the PC utility program is optional.

In principle, there is no reason why an AT24C64 cannot be programmed externally using a standard device programmer just as you would program an EPROM for a use in a typical embedded computer. Although workable, this approach would, at the least, prove cumbersome throughout the development cycle. The difficulty of this approach would be exacerbated in a system using multiple memory chips. Obviously, it would be completely unworkable in the event a Virtual Machine computer was rendered as a surface mount assembly.

Figure 1. 8K Virtual Machine



Virtual Machine I/O

The Virtual Machine handles physical I/O (as well as virtual I/O) through the use of input/output instructions. It is natural to reserve certain I/O addresses for on-chip functions such as serial I/O and for access to the AT89C2051's on-chip parallel I/O ports. Additional I/O addresses are assigned to second level functions such as serial port configuration and direct I/O bit set and clear functions. The bit manipulation functions are important when an on-chip parallel port is simultaneously used for both input and output.

Consider the ramifications of performing a standard read/modify/write operation on such a port. Normally this would be accomplished by reading the port via an IN instruction, performing a logical operation on the value, and writing the modified data back to the port using OUT. Should an input pin be externally pulling low while the port was being read, the unfortunate outcome of this exercise would be to render that line permanently jammed low and unusable for any further input!

Additional virtual input/output devices are provided for functions such as time-of-day clock, general system timing, pulse width modulation, and pulse accumulation. These are implemented as background interrupt service routines and are accessed as simple input/output devices.

Serviceable as the basic I/O resource set may be, it's often necessary to provide ancillary I/O functions external to the processor. The Virtual Machine accomplishes this transparently by passing any undefined I/O addresses to the external peripheral trap. This handler uses a secondary I^2C bus to implement an auxiliary external peripheral/memory channel.

Here, the instruction's I/O address is taken as the I^2C slave address. For output operations data is passed via the low byte of the virtual accumulator. Input functions return data in the low byte of the virtual accumulator. In both cases the accumulator's high byte is utilized to convey completion status and can be interrogated to determine the outcome of the requested operation. The result code reflects the status of the data link transfer and either indicates valid completion or fault status. Should a fault be reported it could be the result of a peripheral in busy status, a device that is not present, or a legitimate peripheral malfunction.

Virtual Machine Assembly

To clarify the relationship between the Virtual Machine kernel, a virtual assembly language library function, and a virtual C application program, an example is in order. This will also serve to illustrate how easily communication to the outside world can be orchestrated in such an environment.

The program depicted in Listing 1 is a library function that supports console I/O using a special I^2C user I/O module. (This is the same module that was detailed in the application note "A Framework for Peripheral Expansion.") The user I/O module contains a standard 20 x 4 LCD, 4 x 4 keypad, and beeper. These are supported using two I^2C-to-parallel port expanders. The underlying premise is that, once the data transport mechanism is hidden, the I^2C ports can be used just like any conventional I/O ports. In this case the concealment is complete since the I^2C driver is written in the AT89C2051's native instruction set and is therefore completely invisible and inaccessible to a virtual program running on the Virtual Machine. Reading and writing to I^2C devices now becomes strictly a matter of IN and OUT.

Looking again to listing 1 reveals how virtual instructions can be combined to generate a useful program. Far from being constrictive, the virtual instruction set yields an economy of expression while retaining a great deal of flexibility. The limited number of registers does, however, require a reliance on the stack for parameter passing and for holding intermediate results. This shouldn't be surprising considering the fact that the Virtual Machine is primarily designed as a C engine. Anyone familiar with the way a C compiler utilizes the stack frame should have little trouble adapting these concepts to writing efficient assembler programs.

Virtual Machine Compilation

Not much can be said about the compilation process for the Virtual Machine. This is truly a virtue since, after all, the primary purpose of a language compiler is to insulate the programmer from the complexities of a particular processor. To those experienced with C compilers for 8051 processors, the most notable omission here is the absence of the multiplicity of libraries for the various memory models that are so necessary when working with a native 8051. Recall that the Virtual Machine supports a single, eminently reasonable, flat 64K memory space.

Listing 2 reveals that there is nothing special and, more importantly, that there are no artificial limitations inherent in a C program written for the Virtual Machine. This program implements a simple calculator function that uses the I^2C user I/O module as the system console device and utilizes the long math functions from the Virtual Machine math library. The actual functionality behind this module is secondary. What is more important is that it looks like a C program and behaves like a C program — and can be abused like a C program. In short, it can be coerced to do the things you need a typical embedded program to do.

Pint Sized Computer

Although tiny by any scale of measure, the Virtual machine behaves the way you would expect any self respecting processor to behave, virtual or not. More to the point, the Virtual Machine in actuality is a fully functional computer system. You would be hard pressed to find a smaller, fully functional, computer with comparable capability that adequately supports the C programming language.

Using surface mount manufacturing techniques, a fully operational computer can be constructed to fit into an area the size of a postage stamp. The Virtual Machine's large program memory space, combined with its secondary I^2C memory/peripheral bus, makes the architecture suitable for handling a number of relatively ambitious embedded projects. Its minuscule size allows it to be placed anywhere.

Sources

If you are interested in experimenting with the Virtual Machine concept, a fully operational PC based Virtual Machine simulator, C compiler with libraries, and assembler are available for downloading from the the Dunfield Development Systems bulletin board at (613) 256-6289. For availability of the Virtual Machine processor, development system, and support software contact Mid-Tech Computing Devices USA; P.O. Box 218; Stafford, CT 06075 (860) 684-2442.

To obtain the listing 1 and listing 2 codes, please download from Atmel's Web Site or BBS.

Appendix A — Virtual Machine Architecture

Table 1. Fundamental Resource Set

ACC	16-bit accumulator 8-bit accesses are auto zero-filled
INDEX	16-bit addressing register, cannot be manipulated as 8 bits
SP	16-bit stack pointer
PC	16-bit program counter

Table 2. General Addressing Modes

Syntax	Coding	Description
#n	x0 ii(ii)	Immediate (8 or 16-bit operand)
aaaa	x1 dd dd	Direct memory address
I	x2	Indirect (through INDEX register) no offset
n,I	x3 oo	Indirect (through INDEX register) with 8-bit offset
n,S	x4 oo	Indirect (through SP) with 8-bit offset
S+	x5	On Top of Stack (remove)
[S+]	x6	Indirect through TOS (remove)
[S]	x7	Indirect through TOS (leave on stack)

Notes: 1. Addressing mode is in lower 3 bits of op code.
2. Mode S+ always pops 16 bits from stack. Only 16-bit values can be pushed.
3. Modes [S+] and [S] will always use a 16-bit address on the top pf the stack but the final target can be 8 or 16 bits.

Table 3. Memory Addressing Instructions

Name	Description	(Unused Address Modes)
LD	Load ACC 16 bits	
LDB	Load ACC 8 bits	
ADD	Add 16 bits	
ADDB	Add 8 bits	
SUB	Subtract 16 bits	
SUBB	Subtract 8 bits	
MUL	Multiply by 16 bits	
MULB	Multiply by 8 bits	
DIV	Divide by 16 bits	
DIVB	Divide by 8 bits	
AND	And 16 bits	
ANDB	And 8 bits	
OR	OR 16 bits	
ORB	OR 8 bits	
XOR	XOR 16 bits	
XORB	XOR 8 bits	
CMP	Compare 16 bits (ACC = 1 if equal)	
CMPB	Compare 8 bits	
LDI	Load INDEX (16 bits only)	
LEAI	Load INDEX with address	(00, 05)
ST	Store ACC 16 bits	(00, 05)
STB	Store ACC 8 bits	(00, 05)
STI	Store INDEX (16 bits only)	(00, 05)
SHR	Shift right (8-bit count only)	
SHL	Shift left (8-bit count only)	

Notes:
1. ACC always contains 16 valid bits. All operations are performed in 16-bit precision. 8-bit operands are zero-filled when they are fetched.
2. SI decrements when data is pushed
3. Data is stored in little endian format.
4. There are no user accessible flags. In the case of CMP, internal flags are maintaned only long enough to accommoate the LT-UGE instruction.

Table 4. Compare Modifiers

Name	Description
LT	ACC = 1 if less than (signed)
LE	ACC = 1 if less than or equal (signed)
GT	ACC = 1 if greater than (signed)
GE	ACC = 1 if greater than of equal (signed)
ULT	ACC = 1 if lower than (unsigned)
ULE	ACC = 1 if lower than or same (unsigned)
UGT	ACC = 1 if higher than (unsigned)
UGE	ACC = 1 if higher than or same (unsigned)

Notes: 1. These instructions must immediately follow a CMP instruction.

2. NOT instruction is used to implement explicit NE.

Table 5. Jump Instructions

Name	Description
JMP	Long jump (16-bit absolute)
JZ	Long jump if ACC=0 (16-bit absolute)
JNZ	Long jump if ACC!=0 (16-bit absolute)
SJMP	Short jump (8-bit PC offset)
SJZ	Short jump if ACC=0 (8-bit PC offset)
SJNZ	Short jump if ACC!=0 (8-bit PC offset)
IJMP	Indirect jump (Address in ACC)
SWITCH	Jump through switch table (ACC=value, INDEX=table)

Note: 1. Switch table format: addr1, value1, addr2, value2, ... 0, default addr

Table 6. Stack Manipulation Instructions

Name	Description
CALL	Call subroutine (16-bit absolute address)
RET	Return from subroutine
ALLOC	Allocate space on stack (8-bit value)
FREE	Release space on stack (8-bit value)
PUSHA	Push ACC on stack
PUSHI	Push INDEX on stack
TAS	Copy ACC to SP
TSA	Copy SP to ACC

Note: 1. Explicit POP instruction are not required since various addressing modes use and remove the top item on stack.

Table 7. Miscellaneous Instructions

Name	Description
CLR	Zero ACC
COM	Complement ACC (ACC = ACC XOR FFFF)
NEG	Negate ACC (ACC = 0 - ACC)
NOT	ACC = 1 if ACC was 0, else ACC = 0
INC	Increment ACC
DEC	Decrement ACC
TAI	Copy ACC to INDEX
TIA	Copy INDEX to ACC
ADAI	Add ACC to INDEX
ALT	Get alternate result from MUL/DIV
OUT	Output byte in ACC to PORT
IN	Read byte from PORT
SYS	System interface function

Note: 1. ALT obtains the remainder after DIV and obtains the high word after a multiply. This instruction must be executed immediately after the MUL or DIV.