INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

H4.SMR. 403/4

FIFTH COLLEGE ON MICROPROCESSORS: TECHNOLOGY AND APPLICATIONS IN PHYSICS

2- 27 October 1989

Characteristics of the M 6809

A. MARCHIORO
W. VON RUDEN
CERN, EF Division, Geneva, Switzerland

These notes are intended for internal distribution only.

Overview of the lectures

1. M6809 Basics
2. Instruction Set 1
3. Instruction Set 2
4. M6809 Hardware
5. Connecting Memory and I/O
6. Advanced Addressing Modes
7. Interrupts

Lecture 1: M6809 Basics

- Introduction to the M6800 family
- Main Characteristics of the M6809
- M6809 Programming Model
- Programs and Data in Memory
- Simple Instructions and Addressing Modes

The M6800 Family of Microprocessors

- MC 6800, MC6802, MC6808

  First generation of simple 8 bit machines

- MC6801, MC6803, MC6804, MC68701, MC68HC11

  Single chip microcomputer for high system integration

- MC6809, MC6809E

  8 bit µP (16 bit internal) with advanced instruction set

  powerful addressing modes and high level language support through

  • good stack organisation

  • position independent code

  • multitask and multiprocess organisation

- MC6805, MC68705, MC146805...

  CMOS processors with special built in functions (ADC, PLL)

  for industrial control

M6809 Characteristics: M6809 Basics

Why did we choose the MC6809 ?

- Very elegant and simple 8 bit machine with 16 bit registers
- Simple hardware architecture
- Powerful addressing modes
- Orthogonal instruction set
- Position independent code
- Good stack architecture for high level languages (PASCAL etc.)
- Memory mapped I/O
- Wide range of similar (compatible) machines and support chips available in the MC6800 family
- Very good support by Motorola, in particular for our Colleges

Main Characteristics of the M6809

Hardware:

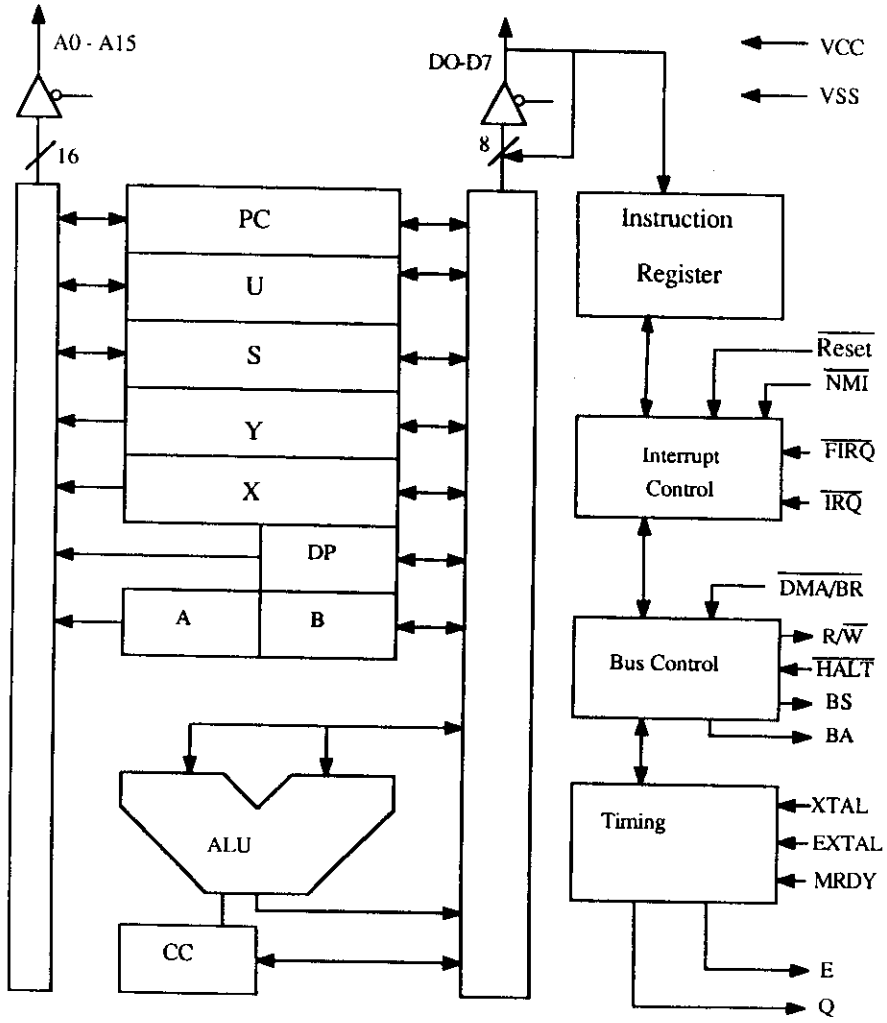
- 4 - 6 - 8 MHz clock, 1 - 0.66 - 0.5 µs machine cycle
- M6809E version with external clocks
- 16 address lines and 8 data lines
- 3 interrupt lines
- Control logic for Slow Memories, DMA
- External Event Synchronisation
- Single 5-Volt supply
- Compatible with all M6800 family peripherals

Software:

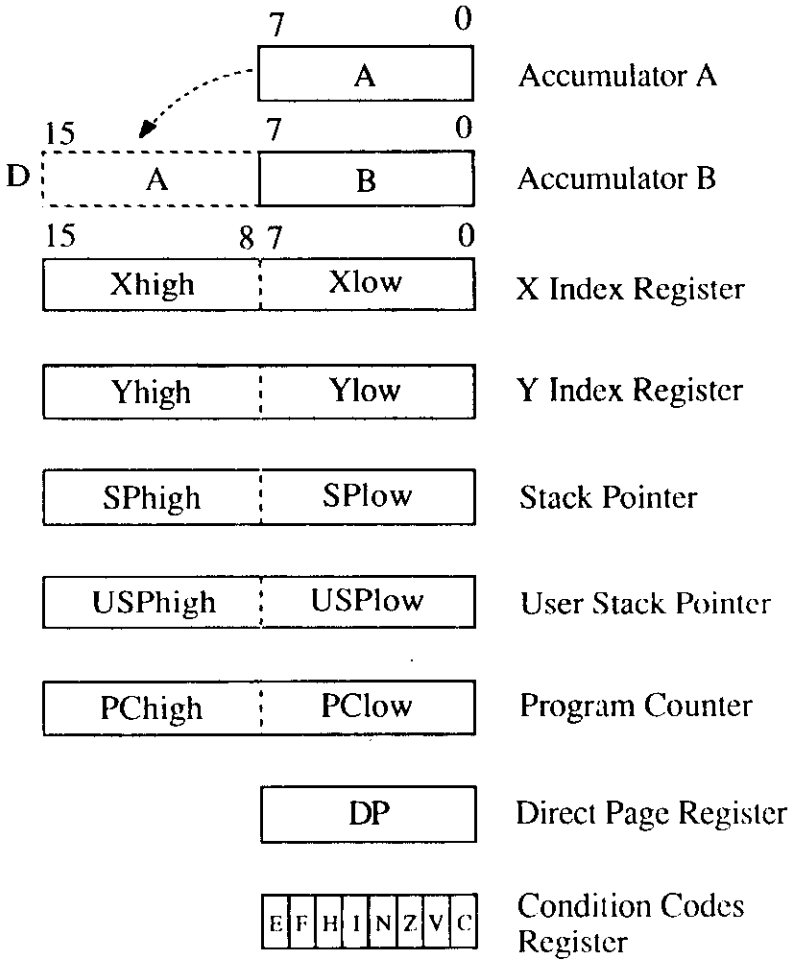
- Many Addressing modes
- 1464 Instructions
- 16-bit arithmetic, Multiply
- Effective Address Calculation
- Save/Restore any Set of Registers
- Efficient Stack Manipulation

M6809 Characteristics: M6809 Basics

M6809 Block Diagram



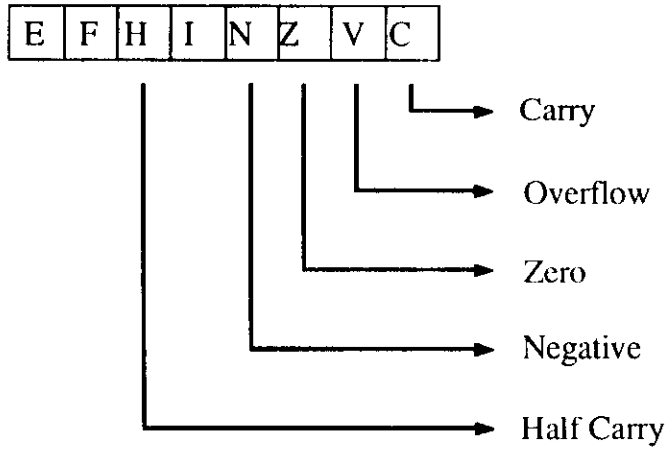
M6809 Programming Model



M6809 Characteristics: M6809 Basics

The Condition Code Register

The Condition Code Register (CC) has two main functions: it contains the result of the last arithmetic operation in H, N, Z, V, C and it controls the interrupt operations with E, F and I. The first function is needed to control the flow of the program by testing such things as "result was zero" or an "overflow" occured. These bits are explained here, whereas the interrupt related bits will be treated in lecture 7.



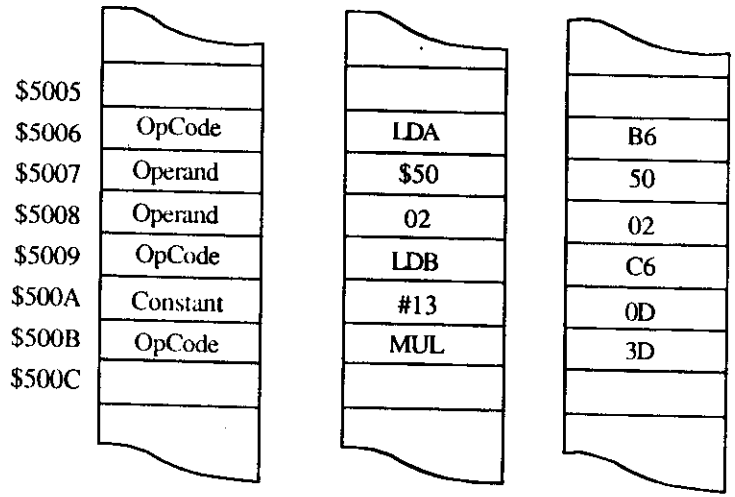
Programs and Data in memory

The M6809 is an **accumulator** based machine, i.e. all data movements and all arithmetic operations go via the accumulators. Before discussing the memory usage by program and data, we have a look at the instruction format. The M6809 has a large instruction set with many addressing modes and therefore the length of an instruction varies from one to five bytes. The general form is

Operation **Operand**

which both can have various lengths. When looking at a program in memory, one cannot tell what the meaning of the bytes is, because instructions contain a mixture of Opcodes and Operand addresses or even constants.

The organization of programs and data in memory is as follows:



Data are stored in memory as bytes; multiple bytes are used to represent 16-bit integers, 32-bit floating point number, strings, arrays, etc. The convention for the M6809 is that the most significant byte is always stored first, for instance the value $1234 is written in memory as follows:

low address..... .... | 12 | 34 | ....high address

This is very convenient because just by opening the memory one can read a value in the correct sequence. (Other machines, like IBM-PC or VAX, have the opposite convention, the lower address contains the LSB of a word).

In the same way also 32-bit words are written in the correct order, for example the value $12345678 is written as:

.... | 12 | 34 | 56 | 78 | ....

Strings are stored from the left to the right in ascending memory addresses. The text "TRIESTE" expressed in ASCII code would be found as

.... | 54| 52 | 49 | 45 | 53 | 54 | 45 | ....

When data are pushed onto the stack, they are stored in the opposite direction, because the stack addressing is backwards. Of course, the order in memory in again the same.

In summary :

We distinguish **three** main areas in memory: program, data and stack. Programs including their constants should never change, so they might be stored in Eprom. Data are variables needed by the programs or information for I/O devices. The stack area is used by the system and the user program for temporary data store.

M6809 Characteristics: M6809 Basics

M6809 Addressing Modes

- Inherent
- Immediate
- Direct
- Extended
- Indexed
- Relative
- Extended Indirect
- Indexed Indirect
- Long Relative

Simple addressing modes and first instructions

1. Inherent

All information needed to execute the instruction is contained in the instruction itself. In the case of the M6809, also the accumulator address information is part of the instruction. Examples:

Mnemonic	Code(hex)
CLRA	4F
CLRB	5F
INCB	5C
NOP	12

2. Immediate

The data needed in addition follows **immediatly** after the opcode. Only constants (data known at program load time) should appear here. The '#' sign is used to indicate this mode. Examples:

Mnemonic	Code
LDA #$25	8625
LDX #300	8E012C
LDY #65535	108EFFFF

3. Extended

Here the opcode is followed by the 16-bit address of the operand, which allows to access data anywhere in the address space of the CPU (0000-FFFF). Examples:

Mnemonic	Code
LDA $2001	B62001
ADDA $2002	BB2002
STA $2003	B72003

This little program will add the numbers in location $2001 and $2002 and store the result in location $2003.

4. Relative

Relative addressing refers to the program counter and is used to make programs position independent. In the M6809 most 'flow of control' instructions use relative addressing (branching). Branch instructions test a condition such as equal, negative, etc. and take a branch, if and only if the condition is true. The branch destination is the present program counter plus a positive or negative offset allowing to transfer control forward or backward in the program. This offset is expressed in two's compliment notation and can be 8 or 16 bits wide.

Branch instructions are shorter and execute faster than Jump instructions and produce position independent code!

Example 'Delay loop'

This is an example for a delay loop in software, which may be needed in simple systems, in particular to save hardware such as monostables, if the CPU time is not needed to do more useful things.

Addr	Code	Label	Instr	Operand	Comment
0200	86 6F		LDA	#111	Set loop counter
0202	12	WAIT	NOP		waist CPU time
0203	12		NOP		
0204	4A		DECA		end of loop?
0205	26 ??		BNE	WAIT	
0207			next instruction		

How to calculate the branch offset ?

As long as ACCA is non-zero, the program should go to 'WAIT', else the next instruction is executed. The offset is the destination address minus the present address, in our case $0202 – $0207, which is $FFFB. For short branch instructions, only the lower byte is used (8-bit offset).

Note: The 'present address' is always the address of the 'next instruction', because the program counter will already be there after reading the instruction and before actually executing it.

Lecture 2: M6809 Instruction Set

- Instruction Set Overview
- Arithmetic and Data Move Instructions
- Logic Instruction
- Pointer Register Instructions
- Branch Instructions
- Instruction Usage

Instruction Set Overview

The instruction set of the M6809 contains 1464 instructions according to Motorola. Such a CPU falls in the category of Complex Instruction Set Computers (CISC) as compared to Reduced Instruction Set Computers (RISC).

CISC type machines employ a large variety of instructions combined with complicated addressing modes implying variable length instructions and variable execution times. CISC machines support high-level languages such as Pascal, which make frequent references to the stack. Modern 16 or 32 bit CPUs employ microcode for instruction decoding and execution, which makes them rather flexible, in particular if there is "writeable control store". But often a large fraction of microstore is needed to implement a few complicated and rarely used instructions.

RISC machines use very simple instructions, usually some 30 - 50, which typically have a fixed length and limited addressing capabilities. The advantage is that instruction decoding becomes simple and that pipelining gets efficient. Such CPUs execute an instruction at every machine cycle, as long as the pipeline is not broken, i.e. no branch instructions. Of course, there are many more instructions needed to execute a given task.

There is no easy way to say, which architecture is preferable, it often depends on the application. In this lecture we will present a M6809 instructions and explain them with simple examples.

Arithmetic and Data Move Instructions

These instructions cover the needs for calculations, such as additions, subtractions, multiplication and decimal arithmetic. The M6809 has no divide instruction or any floating point support, which will have to be treated in software. Most of these instructions support the four major addressing modes (immediate, direct, indexed, extended). All operations are executed via the two accumulators A and B and operations consists of sequences like

load calculate calculate ... store

There is a set of instructions called 'data handling' instructions by Motorola, which could be called arithmetic instructions as well. They consist of shift and rotate operations used frequently in calculations. Note that the multiply instruction works only with unsigned 8 bit integers and shifts and rotates are needed for signed multiplications.

The following example shows how to add to 16-bit numbers N1 and N2 and to store the result in SUM.

The data are stored in memory as follows:

$100	N1 high
$101	N1 low
$102	N2 high
$103	N2 low
$1A0	SUM high
$1A1	SUM low

The program to add these to numbers is:

```
LDA     $101    add LSB of N1 and N2
ADDA    $103
STA     $1A1    save intermediate result
LDA     $100    add MSB's and Carry from last add
ADCA    $102
STA     $1A0    store high byte
```

A second method uses the double accumulator D, which is the combination of Acc A and Acc B, A having the MSB and B the LSB.

```
LDD     $100
ADDD    $102
STD     $1A0
```


Logic Instructions

Logic instructions operate on individual bits rather than on bytes. They differ from arithmetic instructions by not using any interdigit carry. AND, OR, EOR and COM correspond directly to their hardware equivalent but they always work on 8 bits at a time. The instruction 'ANDA #$0F' for example will AND each bit of AccA with the corresponding bit of the data byte $0F and store the result in AccA. In our example bits 0-3 of AccA will remain unchanged, bits 4-7 are forced to zero. Such operations are often called 'masking'. The following rules can be used to employ logic instructions:

AND	turn one or several bits off
OR	turn one or several bits on
EOR	invert one or several bits
COM	invert a complete byte

Logic instruction are very useful to program I/O ports and interface circuits as we will see in lecture 5. The M6809 instruction set does not include direct bit set, bit clea or bit test-and-set instructions. The latter are needed to program semaphores in operating systems or multi-tasking applications. On the M6809 semaphores can be programmed using logical shift instructions.

Pointer Register Instructions

One instruction called LOAD EFFECTIVE ADDRESS exists in the M6809 used for pointer or 'effective address' calculations. Normally the CPU calculates the addresses needed to reference data in memory, but there are cases, where one is not interested to actually fetch the data, but only to calculate the address where the next data should be fetched from.

As an example, assume that several messages are stored in form of text strings in memory and the messages are numbered from 1 to n. Let us also assume that there is a table with the start address of each message. To send a given message to a terminal a program would extract the message address by calculating the pointer to the corresponding entry in the address table and pass this information to a print routine or to the operation system.

The LEA instruction exists for the X, Y, U and S registers and uses only indexed addressing modes.

Branch Instructions

- Unconditional

BRA	branch always
BRN	branch never

- Conditional with one condition code

BNE	Z=0	branch on not equal
BEQ	Z=1	branch on equal
BPL	N=0	branch on plus
BMI	N=1	branch on minus
BCC	C=0	branch on carry clear
BCS	C=1	branch on carry set
BVC	V=0	branch on overflow clear
BVS	V=1	branch on overflow set

- Arithmetic branches (after compare), here we use CMPA [M]

Condition	Unsigned	Test	Signed	Test
A = M	BEQ	Z = 1	BEQ	Z = 1
A ≠ M	BNE	Z = 0	BNE	Z = 0
A > M	BHI	C + Z = 0	BGT	Z + (N⊕V) = 0
A ≤ M	BLS	C + Z = 1	BLE	Z + (N⊕V) = 1
A ≥ M	BHS	C = 0	BGE	N⊕V = 0
A < M	BLO	C = 1	BLT	N⊕V = 1

Instruction Usage

To determine the importance of the different instructions, one may have a look at the frequency of usage. Motorola has published a static analysis of instruction usage based on about 25'000 lines of M6800 assembly language code. The result is

Instruction type	%
Load	23.4 %
Store	15.3 %
Subroutine Calls	13.0 %
Conditional Branches	11.0 %
Unconditional Branches	6.5 %
Compare, Test	6.2 %
Increment, Decrement	6.1 %
Clear	4.4%
Add, Subtract	2.8%
Others	11.3%

Use of the offset in indexed addressing

Offset	%
zero	40.0
5 bits	53.0
6 bits	1.0
8 bits	6.0

Lecture 3: M6809 Addressing Modes

- Direct Addressing
- Long Relative Addressing
- Extended Indirect Addressing
- Indexed Addressing

More on Addressing Modes

1. Direct addressing

Both *extended* and *direct* addressing are flavours of absolute addressing. Extended addressing uses 16 bit addresses to refer to any location within the 64K address space. Direct addressing uses only an 8 bit address, the LSB of the memory address, the MSB being taken from a register, the Direct Page Register (DP). After reset, the DP is cleared to make the M6809 compatible with the M6800, which does not have a DP and where direct addressing refers always to the range $0000 - $00FF.

This addressing mode is useful in cases where optimum execution times are required. Having only one byte for the operand address means one less memory reference. Of course, the DP has to be loaded before, and that can be a headache if different values are used in different parts of the program. There will be always a case where one forgets to load it. We recommend to use it only where really needed or to restrict it to one fixed value.

2. Long Relative Addressing

This mode is identical to the relative addressing mode, except that the offset is 16 bit and therefore any location in the 64K addressing space can be reached. Note that the address calculation is done in two's compliment (signed integers) and that there is no overflow checking. Therefore, addresses which exceed $FFFF will wrap around and start at $0000 again. Use this mode rather then absolute jumps.

M6809 Characteristics: Addressing Modes

3. Extended Indirect Addressing

Indirect addressing refers to operations where the operand contains the address of the data rather than the data itself. This addressing mode saves address calculations in table operations or during parameter passing in high-level languages. The CPU reads the instruction, finds that the addressing mode is extended and will therefore read two more bytes from memory. Indirect implies that this address is used to read two more bytes, the address of the operand, and only then the operation is executed. The instruction LDD [$3200] will refer 8 times to memory; the instruction itself has two bytes, the address $3200 needs to be fetched (2 bytes), then the contents of $3200 and $3201 is read to get the operand address and finally the two data bytes to be loaded in AccD are loaded.

4. Indexed Addressing

Indexed addressing is the most complex mode in the M6809 and therefore we only treat the simple modes here. The more complex ones will be delt with in lecture 6. We distinguish four main groups:

- constant offset from register 35,Y
- accumulator offset from register B,X
- auto increment / decrement register 0,S++
- constant offset from program counter (PC) label,PCR

Note: All these modes exist also as "indirect"

LDX [ARRAY,Y]

M6809 Characteristics: Addressing Modes

Register usage:

1. There are four 16-bit registers in the M6809 which may be used as INDEX register : X, Y, S, U.
2. All offset calculations are done in two's compliment, also for the LEAX type instructions.

Examples for the use of indexed addressing:

- 16-bit counters (delays)
- search operations (string search)
- address calculations (computed goto, menu's)
- parameter passing through registers (monitor calls)
- accessing parameters on the stack
- array manipulations
- data structures

Instruction format:

(pre-byte) opcode post-byte (offset)

Lecture 4: M6809 Hardware

- Synchronous and Asynchronous Machines
- M6809 CPU signals
- M6809E specials
- M6809 and slow devices
- CPU state definitions
- Instruction Timing
- Cycle by Cycle Operation

Synchronous and Asynchronous Machines

Computers are usually based on multiple functional units connected via buses. We find two fundamental types of units called **masters** and **slaves**. Master units such as the CPU or a DMA controller control the bus and determine the transactions to be performed. Slave units respond to the requests of masters by consuming or producing data.

How are such connections established? How does one ensure that the data transfers take place correctly? A well defined protocol is the answer. One finds two basic types of protocols for parallel buses, synchronous and asynchronous.

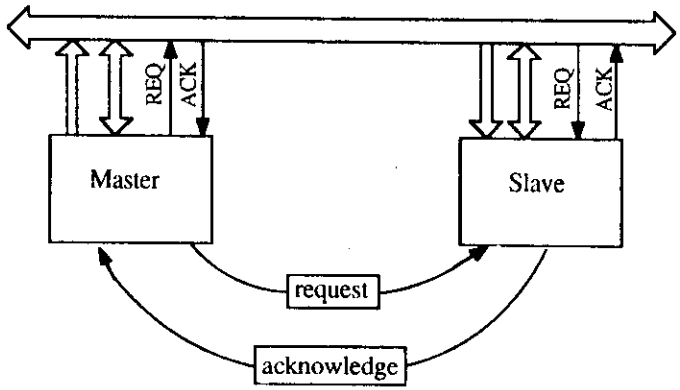
1. Synchronous protocol

In synchronous system the data transfer between a master and a slave (read or write) is simply started from the master, who assumes to get a reply from the slave within a predefined time. In simple systems such as on the M6809 there is not even any confirmation expected from the slave and therefore one may well address non-existing addresses and get in return the data from a floating bus. Another implication is that the slowest unit defines the overall system speed.

Synchronous systems are usually simpler than asynchronous ones, but they are less safe.

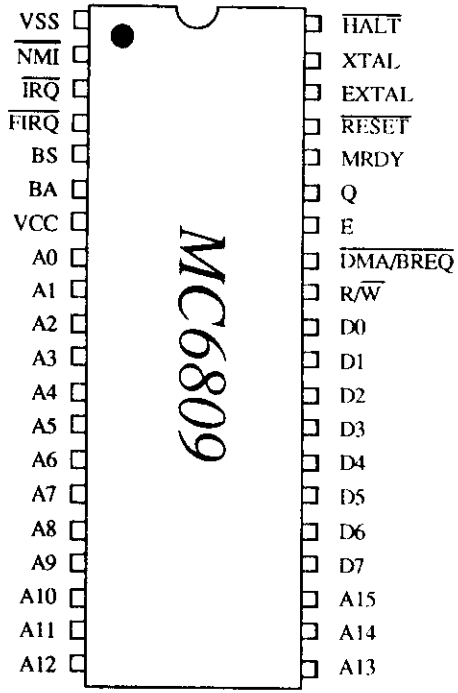
2. Asynchronous protocol

In asynchronous systems the bus activities are also started by a master, but here the master waits until it gets an active response (acknowledge) from the slave. Such answers may be accompanied by extra information about the success or failure of the transaction. In this way, the master adopts automatically to the speed of the slave and there is no problem in mixing slow and fast devices. If a slave does not respond for whatever reason, the master will wait forever and the system hangs, unless a bus timeout has been foreseen.



M6809 Characteristics: M6809 Hardware

M6809 CPU signals



M6809E special signals

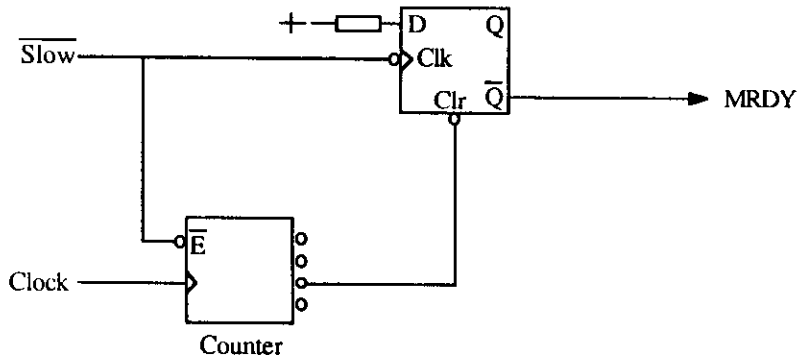
The M6809E has been conceived for systems with external clock generation and allows to build systems with two processors running with phase-shifted clocks. The main differences to the M6809 are

E and Q	are input signals
TSC	(Three State Control) available for multiprocessor applications
LIC	(Last Instruction Cycle) indicates that instruction fetch follows
AVMA	(Advanced Valid Memory Address) says that CPU will take next bus cycle
BUSY	indicates during complex instructions such as Read-Modify-Write that the bus is needed by the CPU. Also used in multi CPU systems.

M6809 and slow devices

For devices with long access times the M6809 provides the MRDY line. When MRDY is high, E and Q run normally and when MRDY goes low, E and Q may be stretched in integral multiples of quarter bus cycles. This stretching takes only place during valid memory address cycles and MRDY is ignored during internal CPU activity to avoid slowing down the CPU unnecessarily.

A possible solution is shown in the following circuit:



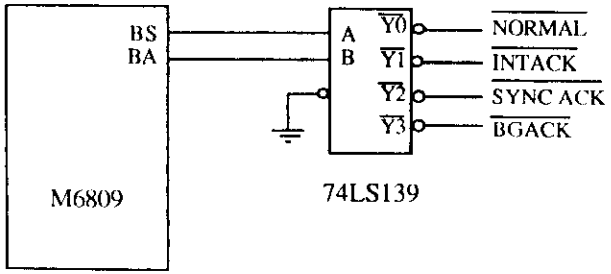
The SLOW signal is generated by the decoder detecting that the CPU wants to address the slow device. The delay needed can be adjusted using different outputs of the counter.

CPU state definitions

The M6809 has two status outputs, BA and BS, indicating the CPU state to the external world. These signals are needed only for advanced hardware implementations like interrupt vectoring or DMA techniques. It would exceed the scope of this lecture to explain them in detail, we just mention the different states and show how to decode them with a simple piece of hardware.

BA	BS	CPU State
0	0	Normal
0	1	Interrupt or Reset Ac knowledge
1	0	Synch Acklowledge
1	1	Halt or Bus Grant Acknowledge

A simple State Decoder:



Instruction Timing

The M6809 is a synchronous machine and therefore the instruction timing can be easily determined if no 'slow devices' are used. The standard version of the CPU runs with a 4MHz clock divided by 4 internally. The basic machine cycle is therefore 1µs.

How to find the execution time of a program? The M6809 reference card (pocke card) has for each instruction and each addressing mode of these instructions an entry marked '~' indicating the number of machine cycles used. These numbers refer to the complete instruction, including instruction fetch, decoding, address calculations and execution. The '+' in the column of indexed addressing modes means that one has to add between 0 and 8 cycles depending on the type of indexed addressing used. There is another table containing this information.

A second possibility is to use the listing of an assembler which also indicates the execution times in terms of clock cycles.

Examples:

instruction	N° of bytes	N° of cycles
CMPA <$F3	2	4
LEAY 5,Y	2+0	4+1
LDA #1	2	2
LDA [$200,PCR]	2+2	4+8
SWI	1	13

Example for a delay loop:

DELAY	EQU	*	_____
	PSHS	X	_____
	LDX	#N	_____
LOOP	LEAX	-1,X	_____
	BNE	LOOP	_____
	PULS	X,PC	_____
		TOTAL	_____

Cycle by Cycle Operation

The cycle by cycle operations of the M6809 are described in the M6809 data sheet. Hardware designers or maintenance people have to know about them in order to understand the signals generated on the buses. For complicated hardware problems a Logic State Analyser may be required. For M6809 based systems 24 channels are sufficient, 32 are preferable. We give here as an example the cycle by cycle operation of a read-modify-write instruction using the extended addressing mode.

The instruction is ASL $312C, the program counter would be at $1000 and the contents of location $312C is supposed to be $01 before instruction execution.

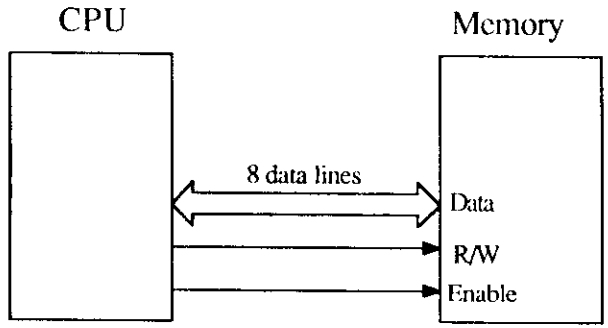
Cycle	Address	Data	R/W line	Function
1	1000	78	1	Opcode fetch
2	1001	31	1	Operand address read high
3	1002	2C	1	Operand address read low
4	FFFF	*	1	Internal cycle (VMA low)
5	312C	01	1	Operand read
6	FFFF	*	1	Internal cycle (VMA low)
7	312C	02	0	Data write
	1003	next instruction		Opcode fetch

Lecture 5: Connecting Memory and I/O

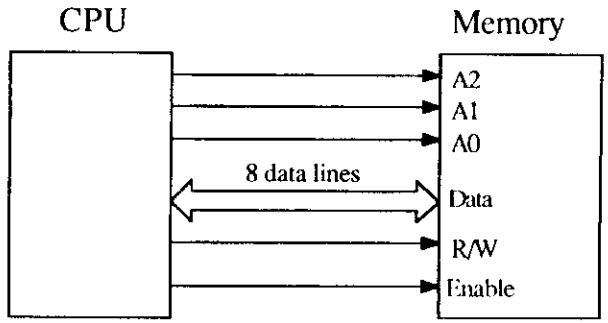
- Addressing memory
- Internal decoding
- External decoding
- Base Address Selection
- Partial Decoding
- Case Study: A Simple I/O Device

Addressing Memory

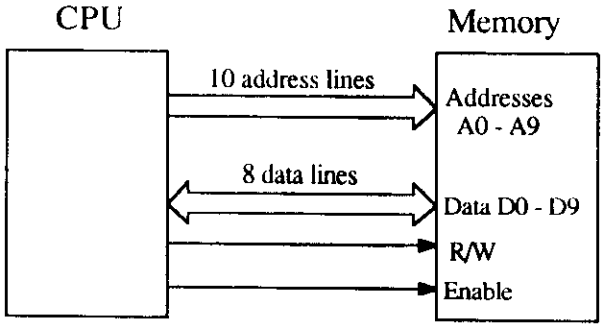
Addressing a one byte memory:



Adding three address lines:



What is the capacity of this memory ?



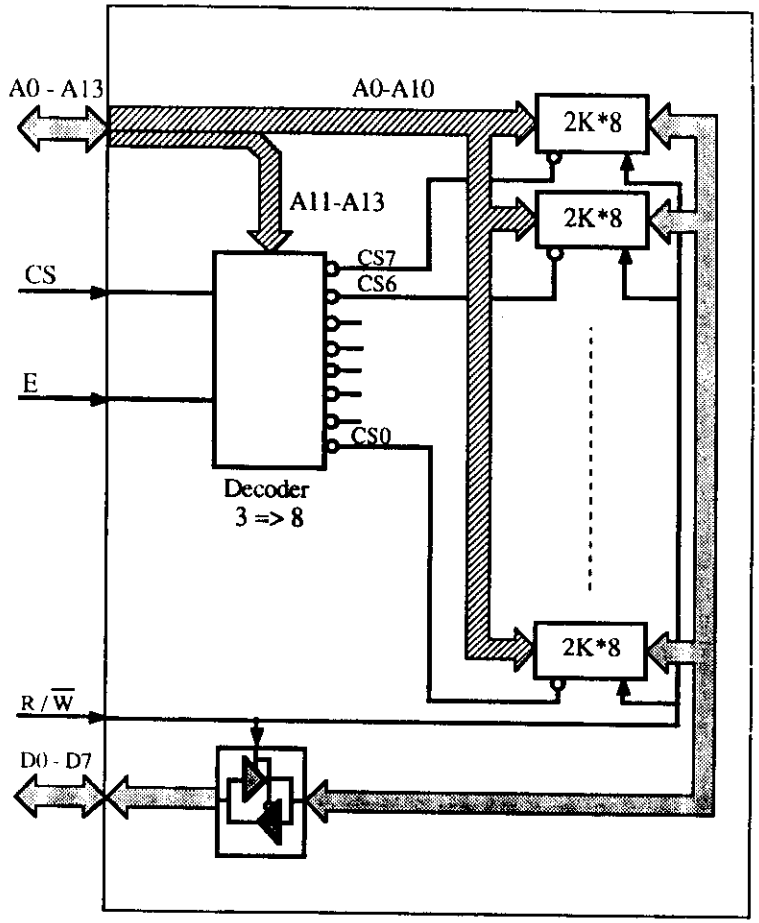
Answer:

10 address lines allow 2^{10}=1024 combinations, the capacity of this memory is 1 Kbytes

Note: The read/write signal R/W is used to determine the direction of the data exchange and the Enable signal provides the timing. Usually, R/W is comes together with the address signals and the Enable gets activated when the data are stable.

Internal Decoding

The next diagram shows the internal decoding for a 16K*8 multi-chip memory



Example: CS6 = CS • E •A13 • A12 • $\overline{A11}$

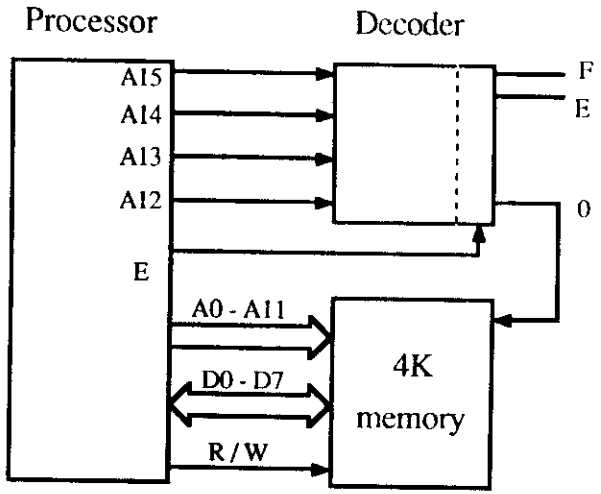
In this example, the memory consists of 8 chips with 2K bytes each. Each chips needs 11 address lines (A0 - A10) to address one out of 2048 bytes. The next 3 lines (A11 - A13) are used to select the one out of 8 chips at any time using a 3 line to 8 line decoder. The chip select signal CS validates the decoder and the enable signal E provides the timing. The read / write signal R / W* is determines the direction of the data transfer.

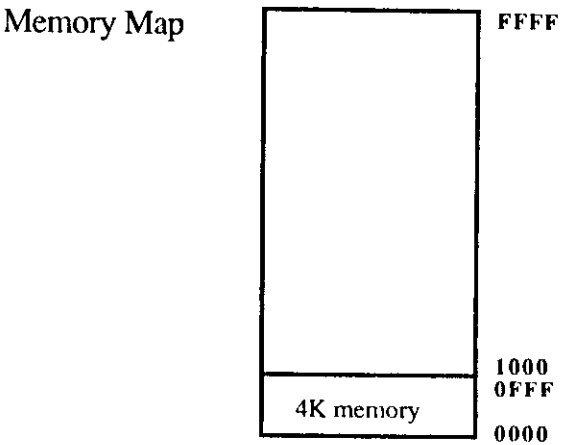
Note: Using the next higher address lines A11 - A13 to select the chips provides a contiguous memory space.

External Decoding

Connecting a memory system to a CPU requires additional decoding to locate the memory in the available address space, which is usually much larger then the space occupied by the memory. The next figure is an example showing how to connect a 4K memory to a CPU with 64K address space. The memory itself may well be constructed from several chips and have its internal decoding. We use a 4 line to 16 line decoder to divide the address space in 16 equal pieces of 4K each. There are many other possibilities to decode addresses, for example using PROMs, PALs, comparators, etc. The choice depends upon the requirement of the overall system and on the possible need for later upgrades.

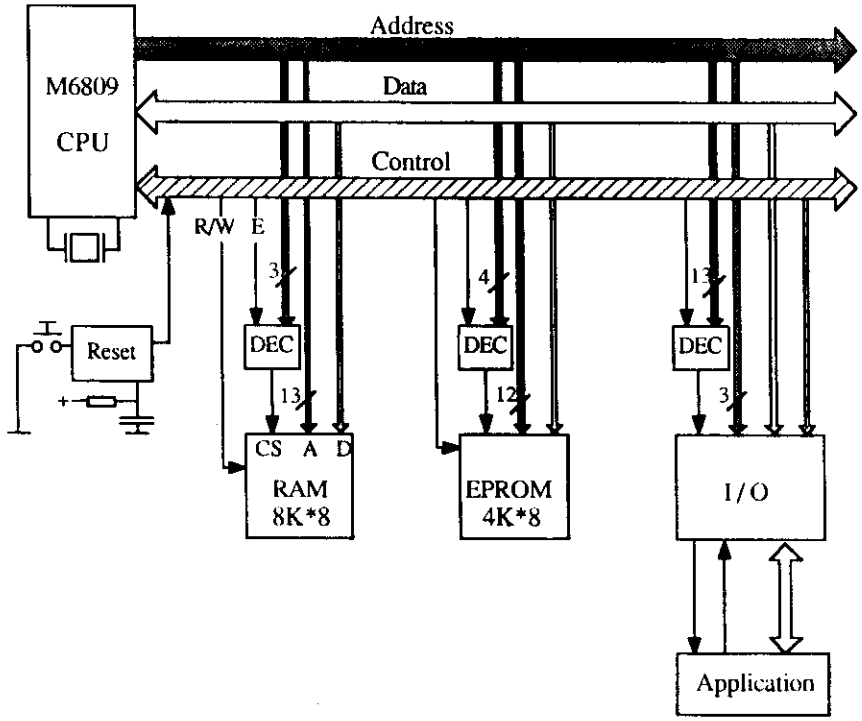
Connecting a 4K byte Memory to a 16 bit System





A Minimum System

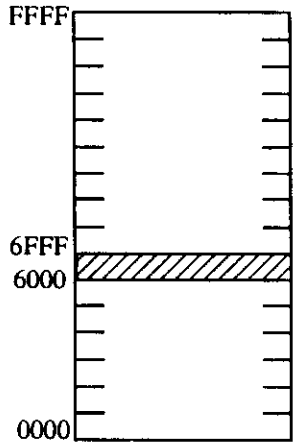
A minimum system consists of a CPU, EPROM or ROM memory, RAM memory and some I/O circuit. The system has an address bus with 16 lines, a data bus with 8 lines, and a control bus which carries the control and timing signals. Each unit hooked up to the bus uses a certain number of address lines, say n, for internal decoding and 16-n lines for external decoding. The external decoding determines the memory map. For the M6809 the classical configuration is RAM starting from $0000, EPROM at the top of the map and I/O below the EPROM.



The Concept of the Base Address

Example 1: **A 4K*8 memory on the M6809**

- The M6809 has 16 address lines fi 64K addresses
- A 4K*8 memory needs 12 address lines
- We have **16** possibilities to place a 4K block in 64K

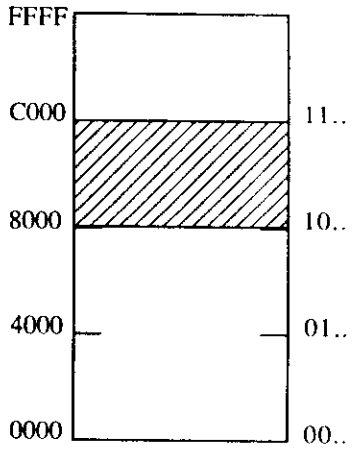


$A_{15}\ A_{14}\ A_{13}\ A_{12}$ $A_{11}\ A_{10}\ A_9\ A_8\ A_7\ A_6\ A_5\ A_4\ A_3\ A_2\ A_1\ A_0$

Base Address Internal Address

Example 2: A 16K*8 memory on the M6809

- A 16K*8 memory needs 14 address lines
- We have **4** possibilities to place a 16K block in 64K

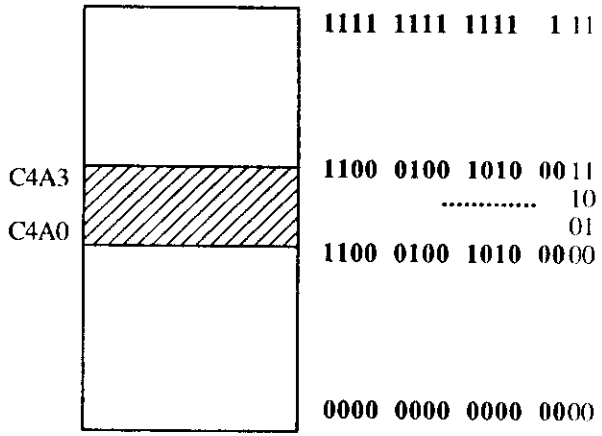


A_{15} A_{14}
Base Address

A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
Internal Address

Example 3: A 4 byte I/O device on the M6809

- A 4 byte I/O device needs 2 address lines
- We have **16384** possibilities to place a 4byte block in 64K



A_{15} A_{14} A_{13} A_{12} A_{11} A_{10} A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2
Base Address

A_1 A_0
Internal Address

Partial Decoding

As can be seen from the last example, full decoding of addresses for I/O devices is rather complex. Normally there is enough address space available to accomodate the devices needed. Partial decoding is a technique to reduce hardware complexity for applications, where full decoding is not needed. In the following example we show the partial decoding for a 4 byte device using only 8 address lines.

$A_{15}\,A_{14}\,A_{13}\,A_{12}\,A_{11}\,A_{10}\,A_9\,A_8$	$A_7\,A_6\,A_5\,A_4\,A_3\,A_2$	$A_1\,A_0$
Base Address	unused	Internal Address

For a device at address $C400 we have

1100	0100	XXXX	XX11
			10
			01
1100	0100	XXXX	XX00

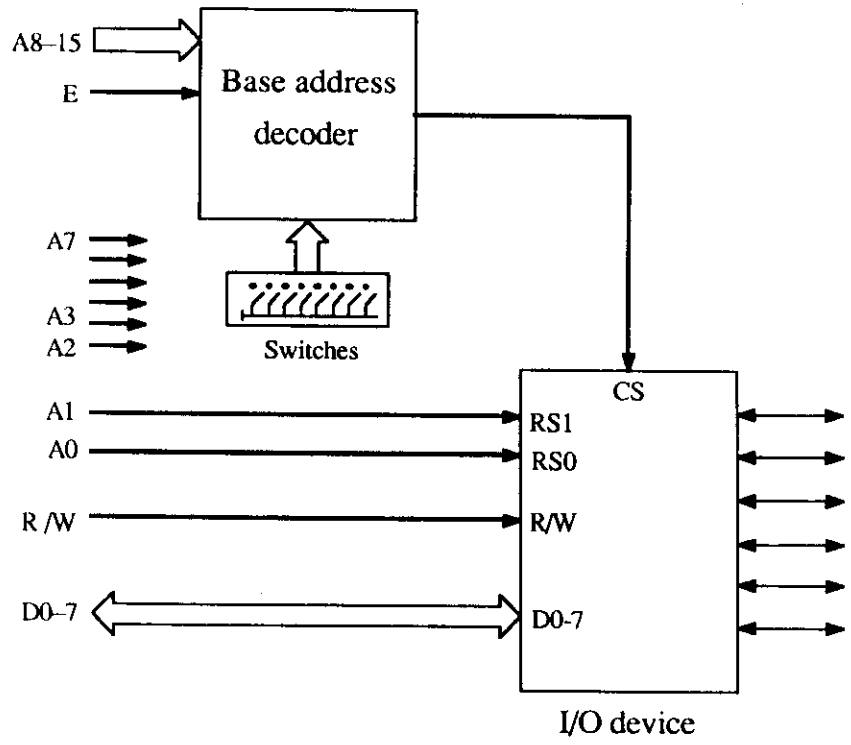
The device will also respond to the addresses

C4FC	–	C4FF
C4F8	–	C4FB
	...	
C484	–	C487
	...	
C404	–	C407
C400	–	C403

fi The device exists **64** times in the memory map

Practical Implementation

Whenever the timing signal E becomes valid the base address decoder compares the pattern defined on the switches with the address line A8 to A15. If they match, a chip select signal CS is generated activating the internal circuitry of the I/O device. Address lines A0 and A1 determine the internal register to be used. Address lines A2 to A7 are not connected, and therefore the device will respond to a range of 64 times 4 addresses.



Lecture 6: Advanced Addressing modes

- Application of Indexed addressing modes
- Case Study: Average of Arrays
- Case Study: Circular Buffer

Application of Indexed Addressing Modes

Reminder:

• constant offset from register	35,Y
• accumulator offset from register	B,X
• auto increment / decrement register	,S++
• constant offset from program counter (PC)	label,PCR

Note: All these modes exist also as "indirect"

LDX [ARRAY,Y]

This lecture deals with auto increment / decrement and indirect and examples, questions and answers

The appendix contains an overview of the indexed addressing modes, followed by examples for auto increment/decrement and the indexed indirect addressing. Rather then spending time on going through each instruction, we will present two applications, the first one calculating the average of two arrays element by element storing the result in a third array, the second using a circular buffer.

Case Study: Average of two arrays

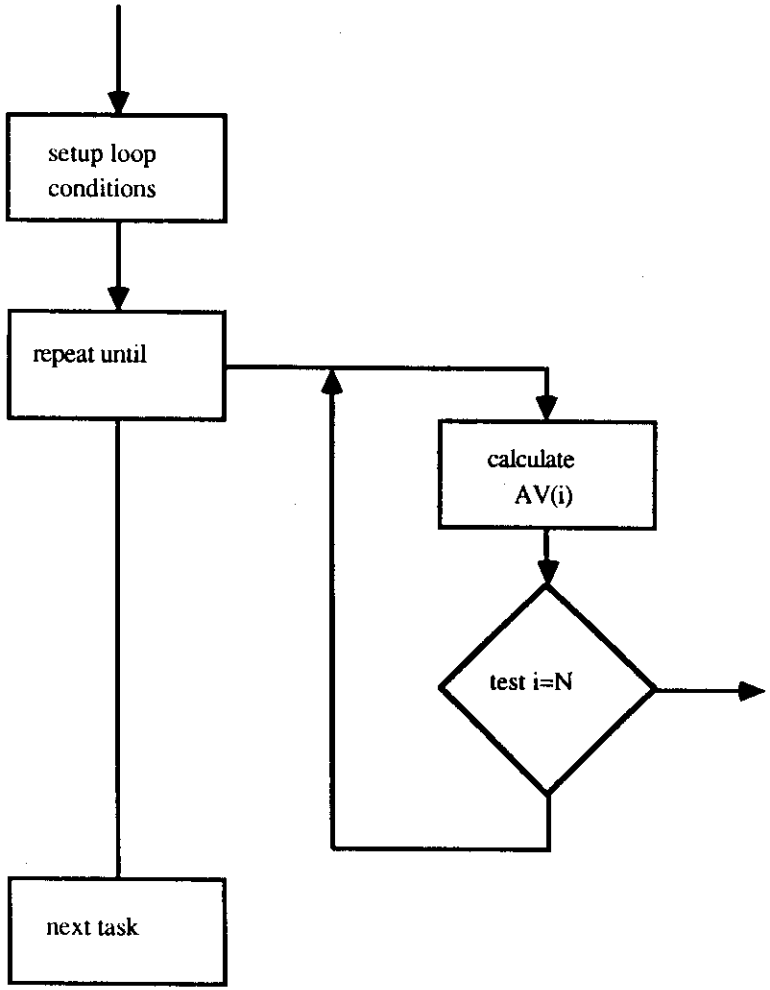
We would like to perform the following operation on the arrays K(i), L(i) and AV(i):

$$AV(i) = (K(i) + L(i))/2$$

Such a program written in Basic would look like

```
FOR I=1 TO N
AV(I) = (K(I) + L(I))/2
NEXT I
```

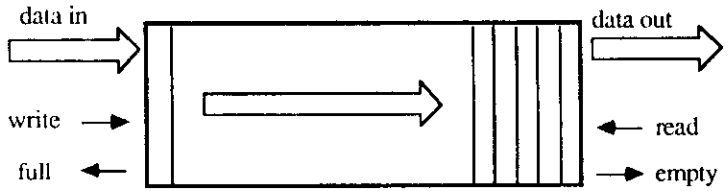
where N is the number of elements in each array, assuming of course that all three arrays have the same length. To write the same program in assembly language, we will first define the algorithm and draw the corresponding flow-chart.



```
        NAM     AVERAGE
*
*       This program calculates AV(I)=(K(I)+L(I))/2, where AV,
*       K and L are one-dimensional arrays of equal length N.
*       The arrays have been previously initialised by another
*       program. The data are assumed to be small enough not
*       to produce overflows.
*
        LIB     MONCALLS    include ROSY definitions
*
*       Reserve space needed
*
N       EQU     10          length of arrays
AV      RMB     N           array for result
K       RMB     N
L       RMB     N
*
*       PROGRAM STARTS HERE
*
START   EQU     *
        LDX     #0          index for all arrays
*
NEXT    EQU     *
        LDA     K,X         get element K(i)
        ADDA    L,X         add L(i)
        ASRA                divide by two (signed)
        STA     AV,X        store result in AV(i)
*
        LEAX    1,X         next element
        CMPX    #N          all done?
        BLO     NEXT        if not, do next
*
EXIT    EQU     *
        MON     RETURN      back to system
```

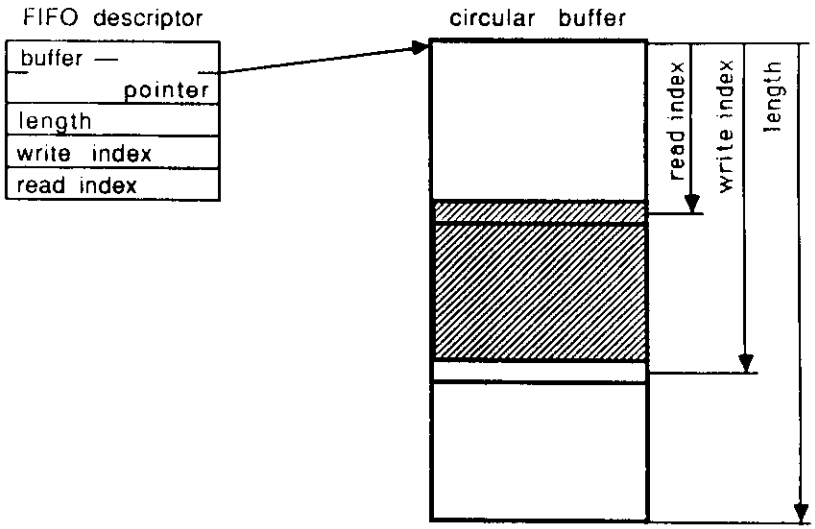

Case Study: 'Circular buffer'

Data from peripherals often comes in bursts or at times where it is not expected, whereas programs that handle the data are in general executing steadily. Circular buffers are used to derandomize data-flow and to isolate the data collection function from the data analysis function. Examples are the asynchrounous data transmission by a network or a type-ahead buffer for a terminal such as in ROSY.



In practice, it would be unreasonable to implement a FIFO in this way because it would imply many copies from memory to memory and the "fall-through time" becomes a function of the amount of data in the buffer. A circular buffer provides the same functions, but is by far more efficient.

The 'circular buffer' concept



A circular buffer consists of a piece of memory called 'buffer', which is managed with two pointers, the 'write pointer' and the 'read pointer' (also called 'head' and 'tail' pointers). At the beginning, both point to the same memory location, not necessarily the first of the buffer, and that condition is called 'buffer empty'. Each time a byte is stored in the buffer, the write pointer moves forward by one, until it reaches the end of the buffer, where it is set back to its beginning. Should it reach the read pointer from the back, no writing is performed and this condition is called 'buffer full'. Each time a byte is read from the buffer, the read pointer moves forward towards the write pointer and reaches it eventually when the buffer becomes empty.

In summary:

If read index = write index: buffer empty

If write index= read index-1: buffer full

Two routines are needed to control such a scheme, namely **PUT** and **GET**, which will write or read one byte from the buffer. We give an example for the PUT routine the GET routine can be derived easily from it. Both routines use the carry bit in the CPU's status register to return a 'completion code', i.e. if C=1, the operation was successfull, else C=0 (buffer full or empty). From the user's program the calling sequence may be

```
LDX     #FIFO       points to the descriptor
LDA     #'H         we want to store ASCII 'H'
JSR     PUT
BCS     OK          PUT returns C=1 for success,
                    C=0 if buffer was full
```

What follows is an example for the subroutine PUT. The convention is that ACCA contains the byte to be stored and the index register X points to the FIFO descriptor.

```
*
*       Subroutine PUT
*
*       Stores the contents of ACCA in a circular buffer who's
*       descriptor is pointed to by X. On return, C=1 if ok, C=0
*       if buffer already full
*
PUT     EQU     *
        PSHS    Y,B,CC          save registers to be used
        LDB     3,X             If write index+1 = length,
        INCB
        CMPB    2,X
        BNE     PUT1
        CLRB                      then set it to begin of buffer
PUT1    EQU     *               If new write index = read index,
        CMPB    4,X
        BNE     STORE
        LDB     0,S               then flag buffer full
        ANDB    #$FE                    set C=0 in CC on stack
        STB     0,S
        BRA     PUTOUT
STORE   EQU     *                 else store date
        LDY     0,X                     pointer to buffer
        STA     B,Y
        STB     3,X                     update write index
        LDB     0,S                     set C on stack
        ORB     #1
        STB     0,S
PUTOUT  EQU     *               endif
        PULS    CC,B,Y,PC       return with original registers
```


Here we show a second version which uses a few tricks to get it faster. Note that the CC register is not saved.

```
*
*       Subroutine PUT (fast version)
*
*       Stores the contents of ACCA in a circular buffer who's
*       descriptor is pointed to by X. On return, C=1 if ok, C=0
*       if buffer already full
*
PUT     EQU     *
        PSHS    Y,B             save registers to be used
        LDB     3,X             If write index+1 = length,
        INCB
        CMPB    2,X
        BNE     PUT1
        CLRB                      then set it to begin of buffer
PUT1    EQU     *               If new write index = read index,
        CMPB    4,X
  (*)   BEQ     PUTOUT
        LDY     0,X             else store date
        STA     B,Y
        STB     3,X                     update write index
        ORCC    #1                      set flag
PUTOUT  EQU     *               endif
        PULS    B,Y,PC
```

(*) if two operands are equal, the C-bit is always set to 0.

Appendix to 'Indexed addressing'

The following slides show the different flavours of the indexed addressing modes. For more information, please consult the M6809 programmers manual.

Lecture 7: Interrupts

- I/O Methods
- Interrupt Principle
- Interrupts with the M6809
- Interrupt Vector Table
- Condition Codes for Interrupts
- Stacking of registers
- IRQ Timing example
- Case Study

Introduction:

The three classical methods of input/output are

- programmed
- interrupt driven
- direct memory access

They can be classified as follows:

	programmed	interrupt	DMA
hardware	low	low	high
software	simple,limited	medium	complex
speed	low	medium	high

The actual transfer speed in both programmed and interrupt mode is not different, but the programmed mode will be much slower and cumbersome to program, if many I/O devices have to be serviced. Both interrupts and DMA allow for 'multi-tasking', programmed I/O needs complex 'polling'.

M6809 Characteristics: Interrupts

Basic principle of interrupts

When an interrupt occurs, the context of the running program must be preserved, the control is passed to the 'interrupt service', which does the necessary work to handle the interrupt. At the end of this 'service', the previous status must be restored and control is returned to the interrupted program.

This means, that an interrupted program does not realize that it was interrupted, except that its execution is slowed down.

Interrupts with the M6809

The M6809 has the **hardware** interrupts Reset, NMI, IRQ and FIRQ and the **software** interrupts SWI, SWI2 and SWI3.

The CPU handles all hardware interrupts in a very similar way, the differences will be explained as we go along. When an interrupt occurs, the M6809 finishes the execution of the present instruction, saves all the registers on the S stack (only the PC and the CC for FIRQ), updates the interrupt mask bits I and F and the E bit. It then fetches the interrupt vector from the table located at the top of the memory map and puts it in the PC, which means that instruction execution is passed to the 'interrupt service' routine. This routine should clear the interrupt source and finally execute the 'RTI' instruction, whereby the CPU will recover its status from the stack and resume instruction execution in the interrupted program.

The **Reset** is not really an interrupt, because it does not save the CPU status, but it is usually included in the discussion of the interrupts due to the way it is done in the CPU.

The **NMI** or Non-Maskable-Interrupt is an edge triggered input to the CPU. As the name says, it cannot be masked and whenever a high to low transition is detected on the NMI input, the CPU will execute the NMI sequence. The only exeption is after Reset, where the NMI is blocked until the S stack pointer is loaded for the first time. **Caution:** If more than one device is connected to the NMI line, special care has to be taken to avoid dead-locks, which may arise when the second device pulls the NMI line down while the first one is being treated and the line was low already; no further transition can occur and the system may hang.

The **IRQ** or Interrupt ReQuest is a level sensitive interrupt input of the M6809 and is the most commonly used interrupt, because it is the easiest to use.

The **FIRQ** or Fast Interrupt ReQuest is foreseen for devices which need a fast service and where only a small number of CPU registers are needed to service it. It is the programmers responsability to save such registers and to restore them before the RTI is executed.

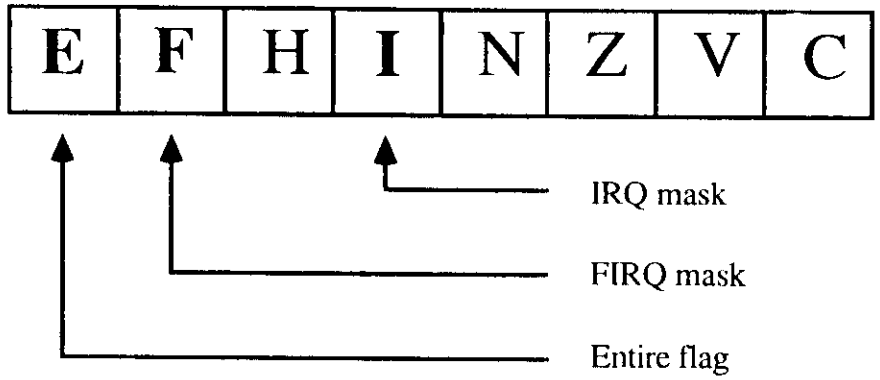
The **SWIs** or SoftWare Interrupts are under program control and they are very useful to implement operating system features such as breakpoints and monitor calls. Monitor calls are independent of changes in the system software, as long as the numbers are fixed, and they also allow the system to use all registers, because the CPU saves them automatically. The interrupt sequence for SWIs is identical to the hardware interrupt sequence, with the only the difference that they are synchronized to the flow of the program.

Interrupt vector table

Reset	FFFE/F	highest priority
NMI	FFFC/D	
SWI	FFFA/B	
IRQ	FFF8/9	
FIRQ	FFF6/7	has priority over IRQ
SWI2	FFF4/5	
SWI3	FFF2/3	lowest priority
reserved	FFF0/1	

The priorities are determined through the way the interrupt mask bits are set in the condition code register (CC) by the interrupts. The next figure shows the position of the bits related to interrupts. The 'Entire' bit is needed, because there is only one RTI instruction to be used both with normal interrupts and the FIRQ. E=1 means that all CPU registers have been saved on the stack, for E=0 only the PC and the CC registers are saved. Note that this bit is set/cleared by the CPU before CC is saved on the stack.

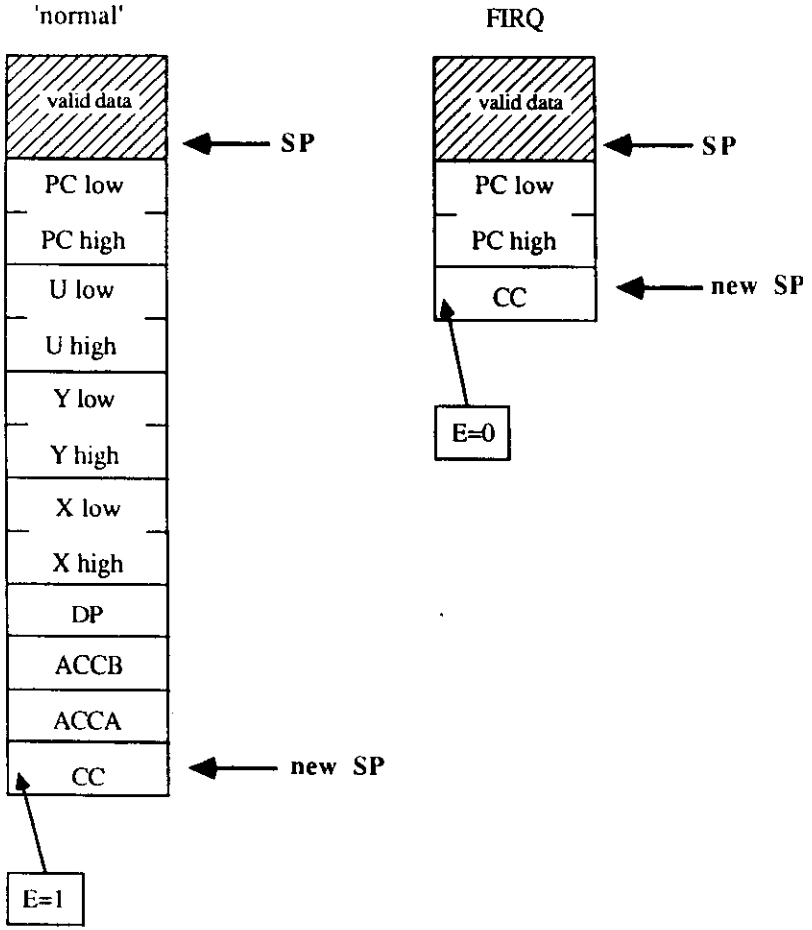
M6809 Characteristics: Interrupts



Interrupt	bits set	bits not set	will go through
Reset	I, F		NMI after 'LDS'
NMI	E, I, F		NMI
SWI	E, I, F		NMI
IRQ	E, I	F	NMI, FIRQ
FIRQ	I, F	E	NMI
SWI2	E	I, F	NMI, FIRQ, IRQ
SWI3	E	I, F	NMI, FIRQ, IRQ

Stacking of registers

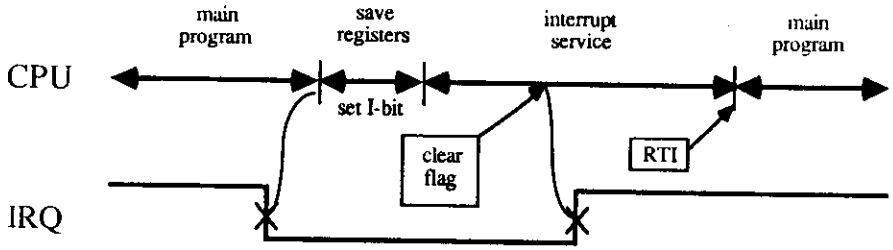
The CPU saves the registers automatically when an interrupt has been accepted. We distinguish two cases:



When 'RTI' is executed, CC is pulled first and therefore the CPU will know, if all registers have to be recovered or only the PC in the case of the FIRQ.

Timing example for IRQ

The following example shows a typical IRQ sequence. Note that the IRQ line goes up as soon as the interrupt condition in the peripheral chip has been cleared, but the interrupt service routine is only finished when RTI is executed. Should another interrupt arrive on the same line before the RTI, the IRQ line goes low again or stays low, but the interrupt will only be serviced after the RTI, because the I-bit will be up during the complete service time.



Case study

In the following, we give an example of using the IRQ with a PIA on the ROSY station. We assume, that a push-button has been connected to CA1 and that we like to see the high-to-low transition. Whenever the button is pressed, we increment a counter. If the counter reaches a predefined maximum value, the main program is informed by setting a flag and a message is printed.

The general strategy is as follows:

Main program:

1. define the interrupt vector
2. initialize the hardware
3. enable the IRQ
4. wait for flag and do other things

Interrupt service:

1. determine interrupt source
2. clear interrupt flag and increment counter
3. if count=max, set flag and clear counter
4. return from interrupt


```
        NAM     COUNT
*
*       This program uses a push-button connected to CA1 of the PIA at address
*       $EF08 to count pulses via interrupts. If a maximum count is reached, an
*       event flag is set for the main program, which will print a message.
*
        LIB     MONCALLS        include ROSY definitions
*
EOT     EQU     4
PIA     EQU     $EF08           define base address
ORA     EQU     PIA
CRA     EQU     PIA+1
*
*       reserve variables needed
*
CNT     RMB     1               counter
EVENT   RMB     1               event flag
MAXCNT  EQU     100             define max count
*
*       Main program starts here
*
Start   EQU     *
        CLR     CNT             init all stuff
        CLR     EVENT
        LEAX    PUSHB,PCR       vector for push-button
        LDB     #3              vector code for IRQ
        MON     VECTOR          set vector in ROSY
        TSTB                    any error?
        BNE     WRONG           yes, in deed
        LDA     #5              initialize PIA
        STA     CRA             set access ORA and int. enable
```

```
        ANDCC  #$EF             now enable IRQs
*
*       Main loop starts here
*
MAINLP  EQU    *
        TST    EVENT            any event?
        BEQ    NONE             no !
        LEAX   MAXMSG,PCR       send message
        MON    PRINT
        CLR    EVENT            reset flag
*
NONE    EQU    *
        •
        here we can do other things
        •
        BRA    MAINLP
*
MAXMSG  FCC    /Maximum count reached/
        FCB    EOT
*
WRONG   EQU    *                Error handling
        MON    ERROR
        MON    RETURN           give up if error
```

M6809 Characteristics: Interrupts

```
*
*       Interrupt service routine
*
PUSHB   EQU    *
        LDA    CRA              was it PIA ?
        BMI    FOUND            yes
        LDB    #14              send error message
        MON    ERROR            "Undefined IRQ"
        BRA    P_OUT
*
FOUND   EQU    *
        LDA    ORA              clear CRA-7 flag
        LDA    CNT              increment count
        INCA
        CMPA   #MAXCNT          maximum reached ?
        BLO    NEXT             not yet
        CLRA                    reset cnt, set flag
        INC    EVENT
NEXT    EQU    *
        STA    CNT              save new count
P_OUT   RTI                     That's all folks !
*
        END
```