M68HC11

Reference Manual

M68HC11
Microcontrollers

M68HC11RM/D
Rev. 6.1

M68HC11

Reference Manual

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.freescale.com

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision History

Date	Revision Level	Description	Page Number(s)
June, 2001	4	Reformatted to meet current publications standards	
February, 2002	5	**Index** — Updated	631
		Figure 9-4. Baud Rate Control Register (BAUD) — Address designation corrected to $102B	327
April, 2002	6	ADD Instruction —Corrected table head from ADCA to ADDA	496
		AND Instruction — Corrected table head from ADCA to ANDA	498
		ASL Instruction —Corrected table heads ASLA (IMM) to ASLA (INH) and ASLB (DIR) to ASLB (INH)	499
		ASR Instruction — Corrected table heads ASRA (IMM) to ASRA (INH) and ASRB (DIR) to ASRB (INH)	501
		BIT Instruction — Corrected second table entry for Data under BITA (IND,Y) from A5 to A5 and under BITB (IND,Y) from E5 to E5	510
		CLR Instruction — Corrected table head from CLRA (IMM) to CLRA (INH) and CLRB (DIR) to CLRB (INH)	529
		STY Instruction — Corrected second table entry for Data under STY (IND,X) EE to EF	584
		WAI Instruction — Changed I bit designation from 1 to —	598

This section discusses the CPU architecture, addressing modes, and the instruction set (by instruction types). Examples are included to show efficient ways of using this architecture and instruction set. To condense this section, detailed explanations of each instruction are included in **Appendix A. Instruction Set Details**. These explanations include detailed cycle-by-cycle bus activity and Boolean expressions for condition code bits. This section should be used to gain a general understanding of the CPU and instruction set.

6.3 Programmer's Model

Figure 6-1 shows the programmer's model of the M68HC11 CPU. The CPU registers are an integral part of the CPU and are not addressed as if they were memory locations. Each of these registers is discussed in the subsequent paragraphs.

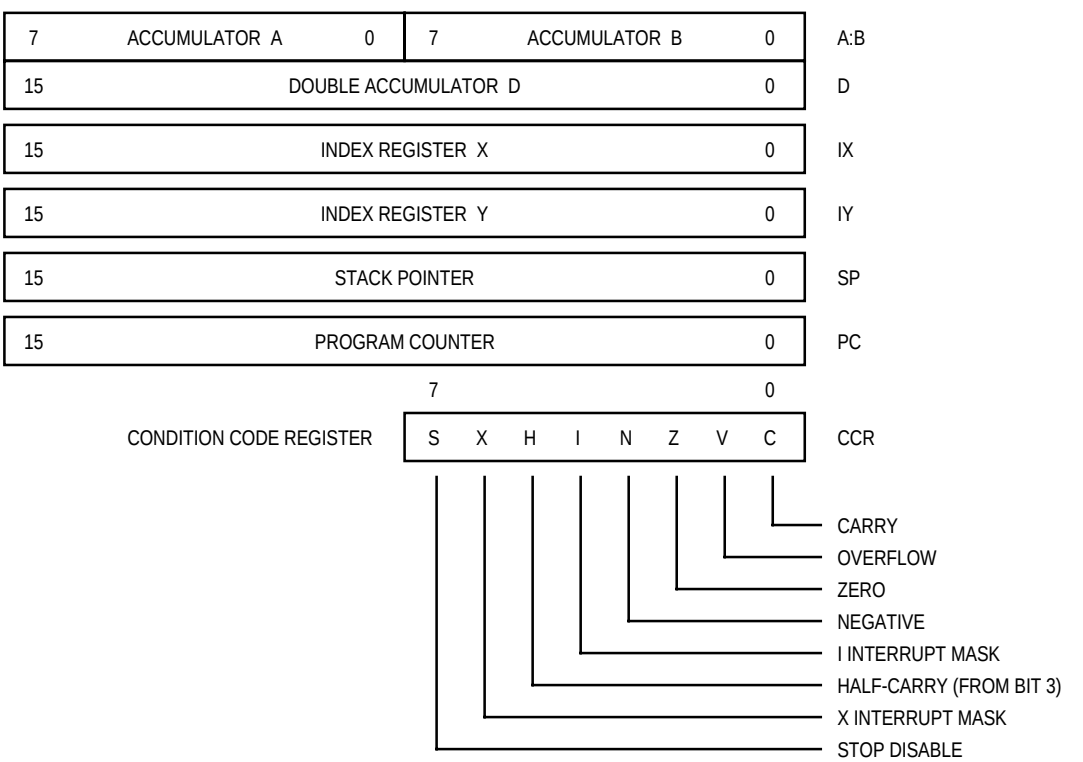

Figure 6-1. M68HC11 Programmer's Model

6.3.1 Accumulators (A, B, and D)

Accumulators A and B are general-purpose 8-bit accumulators used to hold operands and results of arithmetic calculations or data manipulations. Some instructions treat the combination of these two 8-bit accumulators as a 16-bit double accumulator (accumulator D).

Most operations can use accumulator A or B interchangeably; however, there are a few notable exceptions. The ABX and ABY instructions add the contents of the 8-bit accumulator B to the contents of the 16-bit index register X or Y, and there are no equivalent instructions that use A instead of B. The TAP and TPA instructions are used to transfer data from accumulator A to the condition code register or from the condition code register to accumulator A; however, there are no equivalent instructions that use B rather than A. The decimal adjust accumulator A (DAA) instruction is used after binary-coded decimal (BCD) arithmetic operations, and there is no equivalent BCD instruction to adjust B. Finally, the add, subtract, and compare instructions involving both A and B (ABA, SBA, and CBA) only operate in one direction; therefore, it is important to plan ahead so the correct operand will be in the correct accumulator.

6.3.2 Index Registers (X and Y)

The 16-bit index registers X and Y are used for indexed addressing mode. In the indexed addressing mode, the contents of a 16-bit index register are added to an 8-bit offset, which is included as part of the instruction, to form the effective address of the operand to be used in the instruction. In most cases, instructions involving index register Y take one extra byte of object code and one extra cycle of execution time compared to the equivalent instruction using index register X. The second index register is especially useful for moves and in cases where operands from two separate tables are involved in a calculation. In the earlier M6800 and M6801, the programmer had to store the index to some temporary location so the second index value could be loaded into the index register.

The ABX and ABY instructions along with increment and decrement instructions allow some arithmetic operations on the index registers, but, in some cases, more powerful calculations are needed. The exchange instructions, XGDX and XGDY, offer a simple way to load an index value into the 16-bit double accumulator, which has more powerful arithmetic capabilities than the index registers themselves.

It is very common to load one of the index registers with the beginning address of the internal register space (usually $1000), which allows the indexed addressing mode to be used to access any of the internal I/O and control registers. Indexed addressing requires fewer bytes of object code than the corresponding instruction using extended addressing. Perhaps a more important argument for using indexed addressing to access register space is that bit-manipulation instructions are available for indexed addressing but not for extended addressing.

6.3.3 Stack Pointer (SP)

The M68HC11 CPU automatically supports a program stack. This stack may be located anywhere in the 64-Kbyte address space and may be any size up to the amount of memory available in the system. Normally, the stack pointer register is initialized by one of the first instructions in an application program. Each time a byte is pushed onto the stack, the stack pointer is automatically decremented, and each time a byte is pulled off the stack, the stack pointer is automatically incremented. At any given time, the stack pointer register holds the 16-bit address of the next free location on the stack. The stack is used for subroutine calls, interrupts, and for temporary storage of data values.

When a subroutine is called by a jump-to-subroutine (JSR) or branch-to-subroutine (BSR) instruction, the address of the next instruction after the JSR or BSR is automatically pushed onto the stack (low half first). When the subroutine is finished, a return-from-subroutine (RTS) instruction is executed. The RTS causes the previously stacked return address to be pulled off the stack, and execution continues at this recovered return address.

Whenever an interrupt occurs (provided it is not masked), the current instruction finishes normally, the address of the next instruction (the current value in the program counter) is pushed onto the stack, all of the CPU registers are pushed onto the stack, and execution continues at the address specified by the vector for the highest priority pending interrupt. After completing the interrupt service routine, a return from interrupt (RTI) instruction is executed. The RTI instruction causes the saved registers to be pulled off the stack in reverse order, and program execution resumes as if there had been no interruption.

Another common use for the stack is for temporary storage of register values. A simple example would be a subroutine using accumulator A. The user could push accumulator A onto the stack when entering the subroutine and pull it off the stack just before leaving the subroutine. This method is a simple way to ensure a register(s) will be the same after returning from the subroutine as it was before starting the subroutine.

The most important aspect of the stack is that it is completely automatic. A programmer does not normally have to be concerned about the stack other than to be sure that it is pointing at usable random-access memory (RAM) and that there is sufficient space. To ensure sufficient space, the user would need to know the maximum depth of subroutine or interrupt nesting possible in the particular application.

There are a few less common uses for the stack. For instance, the stack can be used to pass parameters to a subprogram, which is fairly common in high-level language compilers but is often overlooked by assembly-language programmers. There are two advantages of this technique over specific assignment of temporary or variable locations. First, the memory locations are only needed for the time the subprogram is being executed; they can be used for something else when the subprogram is completed. Second, this feature makes the subprogram re-entrant so that an interrupting program could call the same subprogram with a different set of values without disturbing the interrupted use of the subprogram.

In unusual cases, a programmer may want to look at or even manipulate something that is on the stack, which should only be attempted by an experienced programmer because it requires a detailed understanding of how the stack operates. Monitor programs like BUFFALO sometimes

place items on a stack manually and then perform an RTI instruction to go to a user program. This technique is an odd use of the stack and RTI instruction because an RTI would normally correspond to a previous interrupt.

6.3.4 Program Counter (PC)

The program counter is a 16-bit register that holds the address of the next instruction to be executed.

6.3.5 Condition Code Register (CCR)

This register contains five status indicators, two interrupt masking bits, and a STOP disable bit. The register is named for the five status bits since that is the major use of the register. In the earlier M6800 and M6801 CPUs, there was no X interrupt mask and no STOP disable control in this register.

The five status flags reflect the results of arithmetic and other operations of the CPU as it performs instructions. The five flags are half carry (H), negative (N), zero (Z), overflow (V), and carry/borrow (C). The half-carry flag, which is used only for BCD arithmetic operations (see **6.5.1.2 Arithmetic Operations**), is only affected by the add accumulators A and B (ABA), ADD, and add with carry (ADC) addition instructions (21 opcodes total). The N, Z, V, and C status bits allow for branching based on the results of a previous operation. Simple branches are included for either state of any of these four bits. Both signed and unsigned versions of branches are provided for the conditions <, ≤, =, ≠, ≥, or >.

The H bit indicates a carry from bit 3 during an addition operation. This status indicator allows the CPU to adjust the result of an 8-bit BCD addition so it is in correct BCD format, even though the add was a binary operation. This H bit, which is only updated by the ABA, ADD, and ADC instructions, is used by the DAA instruction to compensate the result in accumulator A to correct BCD format.

The N bit reflects the state of the most significant bit (MSB) of a result. For twos complement, a number is negative when the MSB is set and positive when the MSB is 0. The N bit has uses other than in twos-complement operations. By assigning an often tested flag bit to the MSB of a register or memory location, the user can test this bit by loading an accumulator.

The Z bit is set when all bits of the result are 0s. Compare instructions do an internal implied subtraction, and the condition codes, including Z, reflect the results of that subtraction. A few operations (INX, DEX, INY, and DEY) affect the Z bit and no other condition flags. For these operations, the user can only determine = and ≠.

The V bit is used to indicate if a twos-complement overflow has occurred as a result of the operation.

The C bit is normally used to indicate if a carry from an addition or a borrow has occurred as a result of a subtraction. The C bit also acts as an error flag for multiply and divide operations. Shift and rotate instructions operate with and through the carry bit to facilitate multiple-word shift operations.

In the M68HC11 CPU, condition codes are automatically updated by almost all instructions; thus, it is rare to execute any extra instructions to specifically update the condition codes. For example, the load accumulator A (LDAA) and store accumulator A (STAA) instructions automatically set or clear the N, Z, and V condition code flags. (In some other architectures, few instructions affect the condition code bits; thus, it takes two instructions to load and test a variable.) The challenge in a Motorola processor lies in finding instructions that specifically do not alter the condition codes in rare cases where that is desirable. The most important instructions that do not alter condition codes are the pushes, pulls, add B to X (ABX), add B to Y (ABY), and 16-bit transfers and exchanges. It is always a good idea to refer to an instruction set summary such as the pocket guide (Motorola document order number MC68HC11A8RG/AD) to check which condition codes are affected by a particular instruction.

The STOP disable (S) bit is used to allow or disallow the STOP instruction. Some users consider the STOP instruction dangerous because it causes the oscillator to stop; however, the user can set the S bit in the CCR to disallow the STOP instruction. If the STOP instruction is encountered by the CPU while the S bit is set, it will be treated like a no-operation (NOP) instruction, and processing continues to the next instruction.

The interrupt request (IRQ) mask (I bit) is a global mask that disables all maskable interrupt sources. While the I bit is set, interrupts can become pending and are remembered, but CPU operation continues uninterrupted until the I bit is cleared. After any reset, the I bit is set by default and can be cleared only by a software instruction. When any interrupt occurs, the I bit is automatically set after the registers are stacked but before the interrupt vector is fetched. After the interrupt has been serviced, an RTI instruction is normally executed, restoring the registers to the values that were present before the interrupt occurred. Normally, the I bit would be 0 after an RTI was executed. Although interrupts can be re-enabled within an interrupt service routine, to do so is unusual because nesting of interrupts becomes possible, which requires much more programming care than single-level interrupts and seldom improves system performance.

The XIRQ mask (X bit) is used to disable interrupts from the $\overline{\text{XIRQ}}$ pin. After any reset, X is set by default and can be cleared only by a software instruction. When $\overline{\text{XIRQ}}$ is recognized, the X bit (and I bit) are automatically set after the registers are stacked but before the interrupt vector is fetched. After the interrupt has been serviced, an RTI instruction is normally executed, causing the registers to be restored to the values that were present before the interrupt occurred. It is logical to assume the X bit was clear before the interrupt; thus, the X bit would be 0 after the RTI was executed. Although $\overline{\text{XIRQ}}$ can be re-enabled within an interrupt service routine, to do so is unusual because nesting of interrupts becomes possible, which requires much more programming care than single-level interrupts.

6.5.1.1 Loads, Stores, and Transfers

Almost all MCU activities involve transferring data from memories or peripherals into the CPU or transferring results from the CPU into memory or I/O devices. The load, store, and transfer instructions associated with the accumulators are summarized in **Table 6-1**. Additional load, store, push, and pull instructions are associated with the index registers and stack pointer register (see **6.5.2 Stack and Index Register Instructions**).

Table 6-1. Load, Store, and Transfer Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Clear Memory Byte	CLR			X	X	X	
Clear Accumulator A	CLRA						X
Clear Accumulator B	CLRB						X
Load Accumulator A	LDAA	X	X	X	X	X	
Load Accumulator B	LDAB	X	X	X	X	X	
Load Double Accumulator D	LDD	X	X	X	X	X	
Pull A from Stack	PULA						X
Pull B from Stack	PULB						X
Push A onto Stack	PSHA						X
Push B onto Stack	PSHB						X
Store Accumulator A	STAA	X	X	X	X	X	
Store Accumulator B	STAB	X	X	X	X	X	
Store Double Accumulator D	STD	X	X	X	X	X	
Transfer A to B	TAB						X
Transfer A to CCR	TAP						X
Transfer B to A	TBA						X
Transfer CCR to A	TPA						X
Exchange D with X	XGDX						X
Exchange D with Y	EGDY						X

6.5.1.2 Arithmetic Operations

This group of instructions supports arithmetic operations on a variety of operands; 8- and 16-bit operations are supported directly and can easily be extended to support multiple-word operands. Twos-complement (signed) and binary (unsigned) operations are supported directly. BCD arithmetic is supported by following normal arithmetic instruction sequences, using the DAA instruction, which restores results to BCD format. Compare instructions perform a subtract within the CPU to update the condition code bits without altering either operand. Although test instructions are provided, they are seldom needed since almost all other operations automatically update the condition code bits.

Table 6-2. Arithmetic Operation Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulators	ABA						X
Add Accumulator B to X	ABX						X
Add Accumulator B to Y	ABY						X
Add with Carry to A	ADCA	X	X	X	X	X	
Add with Carry to B	ADCB	X	X	X	X	X	
Add Memory to A	ADDA	X	X	X	X	X	
Add Memory to B	ADDB	X	X	X	X	X	
Add Memory to D (16 Bit)	ADDD	X	X	X	X	X	
Compare A to B	CBA						X
Compare A to Memory	CMPA	X	X	X	X	X	
Compare B to Memory	CMPB	X	X	X	X	X	
Compare D to Memory (16 Bit)	CPD	X	X	X	X	X	
Decimal Adjust A (for BCD)	DAA						X
Decrement Memory Byte	DEC			X	X	X	
Decrement Accumulator A	DECA						X
Decrement Accumulator B	DECB						X
Increment Memory Byte	INC			X	X	X	
Increment Accumulator A	INCA						X
Increment Accumulator B	INCB						X

Table 6-2. Arithmetic Operation Instructions (Continued)

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Twos Complement Memory Byte	NEG			X	X	X	
Twos Complement Accumulator A	NEGA						X
Twos Complement Accumulator B	NEGB						X
Subtract with Carry from A	SBCA	X	X	X	X	X	
Subtract with Carry from B	SBCB	X	X	X	X	X	
Subtract Memory from A	SUBA	X	X	X	X	X	
Subtract Memory from B	SUBB	X	X	X	X	X	
Subtract Memory from D (16 Bit)	SUBD	X	X	X	X	X	
Test for Zero or Minus	TST			X	X	X	
Test for Zero or Minus A	TSTA						X
Test for Zero or Minus B	TSTB						X

6.5.1.3 Multiply and Divide

One multiply and two divide instructions are provided. The 8-bit by 8-bit multiply produces a 16-bit result. The integer divide (IDIV) performs a 16-bit by 16-bit divide, producing a 16-bit result and a 16-bit remainder. The fractional divide (FDIV) divides a 16-bit numerator by a larger 16-bit denominator, producing a 16-bit result (a binary weighted fraction between 0 and 0.99998) and a 16-bit remainder. FDIV can be used to further resolve the remainder from an IDIV or FDIV operation.

Table 6-3. Multiply and Divide Instructions

Function	Mnemonic	INH
Multiply ($A \times B \Rightarrow D$)	MUL	X
Fractional Divide ($D \div X \Rightarrow X$; $r \Rightarrow D$)	FDIV	X
Integer Divide ($D \div X \Rightarrow X$; $r \Rightarrow D$)	IDIV	X

6.5.1.4 Logical Operations

This group of instructions is used to perform the Boolean logical operations AND, inclusive OR, exclusive OR, and one's complement.

Table 6-4. Logical Operation Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
AND A with Memory	ANDA	X	X	X	X	X	
AND B with Memory	ANDB	X	X	X	X	X	
Bit(s) Test A with Memory	BITA	X	X	X	X	X	
Bit(s) Test B with Memory	BITB	X	X	X	X	X	
One's Complement Memory Byte	COM			X	X	X	
One's Complement A	COMA						X
One's Complement B	COMB						X
OR A with Memory (Exclusive)	EORA	X	X	X	X	X	
OR B with Memory (Exclusive)	EORB	X	X	X	X	X	
OR A with Memory (Inclusive)	ORAA	X	X	X	X	X	
OR B with Memory (Inclusive)	ORAB	X	X	X	X	X	

6.5.1.5 Data Testing and Bit Manipulation

This group of instructions is used to operate on operands as small as a single bit, but these instructions can also operate on any combination of bits within any 8-bit location in the 64-Kbyte memory space. The bit test (BITA or BITB) instructions perform an AND operation within the CPU to update condition code bits without altering either operand. The BSET and BCLR instructions read the operand, manipulate selected bits within the operand, and write the result back to the operand address. Some care is required when read-modify-write instructions such as BSET and BCLR are used on I/O and control register locations because the physical location read is not always the same as the location written.

Table 6-5. Data Testing and Bit Manipulation Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY
Bit(s) Test A with Memory	BITA	X	X	X	X	X
Bit(s) Test B with Memory	BITB	X	X	X	X	X
Clear Bit(s) in Memory	BCLR		X		X	X
Set Bit(s) in Memory	BSET		X		X	X
Branch if Bit(s) Clear	BRCLR		X		X	X
Branch if Bit(s) Set	BRSET		X		X	X

6.5.1.6 Shifts and Rotates

All the shift and rotate functions in the M68HC11 CPU involve the carry bit in the CCR in addition to the 8- or 16-bit operand in the instruction, which permits easy extension to multiple-word operands. Also, by setting or clearing the carry bit before a shift or rotate instruction, the programmer can easily control what will be shifted into the opened end of an operand. The arithmetic shift right (ASR) instruction maintains the original value of the MSB of the operand, which facilitates manipulation of twos-complement (signed) numbers.

Table 6-6. Shift and Rotate Instructions

Function	Mnemonic	IMM	DM	EXT	INDX	INDY	INH
Arithmetic Shift Left Memory	ASL			X	X	X	
Arithmetic Shift Left A	ASLA						X
Arithmetic Shift Left B	ASLB						X
Arithmetic Shift Left Double	ASLD						X
Arithmetic Shift Right Memory	ASR			X	X	X	
Arithmetic Shift Right A	ASRA						X
Arithmetic Shift Right B	ASRB						X
(Logical Shift Left Memory)	(LSL)			X	X	X	
(Logical Shift Left A)	(LSLA)						X
(Logical Shift Left B)	(LSLB)						X
(Logical Shift Left Double)	(LSLD)						X
Logical Shift Right Memory	LSR			X	X	X	
Logical Shift Right A	LSRA						X
Logical Shift Right B	LSRB						X
Logical Shift Right D	LSRD						X
Rotate Left Memory	ROL			X	X	X	
Rotate Left A	ROLA						X
Rotate Left B	ROLB						X
Rotate Right Memory	ROR			X	X	X	
Rotate Right A	RORA						X
Rotate Right B	RORB						X

The logical-left-shift instructions are shown in parentheses because there is no difference between an arithmetic and a logical left shift. Both mnemonics are recognized by the assembler as equivalent, but having both instruction mnemonics makes some programs easier to read.

6.5.2 Stack and Index Register Instructions

Table 6-7 summarizes the instructions available for the 16-bit index registers (X and Y) and the 16-bit stack pointer.

Table 6-7. Stack and Index Register Instructions

Function	Mnemonic	IMM	DIR	EXT	INDX	INDY	INH
Add Accumulator B to X	ABX						X
Add Accumulator B to Y	ABY						X
Compare X to Memory (16 Bit)	CPX	X	X	X	X	X	
Compare Y to Memory (16 Bit)	CPY	X	X	X	X	X	
Decrement Stack Pointer	DES						X
Decrement Index Register X	DEX						X
Decrement Index Register Y	DEY						X
Increment Stack Pointer	INS						X
Increment Index Register X	INX						X
Increment Index Register Y	INY						X
Load Index Register X	LDX	X	X	X	X	X	
Load Index Register Y	LDY	X	X	X	X	X	
Load Stack Pointer	LDS	X	X	X	X	X	
Pull X from Stack	PULX						X
Pull Y from Stack	PULY						X
Push X onto Stack	PSHX						X
Push Y onto Stack	PSHY						X
Store Index Register X	STX	X	X	X	X	X	
Store Index Register Y	STY	X	X	X	X	X	
Store Stack Pointer	STS	X	X	X	X	X	
Transfer SP to X	TSX						X
Transfer SP to Y	TSY						X
Transfer X to SP	TXS						X
Transfer Y to SP	TYS						X
Exchange D with X	XGDX						X
Exchange D with Y	XGDY						X