2 The 68008 Microprocesor

2.1 Introduction for those new to machine code

Most people learn BASIC as a first language when they start programming a computer. As a language, BASIC (and SuperBasic) is very easy to understand and fairly easy for beginners to grasp. If errors are made in the program, BASIC *understands* what is wrong and tells you about it with comments like `Not complete`.

Up to a point BASIC is sufficient for the majority of tasks, however, most programmers will reach the stage at which BASIC constrains progress. Many ideas cannot be implemented in BASIC because they require reasonable speed to operate properly. Arcade style games are an obvious example of this, but real time graphics in general will require *speed*.

Once this point is reached, *machine code* is required. This new form of programming is several orders of magnitude faster than BASIC. As we all know, it takes longer for us to read a book written in a *foreign language* than it does to read one written in our own *native language*. To the QL, BASIC is a *foreign language* and *machine code* is its *native language*.

To see how the QL can have a *native language* which is more fundamental than BASIC, you need to know something about the internal structure of the QL.

At the heart of every QL computer there is a powerful microprocessor chip called the 68008. This microprocessor is the brain of the computer and provides it with all of its computing power. In very simple terms, the 68008 gets instructions from memory (one at a time) and performs some function which is directly related to the instruction. This may involve reading in data, doing some addition, saving data in memory or some other simple task. Having executed one instruction, the 68008 gets the next instruction, and so on, in a never ending sequence (until the power is turned off!).

Although memory is addressed as bytes, each instruction is held in a word (two bytes). This word contains all the necessary information so that the 68008 can decide what to do next. For a programmer to work out the 16 bits for each word in memory would prove to be an insurmountable task if it had to be done by hand (16 bits gives some 65536 different ways of forming instruction codes). A special higher level program (usually written in 68008 code) is therefore required to convert codes which are understandable by humans (like 'ADD') into ones which can be understood by the 68008. A program which will do this is called an *assembler*.

Machine code programming with an assembler is substantially different to programming in BASIC. For a start, there are only sixteen *variables* which can be used. These are called *registers*, and are all 32-bits long. Eight of them are used to store memory addresses and are called *address registers*. The other eight are generally used to store data and are called *data registers*. Imagine how difficult it would be to write a BASIC program with only 16 variables! To overcome the limitation of storage, registers can be saved into memory and then loaded back from memory as required. Arithmetic can be peformed by the microprocessor as well, but not in floating point format like BASIC. Simple integer addition, subtraction, multiplication and division are all provided, plus logical operations like AND, OR. Instructions for jumping from one part of a machine code program to another are also available. In all, a very powerful range of instructions have been built in to the 68008, to allow very sophisticated programs to be written. BASIC is itself just another machine code program.

Machine code differs considerably from BASIC, with an important difference being error handling. If a 'save' instruction was given the wrong address in which to save data, it would probably put it straight into the middle of a program. If this modified program was ever executed, it would go wrong as well, and eventually (normally within a few microseconds) the entire QL computer would crash. The only recovery possible would then be the RESET button. All the program and data would have been lost.

You may now be thinking that there is not really much point in writing machine code programs if they're going to be so much trouble. This is not so, since apart from the tremendous speed advantage, many of the facilities provided by QDOS are only accessible directly from machine code. One of the most notable is *multitasking* which will allow many different programs to be running at the same time. There are many more which will only be fully appreciated after you have read this book!

To summarise, the 68008 microprocessor runs programs which are written in *machine code*. This runs faster than BASIC, but is more complicated to write. Only by writing programs in machine code is it possible to take full advantage of the QL.

QDOS is a collection of machine code subroutines which can be used by user programs. A wide selection of standard functions are provided by these routines, which means that programmers don't have to keep on writing the same bits of code over and over again. The sort of thing which QDOS provides for is the output of characters to the screen. If a programmer wanted to do this himself, it would be necessary to store the dot pattern for the character in memory, copy it to a place on the screen, and then run through some checks. These would involve several operations like checking whether the current character is the last one on the line (if so, the next character should go on the next line) etc. You will appreciate how tedious it would be to program like this. Luckily, QDOS has a routine which does it all for you. Simply provide the character code, and QDOS does the rest. Apart from these simple types of routines, very complex functions are also supported by QDOS. Things like saving data on the microdrives, supporting multitasking, and allocating memory, are all covered by QDOS. All of these features are explored in considerable depth in later chapters.

In order to understand how the various QDOS routines can be accessed, it is first necessary to know how the 68008 operates. The rest of this chapter sets out to do this.

2.2 The programmer's model

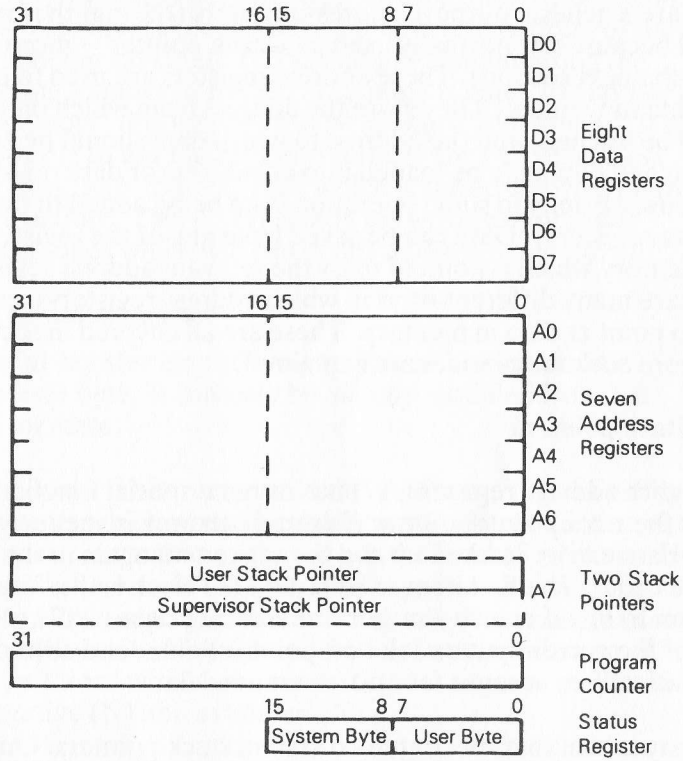


Figure 2.1 Programmer's model of the 68008

Internally, the 68008 microprocessor consists of several registers. These are illustrated in figure 2.1. The registers can be split up into five functional groups:

2.2.1 Data registers

There are 8 data registers, each one being 32-bits long (a 'long word'). Most general data storage and transfer tasks are carried out through this set of registers. Data can be loaded into them from other registers, memory or peripherals like the microdrive. Once there, the data can be manipulated in a variety of different ways, such as adding some value to it. The data can then be stored back into memory, or sent to some peripheral.

2.2.2 Address registers

There are 8 general purpose address registers (the eighth is rather special because it is normally used as a stack pointer – more on this in the next section). These address registers are used to help with data movement. They store the address from which data should be fetched, and the address to which data should be saved. The fetched data may be loaded into an address or data register, or just used briefly in some operation (like being added to one of the data registers). Data can be saved from any of the registers into memory which is pointed to by the relevant address register. There are many different ways in which address registers can be used to point at data in memory. These are all covered in section 2.5 where *addresing modes* are explained.

2.2.3 Stack pointers

The eighth address register (A7) has rather a special function. It acts as the *stack pointer*. A stack is simply an area in memory where data can be stacked up and then taken off again in the reverse order. The first item to be put on the stack is therfore the last item to be taken off – rather like stacking boxes. A7 keeps track of the current put on/take off point for data, and adjusts automatically to account for added/removed data.

You may notice that there are two system stack pointers. Only one of these is present at any given time. The other is invisible. The one which is currently selected is determined by the processor's *priviledge* status. There are two levels of priviledge. The lowest always exists when the 68008 is in *USER* mode. Most programs written by a user, will run in this state. Therefore all pushing and pulling of data to and from the stack changes data on the user stack. The highest priviledge state is the *SUPERVISOR* state. Most of QDOS runs in the supervisor state. More power can be obtained at this level. For example, it is only possible to execute the system RESET instruction from supervisor mode. These two levels of security means that a rogue program is less likely to totally disrupt the system (but it doesn't guarantee that it won't!). Any of the other seven address registers can also be used as stack pointers if desired.

2.2.4 The program counter

This 32-bit register is not directly accessible like the data and address registers. It keeps track of the current address of the program being executed. It is rather like remembering the current line number in BASIC. If a jump is made to another part of the memory, this register will be changed accordingly. It is often posible to use the program counter (or PC) in address calculations, like 'load the data which is stored 20 bytes on from this instruction'. This is very handy for writing *relocatable* programs (ones which will run at any address in memory).

2.2.5 The status register

This 16-bit register is split into two halves, the supervisor byte and the user byte. It can only be *directly* modified from the supervisor state.

Only the lower 5 bits are used in the user byte. These signify certain states which would normally occur after some sort of test or arithmetic instruction. They are at the level of 'primary school arithmetic'. If you add two numbers together, which are too large to fit into a data register, the carry flag will be set. If a number is subtracted from itself, the zero flag will be set, and so on. The other conditions which are recoded by this register are overflow (V), negative (N) and extended (X).

Only five bits are used in the supervisor byte. One bit indicates the processor state (user or supervisor). The other single bit indicates whether the trace mode is on or off (trace is used for stepping through a program one instruction at a time, when it is being tested). The remaining three bits contain the current *interrupt* priority. The priority can be set to one of eight levels. Level 0 is the lowest and level 7 is the highest. All interrupts which are higher in priority than this register's value will not be *masked*. Level 7 interrupts can never be masked.

2.2.6 Interrupts

Interrupts are generated by external pieces of hardware. On the QL, an interrupt is generated 50 times every second by the video circuitry. The arrival of this signal at the 68008 causes several things to happen. The current operation is suspended (assuming that the priority of the interrupt is high enough), and the 68008

jumps to an interrupt service routine. It should save all of the registers before entering this routine, perform whatever function is necessary to service the interrupt, restore the registers, and return. The overall effect of the interrupt is that some function has been performed, but the operation of the current program has not been seriously affected, only delayed by a few microseconds. Normally, interrupts are used to service devices which require immediate attention. If these are not looked at immediately, some data may be lost (eg. microdrive tape will have gone past the read head).

2.3 Data organisation in memory

The 68008 supports operations on four different types of data. These are bit, byte, word or long word, consisting of 1, 8, 16 or 32 bits respectively. The different ways in which this data is organised in memory can be seen in figure 2.2.

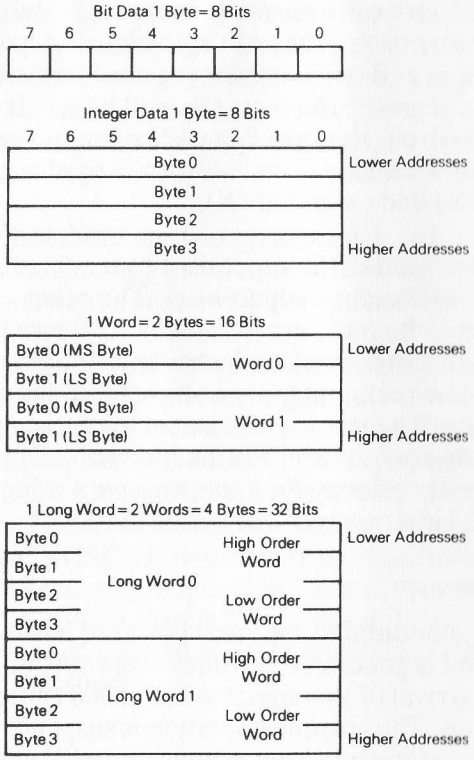


Figure 2.2 Memory data organisation

NOTE that instructions are always formed out of words and that these words must always start at an even address. Sometimes, when storing byte data, there will necessarily be one byte wasted after the data and before the following instruction.

2.4 Traps and exception processing

There are several types of *exceptions* which can occur on the 68008. These fall into the two categories of either internally or externally generated exceptions. As mentioned above, interrupts fall into the externally generated type. Internally generated types are produced by executing a software instruction which is either `TRAP`, `TRAPV`, `CHK` or `DIV`.

Vector Number(s)	Address			Assignment
	Dec	Hex	Space	
0	0	000	SP	Reset: Initial SSP
–	4	004	SP	Reset: Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12	48	030	SD	(Unassigned, Reserved)
13	52	034	SD	(Unassigned, Reserved)
14	56	038	SD	(Unassigned, Reserved)
15	60	03C	SD	Uninitialized Interrupt Vector
16-23	64	04C	SD	(Unassigned, Reserved)
	95	05F		–
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Autovector
26	104	068	SD	Level 2 Interrupt Autovector
27	108	06C	SD	Level 3 Interrupt Autovector
28	112	070	SD	Level 4 Interrupt Autovector
29	116	074	SD	Level 5 Interrupt Autovector
30	120	078	SD	Level 6 Interrupt Autovector
31	124	07C	SD	Level 7 Interrupt Autovector
32-47	128	080	SD	TRAP Instruction Vectors
	191	0BF		–
48-63	192	0C0	SD	(Unassigned, Reserved)
	255	0FF		–
64-255	256	100	SD	User Interrupt Vectors
	1023	3FF		–

Figure 2.3 Vector table

When any of these exceptions occur, a call will be made to a suitable service routine. The addresses of these service routines (one for each exception) are held in the first kilobyte of memory (see above).

The most commonly used exception features are interrupts (one every 20ms to invoke the scheduler) and TRAPs. The QL uses TRAPs to invoke operating system subroutines. The parameters to be passed to these subroutines are placed in the registers. D0 is set to the number of the required routine, and a TRAP #N is generated. The latter parts of this book explains what all of the available routines do. There are 16 TRAPs, but only five have been set aside for use by the operating system. They perform the following functions:

TRAP #0 goes from user mode into supervisor mode
TRAP #1 deals with system resource management
TRAP #2 deals with input/output allocation
TRAP #3 deals with input/output utilisation
TRAP #4 is used in conjunction with BASIC, TRAPs 2 and 3

2.5 Effective addresses

The 68008 instructions operate on data in registers or memory. It is therefore necessary to have a well defined way of addressing the data. A set of very powerful *addressing* modes have been included to suit most applications. These addresses are normally referred to as *effective addreses*. Some modes are only applicable to certain *types* of arguments. For example, it would be inappropriate to use a data register as a stack pointer. In the following mode descriptions, such *type* restrictions are mentioned as they occur. The following modes are provided:

2.5.1 REGISTER DIRECT MODES

These effective addressing modes specify that the operand (the data which is to be operated on) should be in one of the sixteen multifunction registers.

2.5.1.1 Data register direct

The operand is in the data register specified by the effective address.

2.5.1.2 Address register direct

The operand is in the address register specified by the effective address.

2.5.2 MEMORY ADDRESS MODES

These effective addressing modes specify that the operand is in memory and provide its address.

2.5.2.1 Address register indirect

The address of the operand is in the address register specified. The reference is classed as a data reference with the exception of the `JMP` and `JSR` instructions.

2.5.2.2 Address register indirect with postincrement

The address of the operand is in the specified address register (as with register indirect addressing). After the operand address is used, it is incremented by one, two, or four depending upon whether the operand size is byte, word or long word. If the address register is the stack pointer and the operand size is byte, the address is incremented by two rather than one to keep the stack pointer on a word boundary. The reference is classified as a data reference.

2.5.2.3 Address register indirect with predecrement

The address of the operand is in the specified address register (as with register indirect addressing). After the operand address is used, it is decremented by one, two, or four depending upon whether the operand size is byte, word or long word. If the address register is the stack pointer and the operand size is byte, the address is decremented by two rather than one to keep the stack pointer on a word boundary. The reference is classified as a data reference.

2.5.2.4 Address register indirect with displacement

This addressing mode only requires one word of extension. The address of the operand is derived by adding the sign-extended 16-bit displacement integer from the extension word to the address in the addres register. The reference is calssified as a data reference with the exception of the JMP and JSR instructions.

2.5.2.5 Address register indirect with index

This addressing mode requires one word of extension. The address of the operand is the sum of the address in the address register plus the sign-extended displacement integer in the low order 8 bits of the extension word, plus the contents of the index register. The reference is calssified as a data reference with the exception of the JMP and JSR instructions.

2.5.3 SPECIAL ADDRESSING MODES

The special addressing modes use the effective address to specify the special addressing mode instead of the register number.

2.5.3.1 Absolute short address

This addessing mode requires one word of extension. The address of the operand is the extension word. The 16-bit address is sign extended before it is used. The reference is calssified as a data reference with the exception of the JMP and JSR instructions.

2.5.3.2 Absolute long address

This addressing mode requires two words of extension. The address of the operand is entirely contained in the extension words, high order word first. The reference is calssified as a data reference with the exception of the JMP and JSR instructions.

2.5.3.3 Program counter with displacement

This addresing mode requires one word of extension. The address of the operand is obtained by adding the contents of this sign-extended 16-bit displacement word to the value in the program counter. The reference is calssified as a program reference.

2.5.3.4 Program conter with index

One extension word is required for this addressing mode. The operand address is obtained by adding the sign-extended displacement in the lower eight bits of the extension word and the contents of the index register to the program counter. The value in the program counter is the address of the extension word. This reference is classed as a program reference.

2.5.3.5 Immediate data

Either one or two bytes of extension are required, depending upon the size of the operation. Three alternatives are possible:

byte operation – operand in lower byte of extension word

word operation – operand is extension word

long word – operand is in the two following extension words

2.5.3.6 Implicit reference

Some instructions make implicit reference to the program counter (PC), the system stack pointer (SP), the supervisor stack pointer (SSP), the user stack pointer (USP), or the status register (SR). A selected set of instructions can refer to the status register by means of the effective address, these are generally privileged instructions.

2.6 Instruction set summary

This section contains an overview of the 68008 instruction set. The instructions can be grouped under eight major headings:

1. Data Movement
2. Integer Arithmetic
3. Logical operations
4. Shift and Rotate
5. Bit Manipulation
6. Binary Coded Decimal
7. Program control
8. System control

These are now explained in more detail. Note that the following codes are used to indicate particular types of operands:

An – address register
Dn – data register
Rn – any data or address register
PC – program counter
SR – status register
CCR – condition codes (low byte of status register)
SSP – supervisor stack pointer
USP – user stack pointer
SP – active stack pointer (equivalent to A7)
X – extend operand (from condition codes)
Z – zero condition code
V – overflow condition code
Immediate data – immediate data from the instruction
d – address displacement
Source – source effective address
Destination – destination effective address
Vector – location of exception vector

2.6.1 Data movement operations

The `MOVE` instruction allows data to be transferred or stored. Data can be moved in byte, word, or long word units, and transferred from memory to memory, memory to register, register to memory, or register to register. Operations on address registers only allow word and long word operand transfers. In addition to the general move instruction, there are several special data move instructions. `MOVEM` moves multiple registers, `MOVEP` moves peripheral data, `EXG` exchanges register contents, `LEA` loads an effective address, `PEA` pushes an effective address onto the stack, `LINK` links the stack, `UNLK` unlinks the stack, and `MOVEQ` moves a byte of data to the specified address. The data movement operations are illustrated in figure 2.4.

Instruction	Operand Size	Operation
EXG	32	Rx ↔ Ry
LEA	32	EA → An
LINK	–	AN → – (SP) SP → An SP + displacement → SP
MOVE	8, 26, 32	(EA)s → (EA)d
MOVEM	16, 32	(EA) → An, Dn An, Dn → (EA)
MOVEP	16, 32	(EA) → Dn Dn → (EA)
MOVEQ	8	#xxx → Dn
PEA	32	EA → – (SP)
SWAP	32	Dn[31:16] ↔ Dn[15:0]
UNLK	–	An → SP (SP) + → An

NOTES:
s = source
d = destination
[ ] = bit number
– = indirect with predecrement
+ = indirect with postdecrement
# = immediate data

Figure 2.4 – Data movement operations

2.6.2 Integer arithmetic operations

The four basic arithmetic operations of `ADD` (addition), `SUB` (subtraction), `MUL` (multiply) and `DIV` (division) are supported on the 68008. There is also an arithmetic compare `CMP`, clear `CLR`, and negate `NEG`. Add and subtract are available for both data and address operations (but must be either word or long word in the case of addresses).

Signed and unsigned operands can be multiplied and divided. Words are multiplied together to produce long word results. Long words are divided by words to produce word results consisting of a word for the quotient and a word for the remainder.

A series of extended instructions are provided to allow multiprecision and mixed size arithmetic to be accomplished. These instructions are add with extend `ADDX`, subtract with extend `SUBX`, sign extend `EXT`, and negate binary with extend `NEGX`.

The operand can be compared with zero using the `TST` (test) instruction. This causes the condition codes to be set as a result of the operation. The integer arithmetic operations are summarised in figure 2.5.

Instruction	Operand Size	Operation
ADD	8, 16, 32	Dn + (EA) → Dn (EA) + Dn → (EA) (EA) + #xxx → (EA)
	16, 32	An + (EA) → An
ADDX	8, 16, 32	Dx + Dy + X → Dx
	16, 32	-(Ax) + -(Ay) + X → (Ax)
CLR	8, 16, 32	0 → (EA)
CMP	8, 16, 32	Dn - (EA) (EA) - #xxxx (Ax)+ - (Ay)+
	16, 32	An - (EA)
DIVS	32 ÷ 16	Dn ÷ (EA) → Dn
DIVU	32 ÷ 16	Dn ÷ (EA) → Dn

Instruction	Operand Size	Operation
EXT	8 → 16 16 → 32	$(Dn)_8 \rightarrow Dn_{16}$ $(Dn)_{16} \rightarrow Dn_{32}$
MULS	16 × 16 → 32	Dn × (EA) → Dn
MULU	16 × 16 → 32	Dn × (EA) → Dn
NEG	8, 16, 32	0 - (EA) → (EA)
NEGX	8, 16, 32	0 - (EA) - X → (EA)
SUB	8, 16, 32	Dn - (EA) → Dn (EA) - Dn → (EA) (EA) - #xxx → (EA) An - (EA) → An
SUBX	8, 16, 32	Dx - Dy - X → Dx -(Ax) - -(Ay) - X → (Ax)
TAS	8	(EA) - 0, 1 → EA [7]
TST	8, 16, 32	(EA) - 0

NOTES:
[ ] = bit number
= immediate data
- = indirect with predecrement
+ = indirect with postdecrement

Figure 2.5 – Integer arithmetic operations

2.6.3 Logical operations

Integer data operands of byte, word, or long word length can be manipulated using the standard logical operations of **AND**, **OR**, **EOR** and **NOT**. There is also an equivalent set of immediate instructions **ANDI**, **ORI**, and **EORI** which allow these logical operations to be provided with all sizes of immediate data. The logical operations are summarised in figure 2.6.

Instruction	Operand Size	Operation
AND	8, 16, 32	Dn Λ (EA) → Dn (EA) Λ Dn → (EA) (EA) Λ #xxx → (EA)
OR	8, 16, 32	Dn V (EA) → Dn (EA) V Dn → (EA) (EA) V #xxx → (EA)
EOR	8, 16, 32	(EA) ⊕ Dy → (EA) (EA) ⊕ #xxx → (EA)
NOT	8, 16, 32	~(EA) → EA

NOTES:
= immediate data
~ = invert
Λ = logical AND
V = logical OR
⊕ = logical exclusive OR

Figure 2.6 – Logical operations

2.6.4 Shift and rotate operations

This selection of operations allows bits to be shifted or rotated around a byte, word, or long word (for registers – only words can be used with memory).

The shift count can be specified in a data register if the shift or rotate operation is being performed on a register. Memory operations only support single bit shifts or rotates.

There are two types of shift. The arithmetic shift instructions are **ASR** and **ASL**. The logical instructions are **LSR** and **LSL**, for shifting left and right respectively.

Rotates can be performed either with or without the extend bit in the loop. The instructions are **ROXR**, **ROXL**, **ROR** and **ROL**. See figure 2.7 to see how they work.

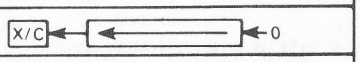
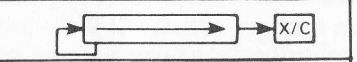
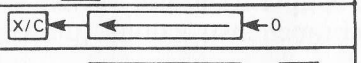
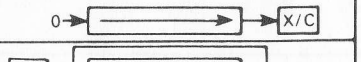
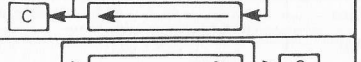
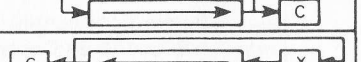
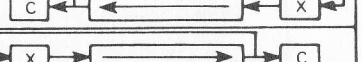
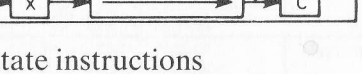
Instruction	Operand Size	Operation
ASL	8, 16, 32	
ASR	8, 16, 32	
LSL	8, 16, 32	
LSR	8, 16, 32	
ROL	8, 16, 32	
ROR	8, 16, 32	
ROXL	8, 16, 32	
ROXR	8, 16, 32	

Figure 2.7 – Shift and rotate instructions

2.6.5 Bit manipulation operations

There are four bit manipulation instructions; bit test **BTST**, bit test and set **BSET**, bit test and clear **BCLR**, and bit test and change **BCHG**. These are illustrated in figure 2.8. Note that the 'Z' refers to the 'zero' bit, which is bit 2 of the status register.

Instruction	Operand Size	Operation
BTST	8, 32	~bit of (EA) → Z
BSET	8, 32	~bit of (EA) → Z 1 → bit of EA
BCLR	8, 32	~bit of (EA) → Z 0 → bit of EA
BCHG	8, 32	~bit of (EA) → Z ~bit of (EA) → bit of EA

NOTE: ~ = invert

Figure 2.8 – Bit manipulation operations

2.6.6 Binary coded decimal operations

There are three multiprecision operations which can be performed on binary coded decimal numbers. They are; add decimal with extend **ABCD**, subtract decimal with extend **SBCD**, and negate decimal with extend **NBCD**. These are illustrated in figure 2.9.

Instruction	Operand Size	Operation
ABCD	8	$Dx_{10} + Dy_{10} + X \rightarrow Dx$ $-(Ax)_{10} + -(Ay)_{10} + X \rightarrow (Ax)$
SBCD	8	$Dx_{10} - Dy_{10} - X \rightarrow Dx$ $-(Ax)_{10} - -(Ay)_{10} - X \rightarrow (Ax)$
NBCD	8	$0 - (EA)_{10} - X \rightarrow (EA)$

Figure 2.9 – BCD operations

2.6.7 Program control operations

Program control operations use a series of conditional and unconditional branch instructions, jump instructions and return instructions.

Instruction	Operation
Conditional	
B_{CC}	Branch Conditionally (14 conditions) 8- and 16-Bit Displacement
DB_{CC}	Test Condition, Decrement, and Branch 16-Bit Displacement
S_{CC}	Set Byte Conditionally (16 Conditions)
Unconditional	
BRA	Branch Always 8- and 16-Bit Displacement
BSR	Branch to Subroutine 8- and 16-Bit Displacement
JMP	Jump
JSR	Jump to Subroutine
Returns	
RTR	Return and Restore Condition Codes
RTS	Return from Subroutine

Figure 2.10 – Program control operations

The conditional instructions can be told to set or branch on any of the following conditions:

CC(HS)	carry clear	LS	low or same
CS(LO)	carry set	LT	less than
EQ	equal	MI	minus
F	never true	NE	not equal
GE	greater or equal	PL	plus
GT	greater than	T	always true
HI	high	VC	no overflow
LE	less or equal	VS	overflow

2.6.8 System control operations

Privileged instructions, trap generating instructions and other instructions which use or modify the status register are all classed under system control. The various options available are summarised in figure 2.11.

Instruction	Operation
Privileged	
ANDI to SR	Logical AND to Status Register
EORI to SR	Logical EOR to Status Register
MOVE EA to SR	Load New Status Register
MOVE USP	Move User Stack Pointer
ORI to SR	Logical OR to Status Register
RESET	Reset External Devices
RTE	Return from Exception
STOP	Stop Program Execution
Trap Generating	
CHK	Check Data Register Against Upper Bounds
TRAP	Trap
TRAPV	Trap on Overflow
Status Register	
ANDI to CCR	Logical AND to Condition Codes
EORI to CCR	Logical EOR to Condition Codes
MOVE EA to CCR	Load New Condition Codes
MOVE SR to EA	Store Status Register
ORI to CCR	Logical OR to Condition Codes

Figure 2.11 – System control operations