EE 3265 MICROPROCESSOR SYSTEMS

Contents

A. UCT SABUS MONITOR V3.1

1 SYSTEM DESCRIPTION

1.1 HARDWARE

8085 CPU, 64k dynamic RAM, 2 serial channels, one 24-line parallel port, user programmable timer, up to 256k ROM.

1.2 SOFTWARE

The 4K monitor (addresses F000-FFFF) provides facilities to

- display memory (ASCII, hex, disassembly)
- modify memory/register contents
- move/fill memory blocks
- access I/O ports
- set serial baud rates
- set/clear breakpoints
- single-step execution
- communicate with PDP 11/23
- display assembler symbol table.

Monitor command <u>arguments</u> may be hex numbers (4 digits max) or symbol table symbol names preceded by colons.

2. COMMANDS.

B - SET/CLEAR OR DISPLAY UP TO 16 BREAKPOINTS

B [<breakpoint #> [, <bp address>]] <CR>

S - subroutine execute T - trace U - select baud rate X - transparent communication

C - Copy memory block D - display memory area.

F - Fill memory G - execute I - input data

L - disassemble M - modify mem. O - load object file P - port I/O R register access

1. INTRODUCTION

As a replacement for hard-wired logic, the microprocessor offers lower cost, fewer components, and greater flexibility. The system designer requires both logic design and software expertise in order to achieve the optimum balance between the two.

	Random Logic	Microprocessor
Design	specific	general
Emphasis	hardware	software
Package count	high	low
Flexibility	low	high
Redesign/modification	difficult	easy??
Circuitry	redesign always	alike
Speed	very fast	moderate

Both have high development costs, but in volume production the lower component count of the microprocessor comes into effect.
In general µP systems can be divided into two groups

HIGH PERFORMANCE SYSTEMS (COMPUTATION & DATA PROCESSING)

Performance (ie speed and computational capabilities) is of utmost performance. Not possible to use random logic (although limited use of random logic can improve performance). µP choice considerations include:

- processor speed
- instruction set (capabilities, flexibility)
- software development and debugging aids available
- system interfacing capability (memory address range, DMA, etc)

Such system are normally produced in relatively small quantities and the software development costs are enormous, hence prices are high.

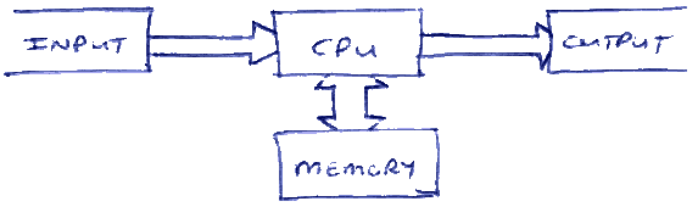
Low Performance Systems (Controllers)

The μP replaces an equivalent random logic design. Performance generally not important; component cost must be kept minimal (as generally high volume production). Software requirements are small and thus not expensive. μP choice considerations:

- chip count and cost
- external interfacing capabilities (I/O lines, etc)
- support components included (timer, serial I/O, etc)
- low power consumption

2. Basic Microprocessor System Architecture.

Fundamental architecture follows a simple von Neumann design



2.1 Memory

Normally a very large array of storage cells organised into words of a fixed width (number of bits). Memory sizes usually specified in units of $2^{10} = 1024$, written K, followed by the width, eg 2K×8 is 2048 bytes.

The CPU uses memory to store both program and data (the only difference between these is content). Memory devices may be read only, or read/write. They fall into two basic classes.

- Volatile – retains contents only while power is applied (RAM)
- Non-volatile – retains contents indefinitely (eg ROM, EPROM)

  Non-volatile read/write devices are available (eg EEROM, electrically-erasable ROM).

2.2 INPUT / OUTPUT

I/O locations (ports) are similar to memory, in that each port appears as a set of storage cells, normally of the same width as memory, and each port has a unique address, but unlike memory the contents of a port are usually accessible to an external system. I/O provides the only means by which the operation of a system may be altered, or by which it may alter another system's operation. Consequently, a μP system without I/O is useless.

2.3 THE CPU.

The central, large scale sequential circuit responsible for the co-ordination of the information flow within the system. The binary information which controls its operation is held in memory and the CPU must ensure that the information is retrieved and used in an orderly and defined manner.

3. COMPONENTS OF THE MICROPROCESSOR.

In its simplest form a μP is a large clocked sequential circuit. It can be separated into a number of logically discrete items, discussed below:

3.1 THE REGISTER

Each register is made up of a number of flip-flops which can be latched. The CPU can move and manipulate the information held in registers. Register size is typically (twice the) memory size eg an 8-bit machine will typically have 8- and 16-bit registers.

A register will usually have some combinational logic associated with it to perform simple manipulations called micro-instructions or micro-operations. A single μP instruction will cause a defined sequence of micro-ops to occur. Some common micro-ops are given below. Normally a register will not perform all of these, but a subset applicable to its purpose.

FUNCTION	MICRO-OPERATION
Load register R	$R_i = I_i$
Clear R	$R_i \leftarrow 0$
Complement R	$R_i \leftarrow \overline{R_i}$
Increment R	$R \leftarrow R+1$
Decrement R	$R \leftarrow R-1$
Shift R left	$R_{i+1} \leftarrow R_i$, $R_0 \leftarrow 0$
Shift R right	$R_{i-1} \leftarrow R_i$; $R_n \leftarrow 0$
Serial load R (left)	$R_{i+1} \leftarrow R_i$, $R_0 \leftarrow I$
Serial load R (right)	$R_{i-1} \leftarrow R_i$; $R_n \leftarrow I$

Some typical circuits to perform these functions are given below; note that in reality, the desired functions for any register will be combined into one circuit.

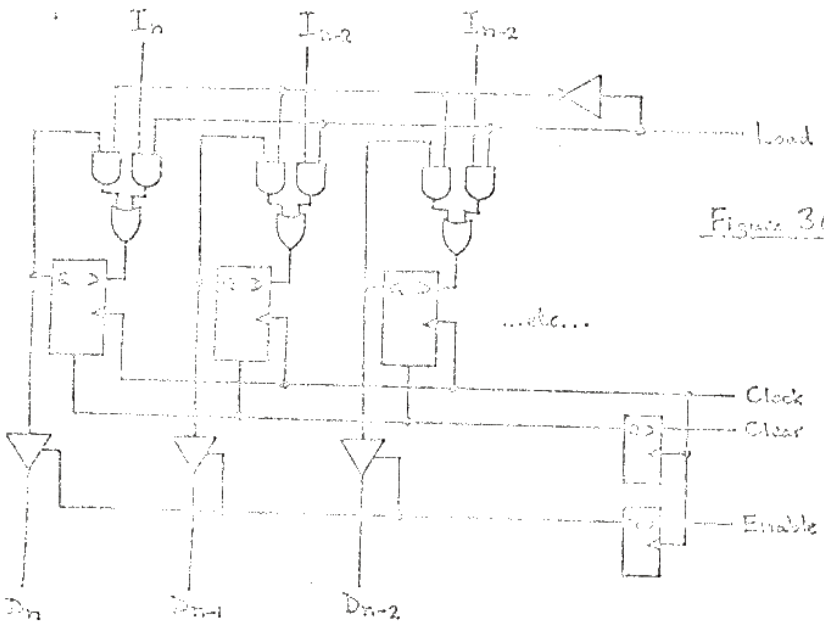


Figure 3(a): REGISTER

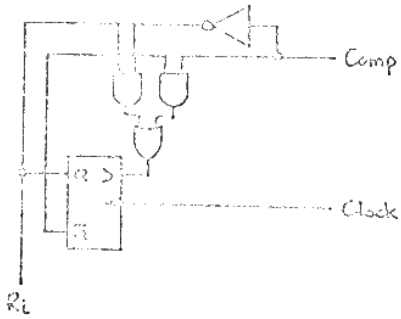
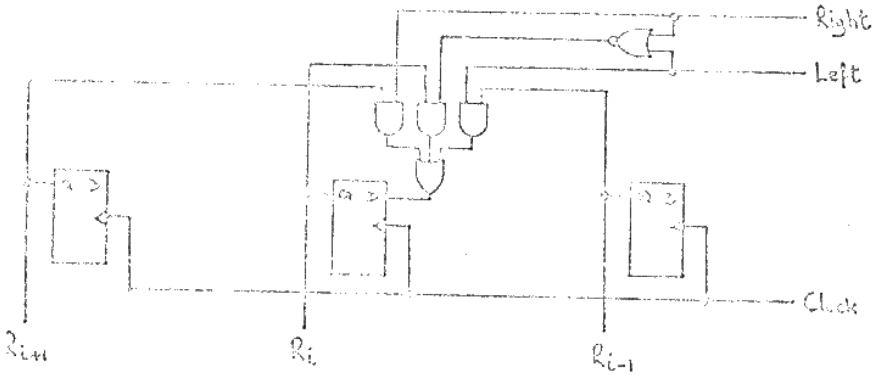


Figure 3(b): COMPLEMENT (n stage)

Figure 3(c): SHIFT LEFT/RIGHT (one stage)



3.2 THE BUS

All the input and outputs of the registers are commoned together via tri-state buffers, so that any registers' inputs may be driven by any other registers outputs. (see fig 3(a) on p5).

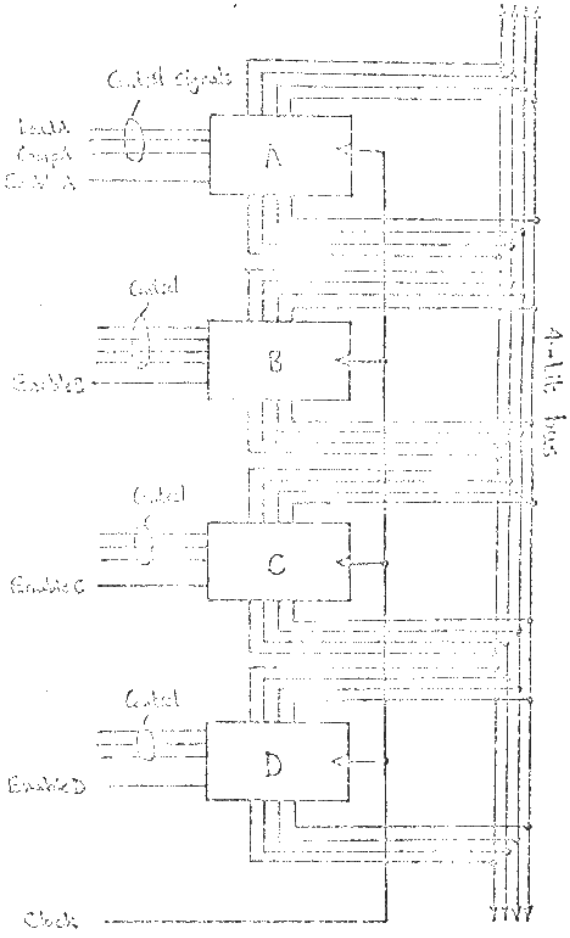
All control lines to the register are latched so that they take effect in synchronism with the μP clock signal. All the registers in the μP, and thus all the information transfers and manipulation, are synchronised to one master clock signal.

Eg to perform $A \leftarrow \overline{D}$, the following micro-ops would be performed:

clock cycle 1	Enable D = 1	(enable D's outputs onto bus)
clock cycle 2	Load A = 1	(write the bus state into A)
clock cycle 3	Comp A = 1	(complement A)
	Enable D = 0	(disable D's outputs)

The timing diagram for these is fig 5. Note that three

Figure 3: BUS STRUCTURE



Shorthand notation:

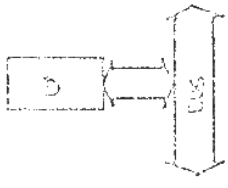
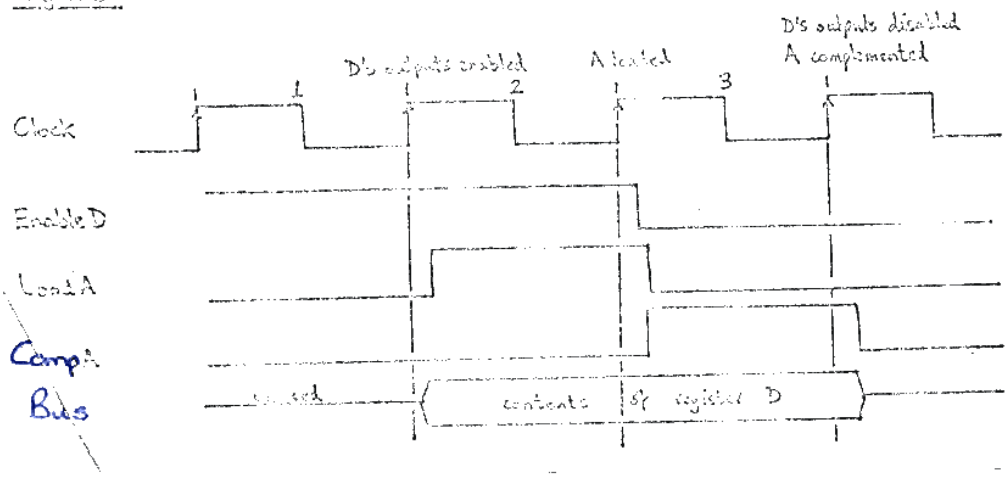


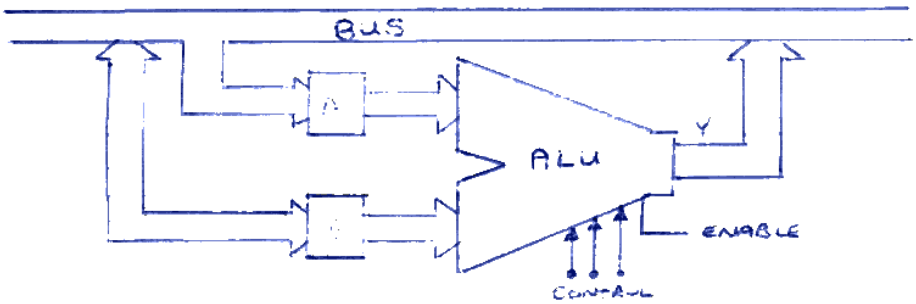
Figure 5: BUS TIMING



. clock cycles were used to achieve what was (presumably) one instruction. If a number of instructions were being performed, the first instruction would probably be performed during the last clock cycle of the previous instruction to increase the speed of the µP

3 3 THE ARITHMETIC / LOGIC UNIT

The ALU is a complex, dedicated combinational circuit used within the CPU to perform logic and arithmetic operations on register contents.



The ALU outputs, like registers, go through tri-state buffers before being connected to the bus. The ALU is obviously much slower than a direct register operation, for this reason some registers will have circuitry to perform the most frequent operations, the rest being done by the ALU.

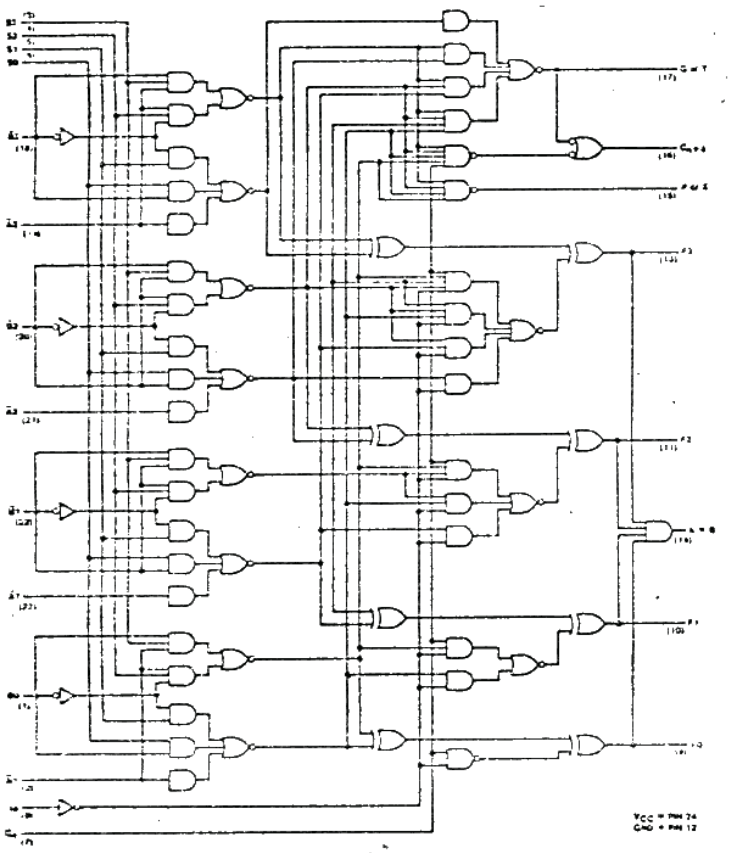
Some typical ALU operations might be:

FUNCTION	MICRO-OPERATION
Addition	$Y = A + B$
2's C subtraction	$Y = A + (\bar{B} + 1)$
Logical AND	$Y_i = A_i \wedge B_i$
Logical OR	$Y_i = A_i \vee B_i$
Logical XOR	$Y_i = A_i \oplus B_i$
Complement A	$Y = \bar{A}$
Increment A	$Y = A + 1$

The internal circuit of a 4-bit ALU is shown below

MSI DM54/DM74181, S181

Logic Diagram

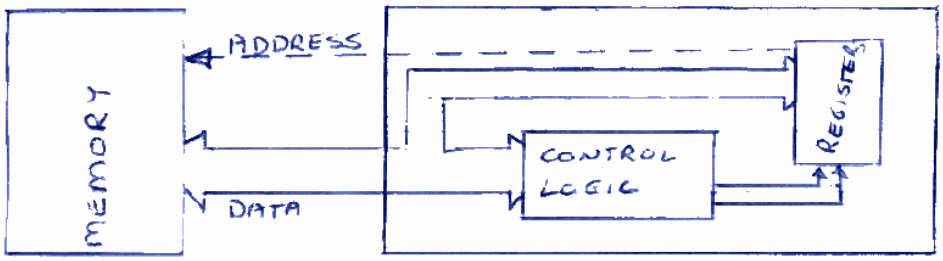


Selection				Active High Data		
				M = H	M = L; Arithmetic Operations	
S3	S2	S1	S0	Logic Functions	C_n = H (no carry)	C_n = L (with carry)
L	L	L	L	$F = \bar{A}$	$F = A$	$F = A$ Plus 1
L	L	L	H	$F = \overline{A + B}$	$F = A + B$	$F = (A + B)$ Plus 1
L	L	H	L	$F = \bar{A}B$	$F = A + \bar{B}$	$F = (A + \bar{B})$ Plus 1
L	L	H	H	$F = 0$	F = Minus 1 (2's Compl)	F = Zero
L	H	L	L	$F = \overline{AB}$	$F = A$ Plus $A\bar{B}$	$F = A$ Plus $A\bar{B}$ Plus 1
L	H	L	H	$F = \bar{B}$	$F = (A + B)$ Plus $A\bar{B}$	$F = (A + B)$ Plus $A\bar{B}$ Plus 1
L	H	H	L	$F = A \oplus B$	$F = A$ Minus B Minus 1	$F = A$ Minus B
L	H	H	H	$F = A\bar{B}$	$F = A\bar{B}$ Minus 1	$F = A\bar{B}$
H	L	L	L	$F = \bar{A} + B$	$F = A$ Plus AB	$F = A$ Plus AB Plus 1
H	L	L	H	$F = \overline{A \oplus B}$	$F = A$ Plus B	$F = A$ Plus B Plus 1
H	L	H	L	$F = B$	$F = (A + \bar{B})$ Plus AB	$F = (A + \bar{B})$ Plus AB Plus 1
H	L	H	H	$F = AB$	$F = AB$ Minus 1	$F = AB$
H	H	L	L	$F = 1$	$F = A$ Plus A*	$F = A$ Plus A Plus 1
H	H	L	H	$F = A + \bar{B}$	$F = (A + B)$ Plus A	$F = (A + B)$ Plus A Plus 1
H	H	H	L	$F = A + B$	$F = (A + \bar{B})$ Plus A	$F = (A + \bar{B})$ Plus A Plus 1
H	H	H	H	$F = A$	$F = A$ Minus 1	$F = A$

*Each bit is shifted to the next more significant position.

4 BASIC OPERATION OF THE MICROPROCESSOR.

4.1 INSTRUCTIONS AND DATA



SIMPLIFIED DIAGRAM OF A MICROPROCESSOR

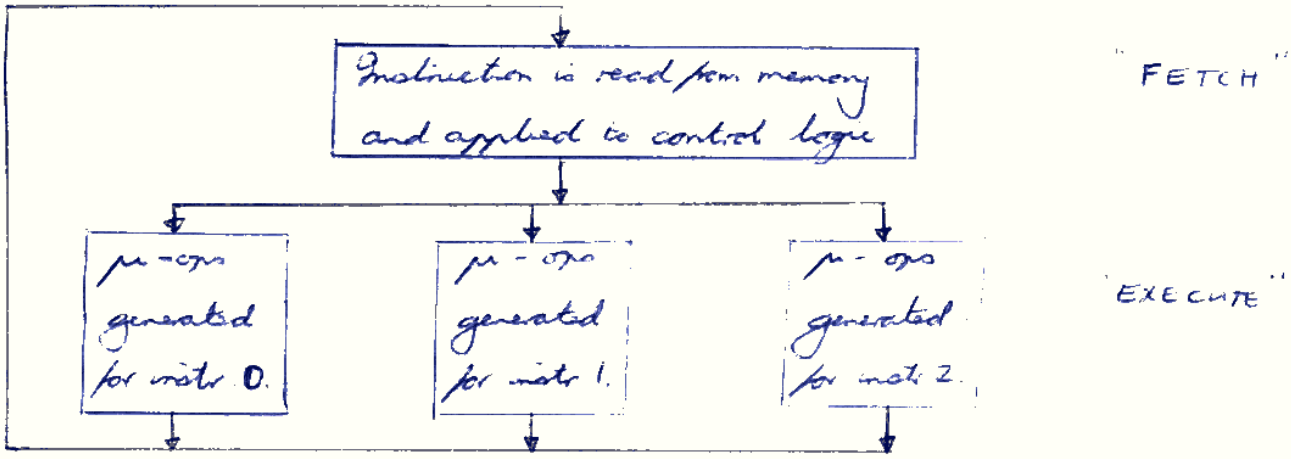
The binary data from memory is applied to both the registers and the control circuitry, and is interpreted in different ways

- the data applied to the control logic is interpreted as an instruction, and causes the necessary sequence of micro-ops to be performed
- the data in the registers is manipulated and moved under control of the micro-ops.

Both instructions and data share a common path to memory, the "data bus", whose width is the same as the memory width. There are thus 2^n possible instructions, many of which will be illegal

4.2 THE CONTROL LOGIC.

This is a small sequential circuit, whose function is to generate the control signals for the micro-operations. The inputs to the control circuitry are the bits of the instruction to be executed; the bits of each particular instruction will thus cause the control circuitry to cycle through a unique state sequence in order to generate the correct micro-operations to perform the instruction. This sequence is shown in a very broad form overleaf.



The first part of the instruction processing (FETCH) is done by micro-ops generated by the control logic itself - this first part is always performed, and the control-logic always generates the same μ-ops for it. The fetch cycle can thus be considered as the last few μ-ops of <u>every</u> possible instruction.

4.3 INTERNAL STRUCTURE OF A SIMPLE MACHINE (see pg 7)

As registers are not necessarily the same width, we assume that the bus is sufficiently wide to be able to transfer any register to any other register. Some of the registers have special functions

THE MEMORY ADDRESS REGISTER (MAR)

This is used to isolate memory from the internal bus of the μP, and also to keep the memory address stable while data is read or written (this address is written/loaded by the CPU)

THE MEMORY DATA REGISTER (MDR)

This is a bi-directional register (i.e. can be written to and read from both memory and CPU) used for temporary storage only; and never manipulated except by reads and writes.

THE INSTRUCTION REGISTER (IR)

The opcode is placed in the IR during the FETCH cycle, and is thereafter used by the control logic to generate the μ-ops.

THE PROGRAM COUNTER (PC)

Contains the address of the next instruction to be executed.

THE ACCUMULATOR (A)

This is the register on which most data manipulation occurs. It drives one input of the ALU directly.

THE CONTROL REGISTER

Loaded only by the control logic, this is used to provide the control signals to allow memory and I/O to be read/written, synchronising data flow between memory & CPU.

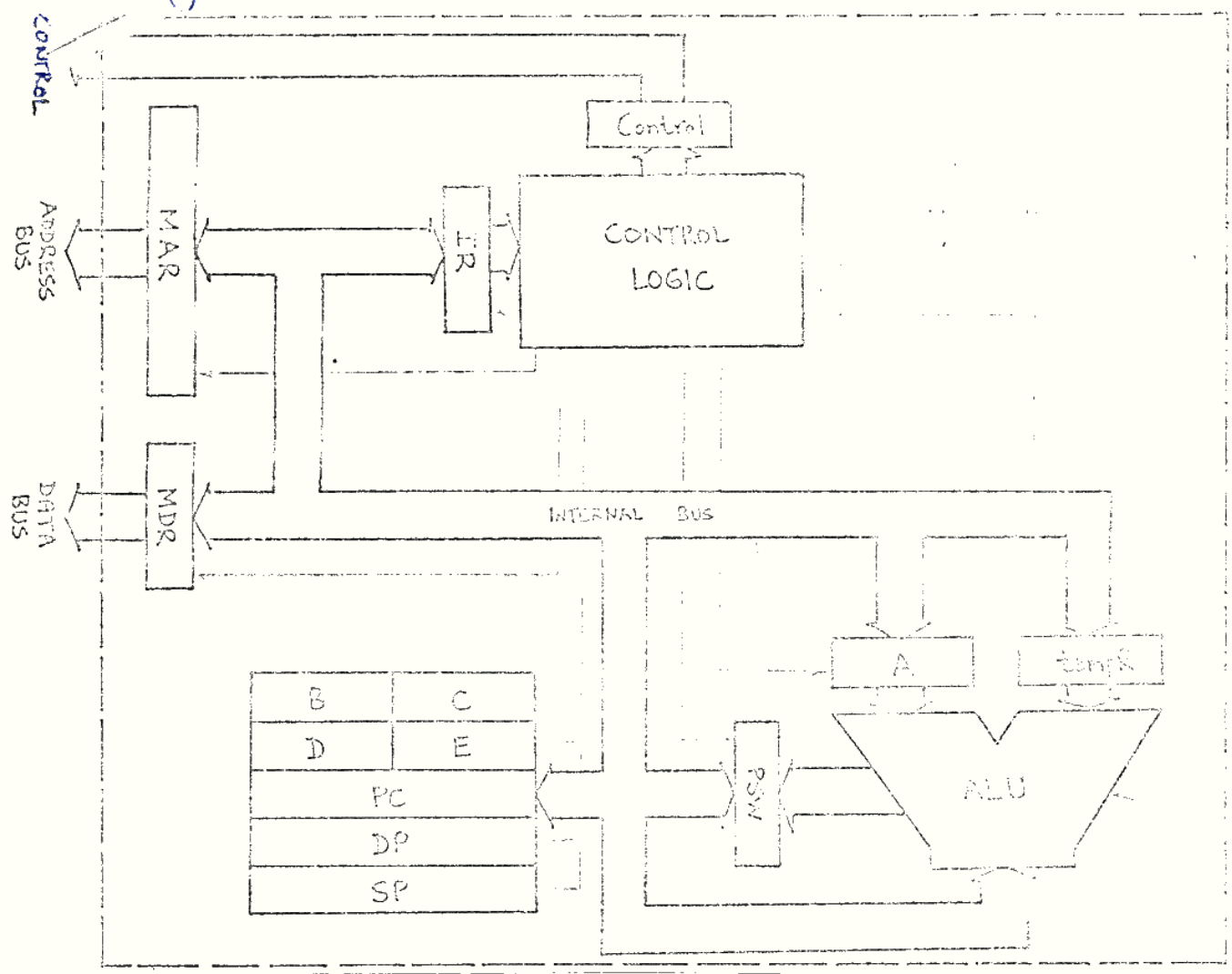


Figure 7: A SIMPLE MICROPROCESSOR

THE DATA POINTER (DP)

Used to contain the address of any operand data required by the instruction. Normally set up during FETCH and used during EXECUTE, during which its contents are loaded into the MAR.

THE STACK POINTER (SP)

Dedicated register used to point to the top of the system stack.

THE GENERAL REGISTERS (B, C, D, E)

General purpose working registers, the same width as the accumulator

We assume that the PC has auto-increment circuitry, and the SP has auto-inc and dec circuitry

4.4 MICROPROCESSOR INSTRUCTIONS

INSTRUCTION FORMAT

Each instruction must require:

- the operation to be performed (opcode)
- the source of the data item
- the source of the second data item, if applicable
- the destination of the result
- the location of the next instruction to be executed

However, we need not explicitly code all this information if we assume instructions appear in memory sequentially and implicitly use internal registers (typically the accumulator). When necessary, we can use multibyte instructions.

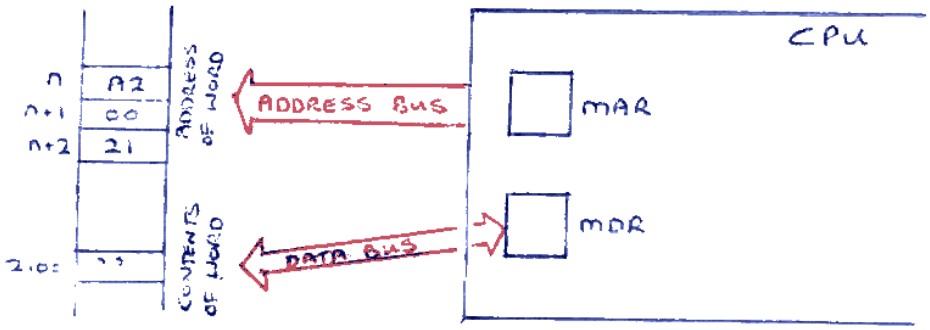
4.5 MICROCYCLES.

The execution of each instruction is made up of a number of smaller operations, µ-ops or microcycles, each taking one clock cycle to perform, and representing one data transfer or register manipulation operation.

Memory accesses require more than one cycle - the address must first be placed in the MAR, which causes the appropriate memory word to be enabled and connected to the MDR, only after which a read or write can occur.

Each microcycle involving data transfer between internal registers uses the internal bus, and thus only one such microcycle can occur at a time. Register manipulations and memory accesses can occur at the same time, however.

Eg ADD 2100, A assume opcode is A2. Then the memory locations and the connection to the CPU appear as follows.



Typical microcycles:

CYCLE	MICRO-OPERATION	DESCRIPTION
1	MAR ← PC	Opcode address into MAR
2a	PC ← PC+1	PC points to 1st operand byte
2b	MDR ← mem(MAR)	Put opcode into MDR
3	IR ← MDR	Transfer opcode to IR
4	MAR ← PC	Operand address into MAR
5a	PC ← PC+1	PC points to 2nd operand byte
5b	MDR ← mem(MAR)	Get 1st operand byte to MDR
6	DP(L) ← MDR	Put into LSByte of DP

7	MAR ← PC	2nd operand byte address to MAR
8a	PC ← PC + 1	Point to next opcode
8b	MDR ← mem (MAR)	Get 2nd operand byte
9	DP(h) ← MDR	Put into MSByte of DP
10	MAR ← DP	Address of operand data into MAR
11	MDR ← mem (MAR)	Get operand data byte
12	tempR ← MDR	Put into tempR of ALU
13	A ← A + tempR	Add A and tempR using ALU.

Notes.

- Four machine cycles are used to perform this instruction, the first three being fetch cycles using the PC, which is automatically incremented by its own built in logic during the next microcycle.
- Thirteen clock cycles are used
- The first fetch (opcode fetch cycle) is the same for all instructions. Thereafter any further fetch cycles are dictated by the opcode itself, which is now in the IR and influences the operation of the control logic.
- The entire instruction is an instruction cycle

5 MICROPROCESSOR INSTRUCTIONS

5.1 THE USE OF MNEMONICS

Machine code in mnemonic form is a crude language for representing machine instructions in a more legible way.

5.2 TYPES OF INSTRUCTIONS.

Instruction types fall into a number of categories:
- Data manipulation instructions (eg arithmetic, logic)
- Data transfer instructions.
- Program control instructions

5.2.1 DATA MANIPULATION INSTRUCTIONS.

These are performed by the ALU, and affect the status flags (flip-flops) comprising the processor status word (PSW) of the CPU. The most common flags found are
- Zero (Z) (set if result is zero)
- Sign (S) (a copy of the MSB of result)
- Carry (CY) (set if MSB is carried out)
- Auxiliary carry (AC) (a 4-bit carry flag for BCD arithmetic)
- Overflow (V) (indicates a possible error caused by signed overflow)
- Parity (P)

The control logic of the μP can prevent any of the flags from being affected by the ALU's operation

5.2.2 TRANSFER INSTRUCTIONS

These fall into three classes:

- memory transfer (actually mem - reg transfer)
- register transfer (reg - reg transfer)
- input/output (reg - port transfer)

5.2.3 PROGRAM MANIPULATION (CONTROL) INSTRUCTIONS.

These are used to affect CPU operation rather than to process or transfer data. Conditional control transfer instructions enable decision making within the CPU.

5.2.4 EXAMPLE 8085 INSTRUCTIONS.

DATA MANIPULATION

ADD	r	add
SUB	r	subtract
ADC	r	add with carry
SBB	r	subtract with carry
ANA	r	bitwise logical AND
XRA	r	bitwise logical XOR
ORA	r	bitwise logical OR
CMA		complement accumulator
RRC		rotate right circular
RAR		rotate accum. right
CMP	r	compare

TRANSFER

LDA addr	load acc.
STA addr	store acc.
MOV r1, r2	set r1 ← r2
OUT port	write acc. to port
IN port	load acc. from port

CONTROL

JMP addr	unconditional jump
JZ addr	jump if zero
JNZ addr	jump if not zero
JM addr	jump if minus
JP addr	jump if plus
JC addr	jump if carry
JNC addr	jump if no carry
JPE addr	jump if parity ~~odd~~ even
JPO addr	jump if parity odd.

6. ADDRESSING MODES.

6.1 IMPLIED (INHERENT) ADDRESSING.

The instruction itself implies the source and destination of the data; used for operations internal to the processor. These are thus one word instructions. eg (8085)

CLC Clear carry bit (PSW implied)
MOV A, B (using internal registers)

6.2 DIRECT (ABSOLUTE) ADDRESSING.

The actual memory address is specified. (long & slow). Some microprocessors allow a direct page register used to specify the most significant word of the address, with the instruction providing the LSW (useful for eg sequential access) eg LDA 2000

6.3 IMMEDIATE ADDRESSING

The data is given explicitly Eg: ADI 20 $(A) \leftarrow (A) + 20_{16}$
LXI H, 0256 $(H) \leftarrow 02$ $(L) \leftarrow 56_{16}$ or $(HL) \leftarrow 256_{16}$
MVI C, 4A $(C) \leftarrow 4A_{16}$
CPI 02 compare (A) with 2.

6.4 INDIRECT ADDRESSING

We specify the location of the address of the data. Two classes

6.4.1 REGISTER INDIRECT

Address of data contained in a register or register set.

eg MOV A, m (A) ← ((HL))
ADD m (A) ← (A) + ((HL))

6.4.2 MEMORY INDIRECT (not supported by 8085)

Address of memory cells containing address is given. Very slow but powerful.

6.5 INDEXED ADDRESSING. (not supported by 8085)

Address of data held in the index register, to which is added the contents of the offset register (2's comp offset)
eg (6809) LDA D, X (A) ← ((D) + (X))
Older microprocessors only allow constant offsets
eg (6800) LDA 3, X (A) ← ((X) + 3)

6.6 AUTO INCREMENT / DECREMENT INDEXED ADDRESSING.

This is supported by the 8085 only with the stack pointer SP.

6.7 RELATIVE ADDRESSING

These are usually PC-relative 2's complement offsets, most typically used by branch instructions. Not supported by 8085.

7 STACK STRUCTURE AND SUBROUTINES.

7.1 THE STACK.

This is an area of memory set aside by the programmer for temporary storage. The address of the top item of the stack is kept in a special register, the Stack Pointer. These usually use post-increment (pop) / pre-decrement (push) addressing, where the inc/dec'ing is done automatically.
The 8085 always pushes / pops 16 bits at a time,
eg: PUSH H pushes the HL register pair onto the stack.
POP D pops the DE register pair off the stack.

7.2 SUBROUTINES.

Subroutine calls in 8085 are done with the CALL operation. When a call instruction is executed, the following sequence of events takes place:

- the address of the instruction following the CALL (ie contents of PC) are pushed onto stack.
- the PC ← subroutine address

When a RET instruction is executed, PC ← (SP)+

Stacking other data during subroutines must be done with care, to prevent an attempt to RET to the wrong address.
The 8085 also supports conditional subroutine calls, although their use is discouraged.

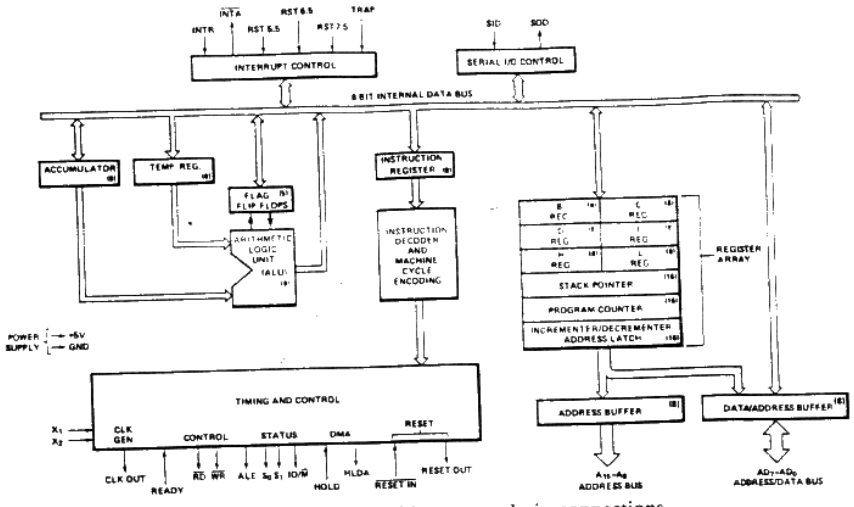


Fig. A4.5 8085A architecture and pin connections.

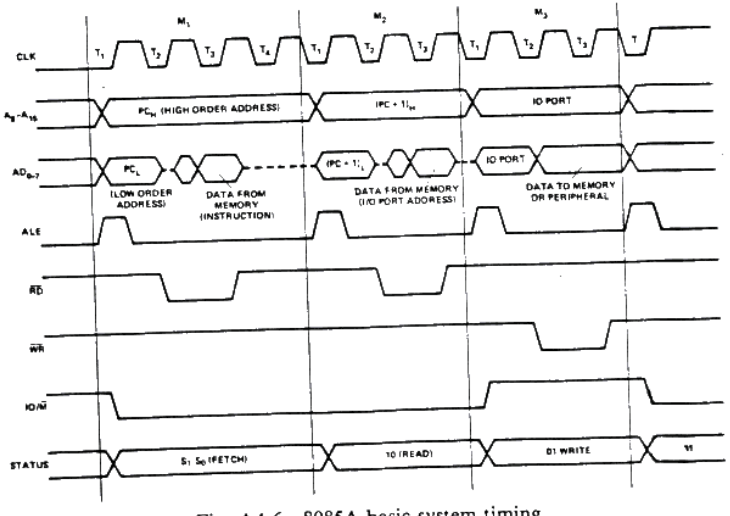


Fig. A4.6 8085A basic system timing.

Mnemonic	Description	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock(2) Cycles
		Instruction Code(1)								
MOVE, LOAD, AND STORE										
MOV r2	Move register to register	0	1	D	D	D	S	S	S	4
MOV M r	Move register to memory	0	1	1	1	0	S	S	S	7
MOV r M	Move memory to register	0	1	D	D	D	1	1	0	7
MVI r	Move immediate register	0	0	D	D	D	1	1	0	7
MVI M	Move immediate memory	0	0	1	1	0	1	1	0	10
LXI B	Load immediate register Pair B & C	0	0	0	0	0	0	0	1	10
LXI D	Load immediate register Pair D & E	0	0	0	1	0	0	0	1	10
LXI H	Load immediate register Pair H & L	0	0	1	0	0	0	0	1	10
LXI SP	Load immediate stack pointer	0	0	1	1	0	0	0	1	10
STAX B	Store A indirect	0	0	0	0	0	0	1	0	7
STAX D	Store A indirect	0	0	0	1	0	0	1	0	7
LDAX B	Load A indirect	0	0	0	0	1	0	1	0	7
LDAX D	Load A indirect	0	0	0	1	1	0	1	0	7
STA	Store A direct	0	0	1	1	0	0	1	0	13
LDA	Load A direct	0	0	1	1	1	0	1	0	13
SHLD	Store H & L direct	0	0	1	0	0	0	1	0	16
LHLD	Load H & L direct	0	0	1	0	1	0	1	0	16
XCHG	Exchange D & E, H & L Registers	1	1	1	0	1	0	1	1	4
STACK OPS										
PUSH B	Push register Pair B & C on stack	1	1	0	0	0	1	0	1	12
PUSH D	Push register Pair D & E on stack	1	1	0	1	0	1	0	1	12
PUSH H	Push register Pair H & L on stack	1	1	1	0	0	1	0	1	12
PUSH PSW	Push A and Flags on stack	1	1	1	1	0	1	0	1	12
POP B	Pop register Pair B & C off stack	1	1	0	0	0	0	0	1	10
POP D	Pop register Pair D & E off stack	1	1	0	1	0	0	0	1	10
POP H	Pop register Pair H & L off stack	1	1	1	0	0	0	0	1	10
POP PSW	Pop A and Flags off stack	1	1	1	1	0	0	0	1	10
XTHL	Exchange top of stack, H & L	1	1	1	0	0	0	1	1	16
SPHL	H & L to stack pointer	1	1	1	1	1	0	0	1	6
JUMP										
JMP	Jump unconditional	1	1	0	0	0	0	1	1	10
JC	Jump on carry	1	1	0	1	1	0	1	0	7/10
JNC	Jump on no carry	1	1	0	1	0	0	1	0	7/10
JZ	Jump on zero	1	1	0	0	1	0	1	0	7/10
JNZ	Jump on no zero	1	1	0	0	0	0	1	0	7/10
JP	Jump on positive	1	1	1	1	0	0	1	0	7/10
JM	Jump on minus	1	1	1	1	1	0	1	0	7/10
JPE	Jump on parity even	1	1	1	0	1	0	1	0	7/10
JPO	Jump on parity odd	1	1	1	0	0	0	1	0	7/10
PCHL	H & L to program counter	1	1	1	0	1	0	0	1	6
CALL										
CALL	Call unconditional	1	1	0	0	1	1	0	1	18
CC	Call on carry	1	1	0	1	1	1	0	0	9/18
CNC	Call on no carry	1	1	0	1	0	1	0	0	9/18
CZ	Call on zero	1	1	0	0	1	1	0	0	9/18
CNZ	Call on no zero	1	1	0	0	0	1	0	0	9/18
CP	Call on positive	1	1	1	1	0	1	0	0	9/18
CM	Call on minus	1	1	1	1	1	1	0	0	9/18
CPE	Call on parity even	1	1	1	0	1	1	0	0	9/18
CPO	Call on parity odd	1	1	1	0	0	1	0	0	9/18
RETURN										
RET	Return	1	1	0	0	1	0	0	1	10
RC	Return on carry	1	1	0	1	1	0	0	0	6/12
RNC	Return on no carry	1	1	0	1	0	0	0	0	6/12
RZ	Return on zero	1	1	0	0	1	0	0	0	6/12
RNZ	Return on no zero	1	1	0	0	0	0	0	0	6/12
RP	Return on positive	1	1	1	1	0	0	0	0	6/12
RM	Return on minus	1	1	1	1	1	0	0	0	6/12
RPE	Return on parity even	1	1	1	0	1	0	0	0	6/12
RPO	Return on parity odd	1	1	1	0	0	0	0	0	6/12
RESTART										
RST	Restart	1	1	A	A	A	1	1	1	12
INPUT/OUTPUT										
IN	Input	1	1	0	1	1	0	1	1	10
OUT	Output	1	1	0	1	0	0	1	1	10
INCREMENT AND DECREMENT										
INR r	Increment register	0	0	0	0	0	1	0	0	4
DCR r	Decrement register	0	0	0	0	0	1	0	1	4
INR M	Increment memory	0	0	1	1	0	1	0	0	10
DCR M	Decrement memory	0	0	1	1	0	1	0	1	10
INX B	Increment B & C registers	0	0	0	0	0	0	1	1	6
INX D	Increment D & E registers	0	0	0	1	0	0	1	1	6
INX H	Increment H & L registers	0	0	1	0	0	0	1	1	6
INX SP	Increment stack pointer	0	0	1	1	0	0	1	1	6
DCX B	Decrement B & C	0	0	0	0	1	0	1	1	6
DCX D	Decrement D & E	0	0	0	1	1	0	1	1	6
DCX H	Decrement H & L	0	0	1	0	1	0	1	1	6
DCX SP	Decrement stack pointer	0	0	1	1	1	0	1	1	6
ADD										
ADD r	Add register to A	1	0	0	0	0	S	S	S	4
ADC r	Add register to A with carry	1	0	0	0	1	S	S	S	4
ADD M	Add memory to A	1	0	0	0	0	1	1	0	7
ADC M	Add memory to A with carry	1	0	0	0	1	1	1	0	7
ADI	Add immediate to A	1	1	0	0	0	1	1	0	7
ACI	Add immediate to A with carry	1	1	0	0	1	1	1	0	7
DAD B	Add B & C to H & L	0	0	0	0	1	0	0	1	10
DAD D	Add D & E to H & L	0	0	0	1	1	0	0	1	10
DAD H	Add H & L to H & L	0	0	1	0	1	0	0	1	10
DAD SP	Add stack pointer to H & L	0	0	1	1	1	0	0	1	10
SUBTRACT										
SUB r	Subtract register from A	1	0	0	1	0	S	S	S	4
SBB r	Subtract register from A with borrow	1	0	0	1	1	S	S	S	4
SUB M	Subtract memory from A	1	0	0	1	0	1	1	0	7
SBB M	Subtract memory from A with borrow	1	0	0	1	1	1	1	0	7
SUI	Subtract immediate from A	1	1	0	1	0	1	1	0	7
SBI	Subtract immediate from A with borrow	1	1	0	1	1	1	1	0	7
LOGICAL										
ANA r	And register with A	1	0	1	0	0	S	S	S	4
XRA r	Exclusive Or register with A	1	0	1	0	1	S	S	S	4
ORA r	Or register with A	1	0	1	1	0	S	S	S	4
CMP r	Compare register with A	1	0	1	1	1	S	S	S	4
ANA M	And memory with A	1	0	1	0	0	1	1	0	7
XRA M	Exclusive Or memory with A	1	0	1	0	1	1	1	0	7
ORA M	Or memory with A	1	0	1	1	0	1	1	0	7
CMP M	Compare memory with A	1	0	1	1	1	1	1	0	7
ANI	And immediate with A	1	1	1	0	0	1	1	0	7
XRI	Exclusive Or immediate with A	1	1	1	0	1	1	1	0	7
ORI	Or immediate with A	1	1	1	1	0	1	1	0	7
CPI	Compare immediate with A	1	1	1	1	1	1	1	0	7
ROTATE										
RLC	Rotate A left	0	0	0	0	0	1	1	1	4
RRC	Rotate A right	0	0	0	0	1	1	1	1	4
RAL	Rotate A left through carry	0	0	0	1	0	1	1	1	4
RAR	Rotate A right through carry	0	0	0	1	1	1	1	1	4
SPECIALS										
CMA	Complement A	0	0	1	0	1	1	1	1	4
STC	Set carry	0	0	1	1	0	1	1	1	4
CMC	Complement carry	0	0	1	1	1	1	1	1	4
DAA	Decimal adjust A	0	0	1	0	0	1	1	1	4
CONTROL										
EI	Enable interrupts	1	1	1	1	1	0	1	1	4
DI	Disable interrupt	1	1	1	1	0	0	1	1	4
NOP	No-operation	0	0	0	0	0	0	0	0	4
HLT	Halt	0	1	1	1	0	1	1	0	5
NEW 8085A INSTRUCTIONS										
RIM	Read Interrupt Mask	0	0	1	0	0	0	0	0	4
SIM	Set Interrupt Mask	0	0	1	1	0	0	0	0	4

NOTES 1 DDD or SSS B 000 C 001 D 010 E 011 H 100 L 101 Memory 110 A 111
2 Two possible cycle times (6/12) indicate instruction cycles dependent on condition flags

9. ASSEMBLY LANGUAGE PROGRAMMING TECHNIQUES.

In order to minimise software costs (and these are far greater than hardware costs), we need programming techniques to increase programmer efficiency and program reliability.

9.2 PROGRAM DEVELOPMENT TOOLS - THE ASSEMBLER.

The assembler is a program to convert our mnemonic-form programs into m/c. The assembler also provides some additional capabilities discussed below

9.2.1 THE USE OF LABELS.

Labels can be used to keep track of addresses. In the 8085 assembler, labels may be up to 6 characters long.

```
eg      LOOP:   MOV  A, M
```

9.2.2 THE USE OF SYMBOLS.

Labels can also be used as symbols for numbers, using the EQU directive. Labels and symbols make the program easier to read and thus easier to understand and maintain.

```
eg      MAX     EQU  20
        LOOP    EQU  $
                MOV  A, M
```

The last two lines are equivalent to the label example; $ is the predefined symbol representing the contents of the assembler's location counter.

9.2.3 THE USE OF EXPRESSIONS.

At any place where a number is required, an expression may be used and the assembler will evaluate the expression and use the result

```
eg   START   EQU   3000
     MAXELT  EQU   50
     ELTSIZ  EQU   10
     END     EQU   START + MAXELT * ELTSIZ
```

9.2.4 ASSEMBLER DIRECTIVES.

Some of the most commonly used directives are

EQU - EQUATE <symbol> EQU <expr>

Associates the expression value on the RHS with the symbol on the LHS

ORG - ORIGIN ORG <expr>

Sets the location counter to the specified value

END

Marks the end of the program

DB - DECLARE BYTES [<symbol>] DB <expr list>

Put specified data bytes into memory

eg DB 10, 20, 30

DW - DECLARE WORDS

Put specified words (16-bit) into memory, with the least significant byte at the lower addresses.

DS - DECLARE STORE [<symbol>] DS <exp>

This reserves an area of memory by adding the expression value to the location counter.

9.3 NOTES ON ASSEMBLY LANGUAGE PROGRAMMING

As assembly programs are usually long and obscure, the programmer must make a special effort to ensure readability. Broadly speaking, the legibility and reliability of a program will be greatly increased if attention is paid to the following aspects:

- program modularity and structure
- use of symbols
- layout

9.3.1 MODULARITY

Programs should be hierarchically designed as a series of tasks and subtasks, with the fundamental (simplest) subtasks being simple subroutines. The program can thus be considered as a tree, with each node having direct access only to its immediate children.

GENERAL RULES REGARDING SUBROUTINES.

- the length of a subroutine is obviously dependent on the function it performs. Assembly language S/R's should be between about 5 & 50 instructions in length.
- subroutines should always end with a RET (don't use 8085 conditional returns)
- subroutines should have only one entrypoint, the first instruction
- subroutine calls may be nested.

9.3.2 THE USE OF SYMBOLS.

Symbol names should be meaningful. Important program constants should always be represented by symbols. An assembler program should also never specify any addresses explicitly. Binary and hex base capabilities are also useful at times.

9.3.3 LAYOUT

COMMENTS

A comment must convey, as concisely as possible, the functions of the instruction that are not immediately apparent, and must not carry any redundant information.

SYMBOLS

The choice of symbol name should convey as much information about its function as possible, within the constraints of having to have the first 6 characters unique. Names such as LOOP1, LOOP2, CKMOFG, etc. should be avoided.

HEADINGS

Each subroutine must have a heading giving its name, expected inputs / outputs, register preservation / destruction status, names of other subroutines called, and a brief functional description. The program as a whole should have a heading indicating its function (by name), name, programmers name, date, and a brief description, as well as a list of any special hardware requirements.

ORDERING OF INFORMATION

The program as a whole should be laid out as follows:

1 - the program header

2 - the equated symbols used, listed in order of logical groups (and alphabetically within those).

3 - the main program

4 - the major subroutines

5 - the remaining subroutines in alphabetical order

6 - the variables again placed in logical groups

7 - the program's stack space. The first instruction of the main program must initialise the stack pointer to point to the top of this stack space.

10 MEMORY DEVICES

10.1 TERMINOLOGY

Bits are grouped into words, which have unique addresses, encoded as binary patterns.
Memory organisation is expressed by (number of words) x (word length)
eg: a 16 K bit device could be organised as 2 K x 8.

10.2 TYPES OF MEMORY

Memory devices can be categorised in a number of ways

10.2.1 TECHNOLOGY - two major types

(a) BIPOLAR - manufactured using normal transistors; very fast, high power consumption, low density devices.

(b) MOS - manufactured using MOSFETS; low power, high density; slower than BIPOLAR but this is changing.

10.2.2 ACCESS CAPABILITIES

(a) READ/WRITE (RAM, RWM)

(b) READ ONLY (ROM)

(c) PROGRAMMABLE ROMS

10.2.3 VOLATILITY

(a) NON-VOLATILE (ROMs & PROMs) retain their contents if power is removed.

(b) VOLATILE (RAM) lose contents if power is removed.

(i) STATIC (BIPOLAR OR MOS) Retain their data as long as power is maintained.

(ii) DYNAMIC (MOS only) Retain their data for a short period (typically 2msec); must be periodically refreshed.

10.2.4 COMMON MEMORY DEVICES

CODE	ORGANISATION	TYPE	SPEED
7489	16×4	BIPOLAR STATIC RAM	35 ns
74S287	256×4	BIPOLAR PROM	50 ns
6116	2K×8	CMOS STATIC RAM	150 ns
2764	8K×8	MOS EPROM	200 ns
6264	8K×8	CMOS STATIC RAM	150 ns
4164	16K×1	MOS DYNAMIC RAM	200 ns
41256	256K×1	MOS DYNAMIC RAM	150 ns.

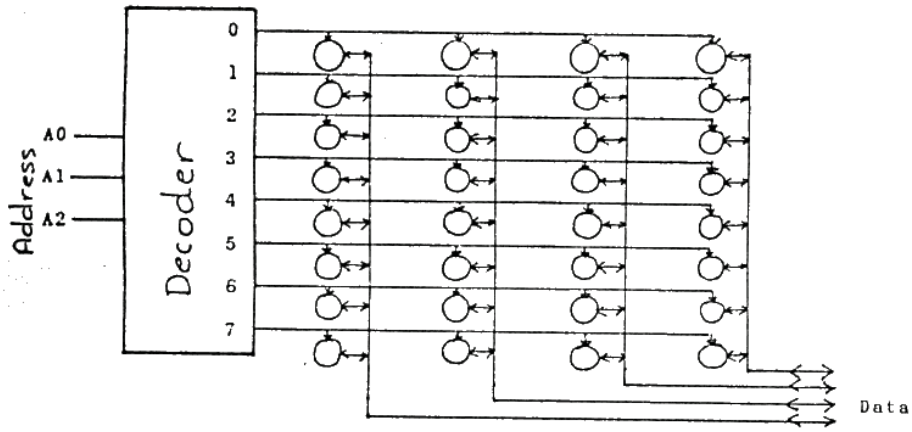
10.3 MEMORY CONSTRUCTION.

A memory device is constructed from a large number of storage cells, each capable of holding one bit of data, connected in such a way that any cell or group of cells may be read or written individually. We represent these cells as 

and assume that a cell can only be read or written to if the enable input is high.

A simple memory device:

We will use as an example a hypothetical 8 x 4 memory device, i.e. 8 words of 4 bits each. It is clear that such a device would have 3 address lines ($2^3=8$). In addition, the memory device has four data lines: when a particular word's address is placed on the three address lines, the contents of that word must appear on these four data lines (for a read access), or the data on the data lines must be written into the addressed word (for a write operation). The logical connection for this device would be:



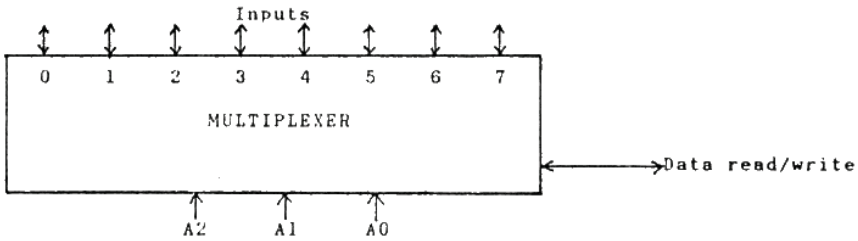
The cells are connected in an 8 x 4 matrix. (We assume that the storage cells do not affect the data read/write signals if they are not enabled.) Each of the rows of 4 cells forms one word, and all the cells in one word are enabled by a common signal. The task of converting the binary encoded address on the three address lines to eight separate mutually-exclusive row-select lines is performed by a decoder. The truth table for this particular decoder is:

Inputs			Outputs							
A2	A1	A0	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Higher density memory devices:

The simple memory matrix scheme outlined above is not practical when higher-density devices are considered. A 1K x 4 memory device would require a decoder capable of converting 10 encoded address inputs (2^{10} = 1K) into 1024 separate row select signals. Clearly the logic required to perform this decoding would be prohibitively complex.

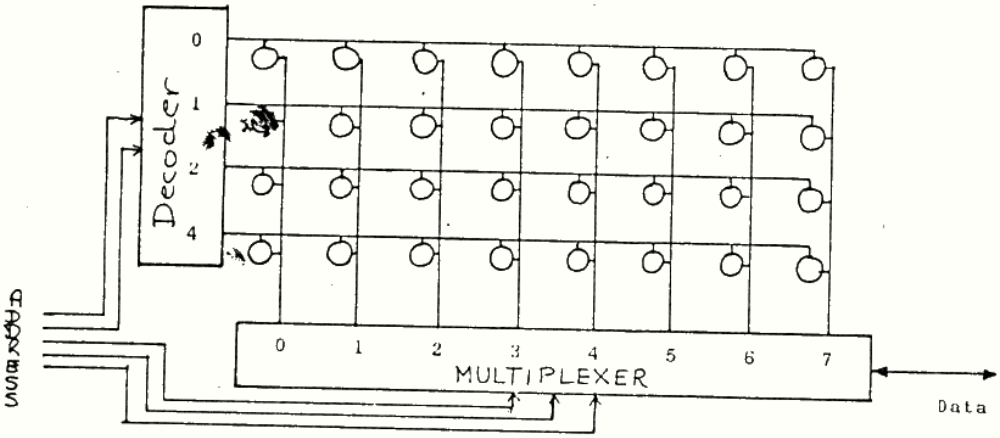
In order to produce the decoding requirements for these memory devices, therefore, a second type of device is used in addition to the decoder. This device is called the multiplexer:



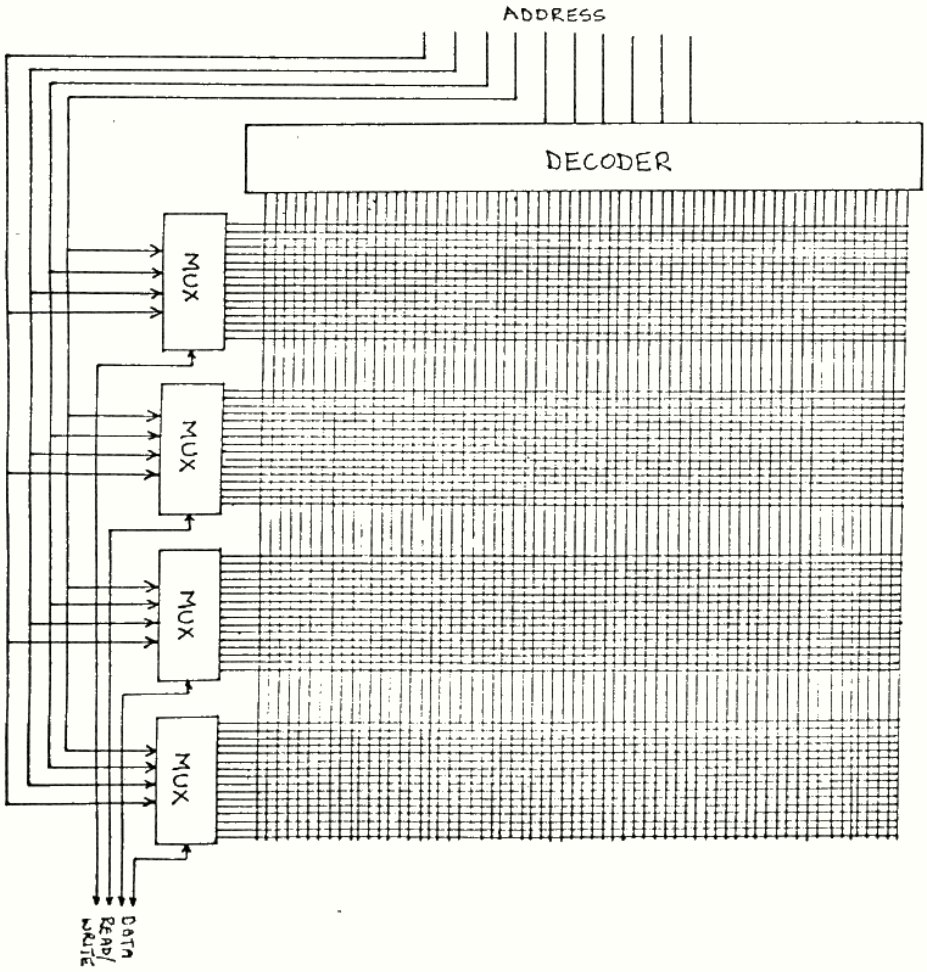
The multiplexer is simply an electronic switch which connects any one of the inputs to the single data read/write output. The input which is connected is selected by the address lines:

A2	A1	A0	Result
0	0	0	Input number 0 is connected to the output
0	0	1	Input number 1 is connected to the output
0	1	0	Input number 2 is connected to the output
0	1	1	Input number 3 is connected to the output
1	0	0	Input number 4 is connected to the output
1	0	1	Input number 5 is connected to the output
1	1	0	Input number 6 is connected to the output
1	1	1	Input number 7 is connected to the output

Using the multiplexer, the decoding requirements can be considerably reduced. Using a 32 x 1 device as an example (32 words => 5 address inputs), the cell connection matrix (which would have to be 32 rows of 1 cell each using the simple decoding scheme illustrated earlier) would now become.



Each address selects an entire row of 8 cells; however, the multiplexer only connects the read/write output of one of the eight to the device's read/write data line. Thus 32 x1 organisation is achieved from a 4 x 8 matrix, there will be considerable extra logic to control the operation of reading and writing to the cells. The block diagram of a 1K x 4 memory device is shown on the next page; note that the use of one 6-to-64 decoder and four 16 input multiplexers reduces the cell matrix to a square 64 x 64 bit matrix.



10.4 Types of storage cells:

We now look in some detail at different types of storage cells used in semiconductor memory devices.

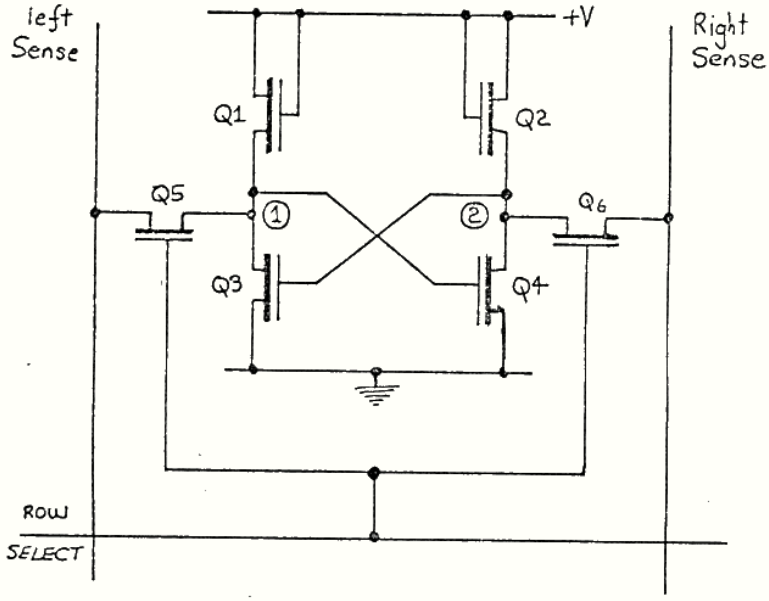
10.4.1 Read/Write (RAM) storage cells

These are subdivided into two different types; the static and dynamic read/write cells.

a. The Static memory cell:

As mentioned earlier, all bipolar memory devices and some MOS memory devices use static techniques. The following description uses the MOS memory cell; however, the bipolar static memory cell is based on exactly the same principles.

The circuit diagram of the MOS static memory cell is as follows:

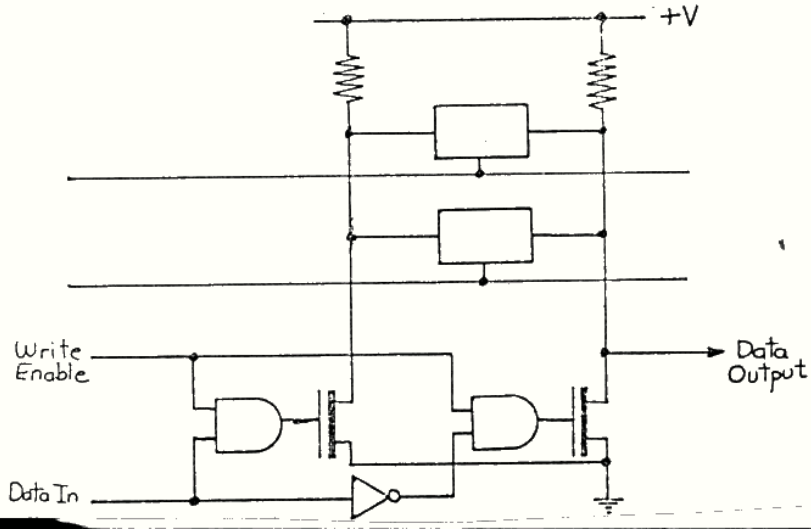


The storage cell is formed by transistors Q_1 to Q_4. Q_1 and Q_2 are permanently turned on and act as load resistors for Q_3 and Q_4. These transistors are cross-coupled to form a bistable, which can exist in one of two stable states. This is therefore the memory element. In the one state, Q_3 is on and Q_4 is off. Node 1 is therefore low and node 2 is high, and a '1' is considered to be stored in the cell. In the other state, Q_3 is off and Q_4 is on. Node 1 is therefore high and node 2 is low, and a '0' is considered to be stored in the cell. Obviously the circuit will remain in either of these states indefinitely; hence the static nature of the cell.

In order to provide a means of reading the state of the cell (i.e. the stored bit value), Q_5 and Q_6 are provided. These transistors connect nodes 1 and 2 to the sense lines, which are connected similarly to all the other cells in the column. Q_5 and Q_6 are turned on the by row-select line: when the row select line is high, therefore, node 1 and 2 are connected to the sense lines and the state of the cell may be read by observing the state of the sense lines.

A similar process is followed to write to the cell. The row-select lines is again raised to turn Q_5 and Q_6 on and connect node 1 and 2 to the sense lines. Now, however, instead of passively observing the state of these lines (as is done when reading the cell) the sense lines are forced into the desired state. Nodes 1 and 2 are therefore forcibly driven to the desired condition, and Q_3 and Q_4 are turned on or off accordingly. The cell is now in the desired state; the row-select line may now be lowered again and the new state will be retained.

The circuitry to achieve this reading and writing operation is shown in simplified form below. Notice that in practice it is only necessary to forcibly drive one sense line low when writing; the other sense line may be passively pulled up.



Note that the column select lines have been omitted for clarity.

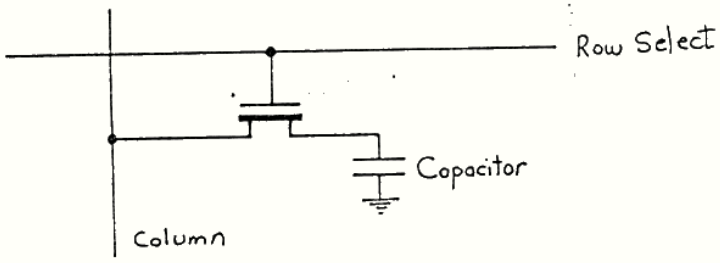
b) The MOS Dynamic memory cell:

The basic storage element used in a dynamic memory cell is a capacitor. The logic level is remembered as follows:

Charged capacitor = '1' stored.
Discharged capacitor = '0' stored.

The problem with this type of storage is that the charge on a capacitor will leak away and to a lesser extent, a discharged capacitor will charge. This means that for only a short time after a logic level, has been written into a dynamic cell, will it remain valid. If this type of cell is to be used as RWM then the capacitors must be refreshed regularly. Refreshing simply means that the capacitor is returned to its original completely charged or discharged state. Normally the time between refreshes is 2ms.

The basic dynamic cell is illustrated below:



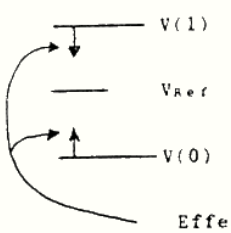
Writing to the cell

This is a simple process. The capacitor is simply charged or discharged. As the cell is being rewritten, there is no need to determine the current contents of the cell.

Reading from the cell

The operation of reading from the cell is far more complex. The storage capacitor C_s, which was initially charged to V(1) (if a '1' was to be stored) or V(0) (if a '0' was to be stored), may have, due to leakage currents, drifted from its original value.

The sensing/refreshing of the storage cell (i.e. the capacitor) is done using a sense amplifier. In the sense amplifier a third voltage level, one between V(1) and V(0), is used. These three voltages are illustrated below:



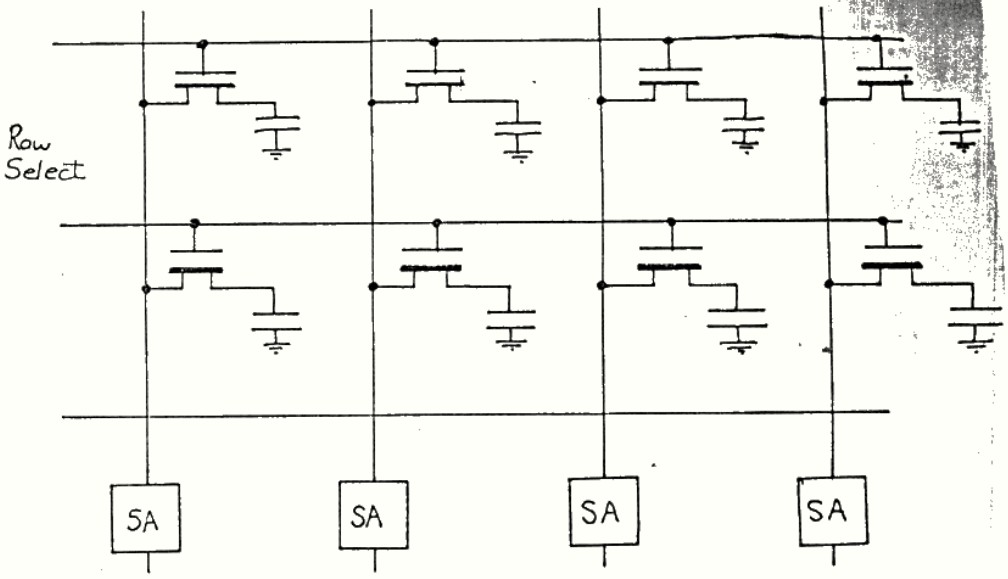
V(1) — Voltage on capacitor if a '1' is stored

V_{Ref} — Normally [V(1) + V(0)] / 2

V(0) — Voltage on capacitor if a '0' is stored

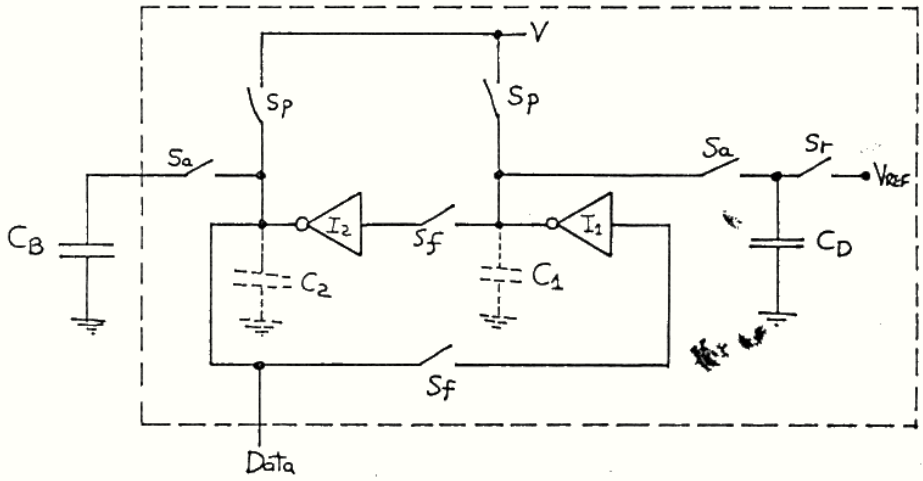
Note: The dynamic RAM cell uses only one MOS transistor.

As before the cells are arranged in a matrix. The difference is that each column line is terminated with a sense amplifier.



SA = Sense amplifier

The circuit diagram for the sense amplifier is given below:



Sense amplifier read operation:

Step 1: Equalisation (precharge C_1 and C_2 to the same voltage)

Close S_P switches.

Step 2: Charge C_D with V_{ref}.

Close S_r
Open S_r

Step 3:

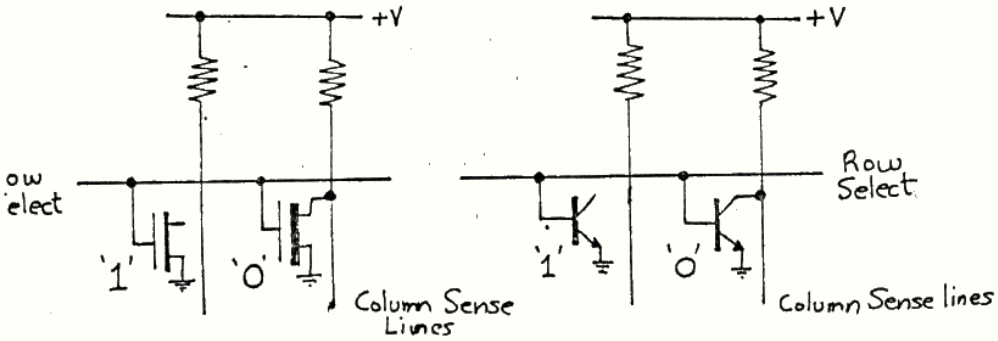
Open S_P
Close S_a

Step 4:

Close S_f

10.4.2 Read-only storage cells

Read-only storage cells are much simpler than read/write cells because there is no need for the circuitry to support a writing operation. In fact, all ROM storage cells consist of a single transistors, as follows:



MOS ROM cells Bipolar ROM cells

The various types of ROM and programmable ROM (PROM) differ only in the way in which the transistors are manufactured and connected.

a) Mask-programmed ROMS:

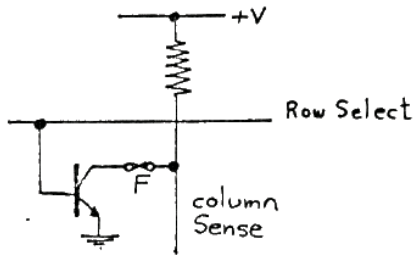
These are generally MOS devices. The matrix of transistors is laid out on the silicon wafer, and the row and column lines are laid on the surface. The actual aluminium tracks connecting the MOSFETs to the column sense lines, however, are left until last. The information that is to be written into the ROM is supplied by the customer to the semiconductor manufacturer. When the connections between the transistors and the column sense lines are laid down, the appropriate ones are omitted so that some of the transistors remain unconnected. The chips are then packaged in the normal way

Because a new mask must be made for every new set of data to be programmed into the ROM, the initial costs of the mask-programmed devices are very high indeed. Thereafter the manufacturing process is completely standard, however, so the costs for very large numbers of mask-programmed ROMs become lower than those of any other form of ROM device.

b) Fusible-link PROMs:

These are ROM devices which may be programmed with information by the user after they have been manufactured. The transistors within the devices are connected to the column sense lines by silicon fuses on the surface of the chip:

The device is programmed by raising the supply voltage to a higher than normal level and turning the transistor on. The resulting high current through the transistor blows the fuse, thereby disconnecting the transistor and writing a '1' into the cell.



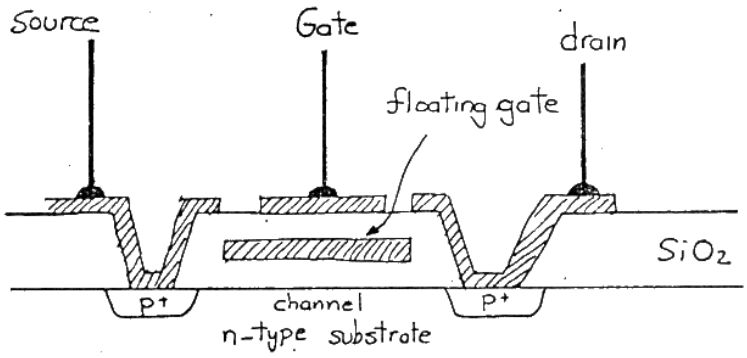
Fusible-link ROM cell

This operation can obviously only be performed once.

c) Erasable PROMs (EPROMs of UVEPSOMs):

When developing software, the higher initial cost of mask-programmed ROMs and the once-only nature of fusible-link PROMs make them unsuitable for development work. For these purposes, the ultra-violet erasable PROM is used.

The transistors in an EPROM are always connected to the column sense lines: the information is stored by determining whether the transistor will turn on or not when the row select line is raised. This sort of transistor is obviously not a normal transistor; its physical structure is shown below:



FAMOS transistor

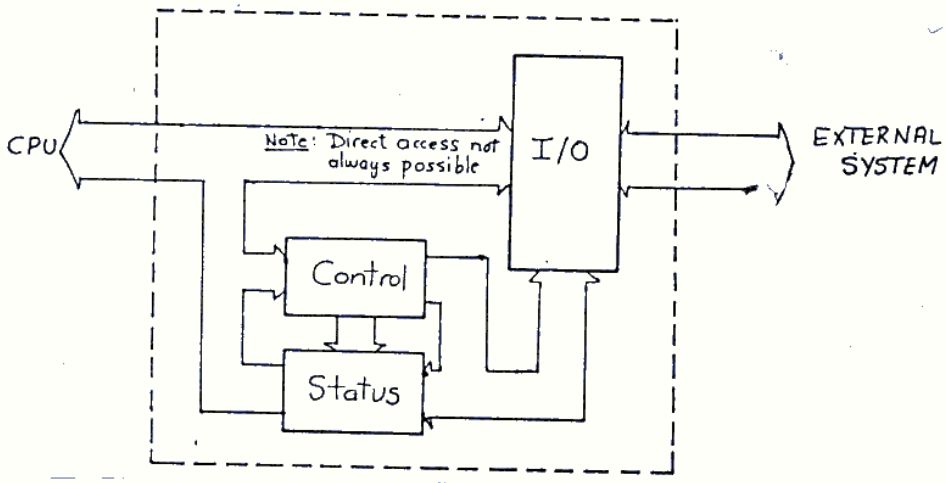
This is called a "Floating-gate Avalanche-injection MOS" transistor (FAMOS). It is just like a normal MOSFET except that it has an additional "floating gate", completely isolated on all sides by silicon dioxide. Normally the floating gate has no charge; the FAMOS transistor then behaves just like a normal MOSFET (i.e. a '0' is stored in the cell formed by the FAMOS transistor). In order to program a '1' into the cell (i.e. effectively disconnect the transistor), a very large voltage (approximately 21 volts, but it differs for different types of EPROMS. The common voltages are: 25V, 21V and 12.5) is applied to the channel and a positive voltage applied to the gate. The resulting avalanche effect causes high-energy electrons attracted by the gate to overcome the insulating layer and move to the floating gate. When the high voltage is removed, the electrons do not have sufficient energy to return across the insulator and are trapped on the floating gate, which thus acquires an overall negative charge. This charge will be retained almost indefinitely because of the excellent insulating properties of silicon dioxide.

Because of the negatively charged floating gate, the application of positive voltage to the normal gate can no longer turn the transistor on and a '1' has therefore been programmed into the cell.

In order to erase the cell (i.e. remove the trapped charge on the floating gate) the device is exposed to high-intensity ultraviolet light. The resulting photo-electric effect drives off the electrons from the gate and restores it to its original uncharged state.

11 PROGRAMMED I/O

The CPU must be able to communicate with peripheral devices. The peripherals may be controlled by the CPU directly, or may function independently and perform the I/O transfers autonomously.



PERIPHERAL BLOCK DIAGRAM

A peripheral I/O device has three major sections:

- Input / Output - this is the section which actually connects the I/O pins of the device. Depending upon the complexity of the peripheral device, its function may range from very simple (eg buffers or latches) to very complex (in which case the CPU seldom has any direct access to the I/O section itself).
- Control - this contains the control logic for the I/O section, and responds to instructions from the CPU.
- Status - this contains the logic used by the microprocessor to determine the current state of the peripheral device.

Broadly speaking, all peripherals may be categorised into one of the following four classes:

- Simple device - no control or status, only simple I/O section directly controlled by the CPU (eg tri-state buffer)

- <u>Basic device</u> - these are devices having a control section, but no status section since they cannot perform any complex operations themselves. The control section merely modifies the operation of the I/O section in some way (eg converts it from input to output, or causes it to latch data instead of just passing it through). The CPU must still have direct access to the I/O section, since it performs most of the processing and I/O operations (eg 8255A parallel peripheral interface).
- <u>Moderately complex device</u> - has all three sections; I/O section can handle some operations by itself, and the CPU thus has only limited access to it (eg 8251A serial interface).
- <u>Extremely complex devices</u> - the control section is as complex as the μP itself; they can perform very complex I/O operations by themselves, and the CPU need only issue commands to cause these to be performed. The CPU has no direct access to the I/O section (eg 2793 FDC)

When the operation of the peripheral device is controlled by the CPU, the I/O information flow is called a programmed I/O transfer. Such an I/O is performed by the CPU using (8085 IN/OUT) specialised I/O instructions. These instructions may be for:

- sending commands to devices
- receiving the device status
- I/O of data to the device.

Two simple techniques for performing programmed I/O are discussed below.

11.1 UNCONDITIONAL TRANSFER.

In this method, the status of the peripheral device is not tested, but the data is simply input or output.

Eg:

```
DATAOUT:  MOV  A, M       ; Fetch the next byte to be output
          OUT  PORTA      ; Output the byte
          INX  H          ; Increment memory pointer
          DCR  B          ; Decrement loop counter
          JNZ  DATAOUT    ; Repeat till finished.
```

11.2 CONDITIONAL TRANSFER (also called POLLED I/O TRANSFER)

Conditional transfer is used when the peripheral requires a relatively large amount of time to perform the given I/O operation. The CPU must first determine the device status, and wait until the device is ready, repeatedly checking the status (polling)

Eg:

```
NXTBYT:  IN   STATUS
         ANI  00000001B
         JNZ  NXTBYT
         MOV  A, M
         OUT  PORTA
         INX  H
         DCR  B
         JNZ  NXTBYT
```

The time spent waiting for the device to be ready can often be put to good use performing other tasks.

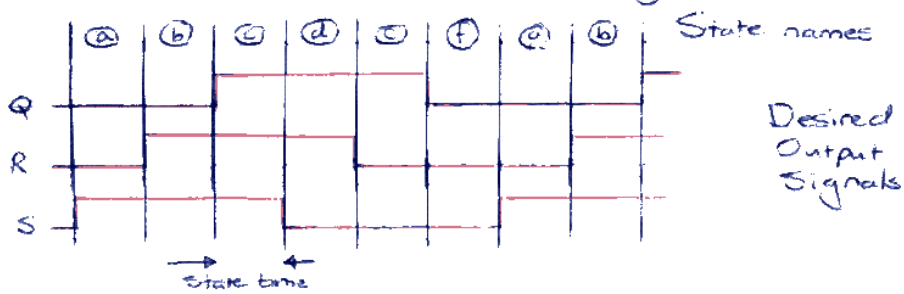
12 ALGORITHMIC STATE MACHINES

12.1 STATE MACHINE DESCRIPTION

A state machine is a generalised term for a sequential circuit which can be exist only in a finite (defined) number of states, and whose progression from one state to the next is clearly defined at all times. A simple example is a binary counter; typically however, we allow inputs which can modify the behaviour of the machine

12.2 THE DESIGN OF SIMPLE STATE MACHINES

Eg a three-phase square wave generator



What is meant by a state?

- a circuit is said to be in a certain state while it remains in a fixed, stable and unique condition. The circuit remains in the condition for the state time during which time the outputs from the circuit are stable. At the end of the state time, the circuit performs a transition to another state or it may remain in the same state. The transition between states is assumed to be instantaneous.

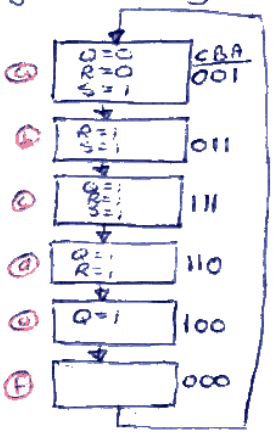
NB It is impossible to establish the state of a circuit by observing the condition of the output signals.

We now represent our states in the form of a state diagram: each state of the circuit is represented by a state box and the transitions

between the states by path lines.

It is common to only list the outputs that are true in the output list. Each state box must have only one exit path, which must lead unambiguously to a single state box.

Our example is:



LOGIC IMPLEMENTATION

We can implement using flip-flops. The binary numbers on the RHS of the state boxes indicates the internal state variables used. Simplest using D-FF's. Steps involved:

1. Draw up a table indicating all possible state variable values, and output variables. Simplify using Karnaugh maps etc. to determine relationships between output variables and state variables (in our eg Q = C, R = B, S = A)
2. Draw up a current state / next state table to determine the circuitry attached to the D-inputs