ECED 3204. Microprocessors

TRANSPARENCIES

Instructor: Jose Gonzalez-Cueto

Department of Electrical & Computer Engineering

Dalhousie University, Halifax

Fall 2013

Course website:

http://myweb.dal.ca/~gonzalej/Teaching/Eced3204/ECED3204.html

(or) **Dept webpage → Faculty → Jose → Teaching → Microprocessors**

Hardware:

1. **Motorola M68HC11EVB board (The evaluation board or EVB)**
   - **Based on the M68HC11 MCU (MicroController Unit)**

Software:

1. **Mini IDE (Integrated Development Environment, Runs on PC)**
   - **M68HC11 Cross Assembler**
   - **Building executable files**
   - **Downloading of 6811 executables to the EVB**
   - **PC ↔ EVB serial communication**

Software (Cont'd):

2. Stan Simmons' QEVB11 Simulator (Runs on PC)

- **Simulates the EVB board in detail**
- **CPU operation, bus timing, serial communication and more...**
- **Downloadable through link in course website**
- **Comes with tutorials. Do tutorials 1-3 over weeks 1-4**
- **Helpful for the Labs, program debugging and testing**

Bibliography

[1] Hughes, L., *Hardware and Software Design for the MC68HC11*, 5th edition, Whale Lake Press, 2004 (textbook) - REQUIRED.

[2] M68HC11 Reference Manual, *a.k.a. The Pink Book* , Motorola.

[3] MC68HC11A8 Programming Reference Guide, Motorola - REQUIRED.

[4] M68HC11EVB Evaluation Board User's Manual, Motorola.

[5] Huang, H., *MC68HC11: An Introduction; Software and Hardware Interfacing* , 2nd edition, Delmar Thomson Learning, 2000 (textbook) (in Library & Bookstore) - REQUIRED.

[6] Gonzalez-Cueto, J.A., *ECED3204 Transparencies*, April 2006 - REQUIRED.

[7] Martin, F., *Introduction to 6811 Programming* , Media Lab, MIT.

[8] Spasov, P., *Microcontroller Technology: The 68HC11*, 4th edition, Prentice Hall, 2002.

[9] Driscoll, Coughlin & Villanucci, *Data Acquisition and Process Control with the M68HC11 Microcontroller*, Merril / Macmillan, 1994 (in Library).

Course Contents

Course Contents (cont'd)

4. Asynchronous serial communication, (Starts on Page 126)
   - Introduction,
     - UART rx & tx units.
     - Registers and character formation/handling in both directions.
     - Serial-to-parallel, parallel-to-serial conversion processes.
     - Errors and error handling.
     - The RS-232 standard.
   - The HC11 and asynchronous serial communication .- the SCI unit,
     - SCI tx unit.
     - SCI rx unit. Error handling.
     - Control/status registers, rx/tx data registers.
     - CPU «--» SCI unit interaction, I/O methods: polling & interrupts.
   - Asynchronous serial communication and the EVB,
     - EVB connectors, EVB serial port connectors.
     - The SCI and ACIA as the EVB UARTs.
     - Use of the D-type flipflop and digital switches.
     - ACIA decode & programming.
     - BUFFALO communication, the RS232 window (SCI terminal) and EVB ports.
     - RS232 drivers & receivers.
   - Interrupts,
     - The HC11 interrupt vector table.
     - Interrupt driven I/O, programming with interrupts.
     - Interrupts and the M68HC11EVB evaluation board.
   - Laboratories,
     (4) Device polling & terminal I/O,
     (5) Asynchronous serial communication and interrupts.

Course Contents (cont'd)

5. HC11 timer system, (Starts at Page 152)
   - Main timer functions
     - Output compare; software timing, waveform generation.
     - Input capture; measuring period & pulse width.
     - Long periods and counter overflows.
     - Algorithms for generating and measuring slow-changing signals.
   - Solving missed output compares and missed overflows. Interrupt priority.
   - Real-time interrupt.
   - Laboratory:
     (6) Timer functions.
6. Parallel I/O communication, (Starts at Page 170)
   - General purpose I/O.
   - HC11 output PortB and bidirectional PortC.
   - Seven segment displays.
     - Hardware issues
       - Using HC11 ports to light 7-segment LED displays.
       - Controlling multiple 7-segment displays.
     - Software issues
       - Light patterns for digits to be displayed. Table lookup.
       - Conversion of 16-bit hex format --» BCD format «--» ASCII-coded decimal format.
   - Strobe and handshake I/O subsystem.
   - Design and service of parallel I/O ports external to the HC11.
   - Laboratory:
     (7) Seven-segment LED displays.
7. Course Review.
8. Final Exam.

Course Assessment

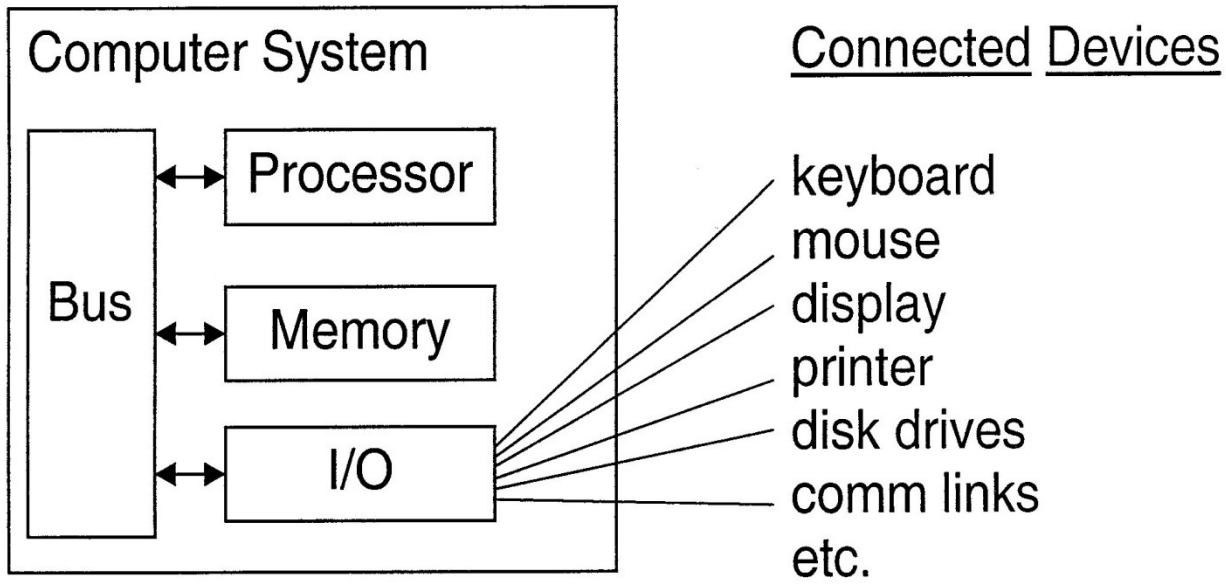
Biweekly Quizzes:	25%
Laboratories:	25%
MidTerm Exam	25%
EndofTerm Exam	25%
Total:	100%

What is a computer?

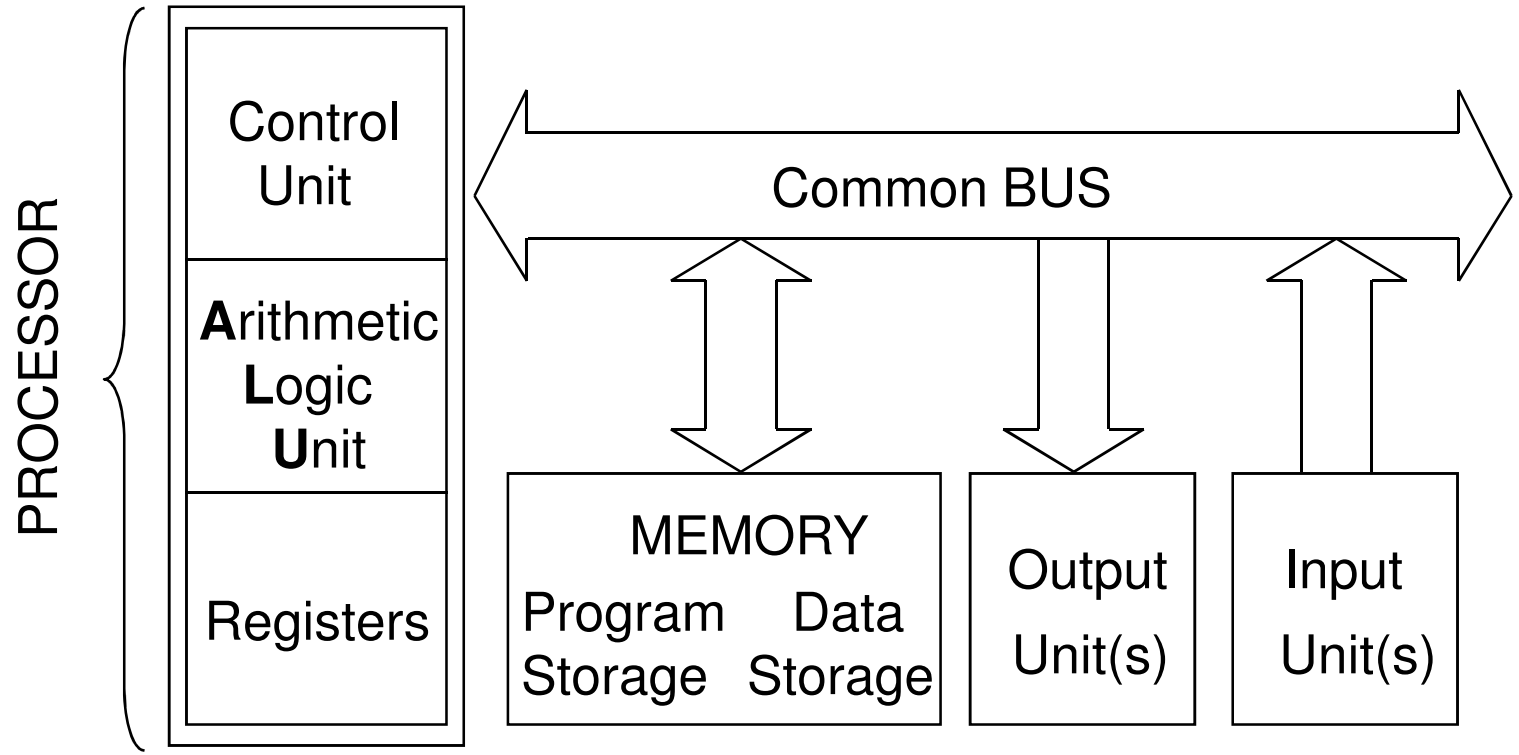
An Electronic Device operating under Control of Instructions (**Software**) stored in its own Memory Unit (part of its **Hardware**)

1. Accepts Data (Input)
2. Processes Data Arithmetically / Logically
3. Displays Information from the processing (Output)
4. Stores results for future use

Block Diagram of the **Basic Computer System**



Computer Hardware Organization



Semiconductor memory types (Physical viewpoint)

- **Random-access memory (RAM):** can be read & written.
  Volatile -> information is lost when power is turned off
- **Read-only memory (ROM):** can be read but not written by
  the processor. Keeps information in absence of power supply

Random-access memory (RAM)

- **Dynamic random-access memory (DRAM):** periodic
  refresh is required to maintain the contents of a DRAM chip
- **Static random-access memory (SRAM):** no periodic
  refresh is required

Read-only memory (ROM)

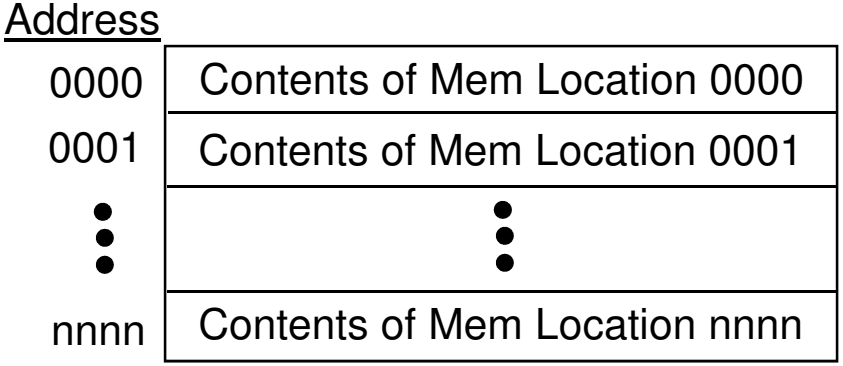
- **Mask-programmed read-only memory (MROM):**
  programmed when being manufactured
- **Programmable read-only memory (PROM):**
  the memory chip can be programmed by the end user

- **Erasable programmable ROM (EPROM)**
  1. electrically programmable many times
  2. erased by ultraviolet light (through a window)
  3. erasable in bulk (whole chip in one erasure operation)

- **Electrically erasable programmable ROM (EEPROM)**
  1. electrically programmable many times
  2. electrically erasable many times
  3. can be erased one location, one row, or whole chip in one operation

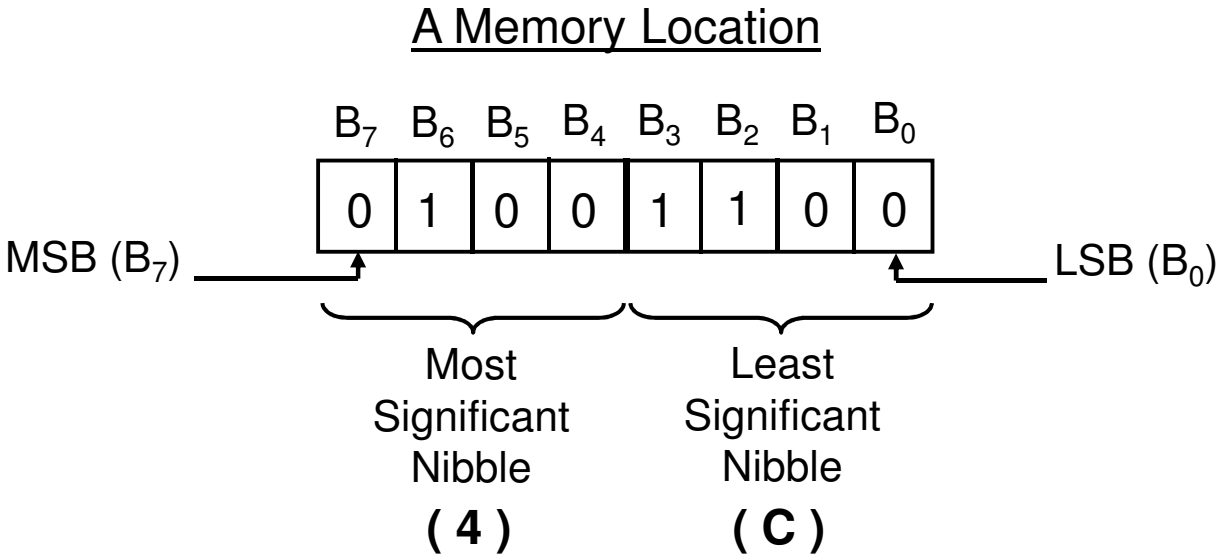
- **Flash memory**
  1. electrically programmable many times
  2. electrically erasable many times
  3. can only be erased in bulk

Memory Organization (Logical point of view)



- Each memory location is associated with an address, and
- Serves as storage for data or program code (instructions)

Basic Unit of Memory – The Byte



- Memory Content = $4C_H$ = \$ 4C = 0100 1100_B = % 0100 1100
- If the content of this memory location is interpreted by the CPU as an instruction:

Machine Code		Assembler Code
\$4C	=	INCA

Data Formats

Formats	1-Byte Range	Example 1 $0100\ 1000_b$	Example 2 $1100\ 1010_b$
Unsigned number	$0..255_d$	$64 + 8 = 72_d$	$128+ 64+ 8+ 2 = 202_d$
Signed number	$-128..127_d$	72_d	$-0011\ 0110_b = -36_h = -54_d$
ASCII character	<NUL>..<DEL> $00 .. 127_d$	48_h = 'H' (Table lookup)	out-of-range (*), or If B_7, $4A_h$ = 'J'
Binary-coded decimal (BCD)	$00..99_d$	48_d	<invalid>

(*) 202_d = '╩', See http://www.lookuptables.com/ for Extended ASCII codes

ASCII Chart

ASCII CHARACTER SET (7-Bit Code)								
MS Dig. / LS Dig.	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Numeric Systems

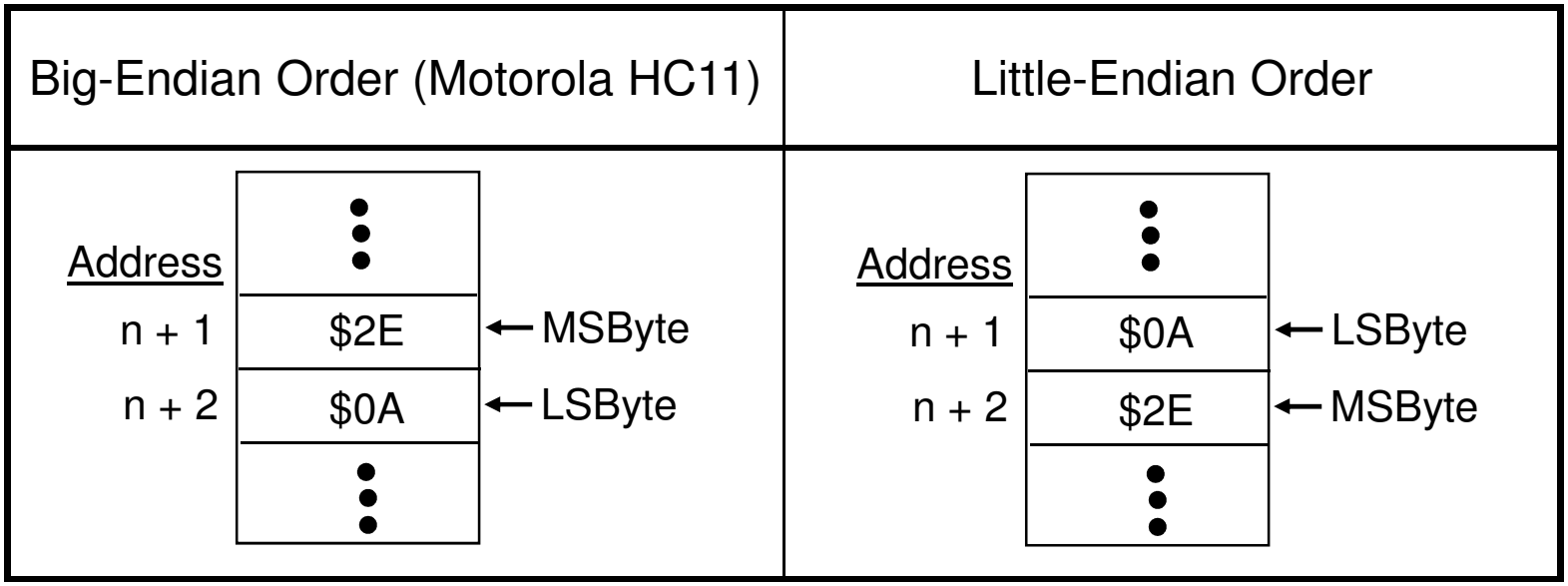
System	Example	HC11 Notation
Binary	$1101\ 1000_b$	%1101 1000
Hexadecimal	$D8_h$	$D8
Decimal	216_d	!216
Octal	330_o	@330

Memory Segments

- Program : Contains program instructions,
  - *e.g.* operation codes (OpCodes), instruction operands
  - Register associated : PC (program counter)
  - Typical access order: Top-to-bottom
- Data : Holds constants & variables used by the program
  - Registers associated : IX, IY (index registers)
  - Access order dependent on data structures & program logic
- Stack : Stores temporary variables,
  - *e.g.* subroutine parameters, return addresses
  - Register associated : SP (stack pointer)
  - Typical access order: Last-In First-Out (LIFO)

Storage of 2-Byte Data in 1-Byte Memory Locations

Example: Storage of 16-bit number $2E0A



MSByte – Most Significant Byte

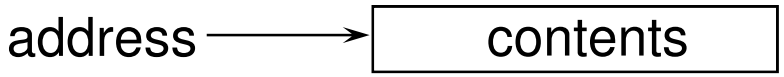
LSByte – Least Significant Byte

Valid for all memory segments:
Program, Data & Stack

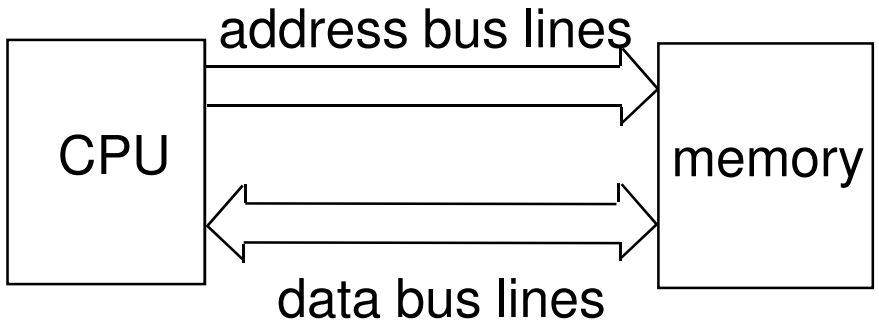
Memory Addressing

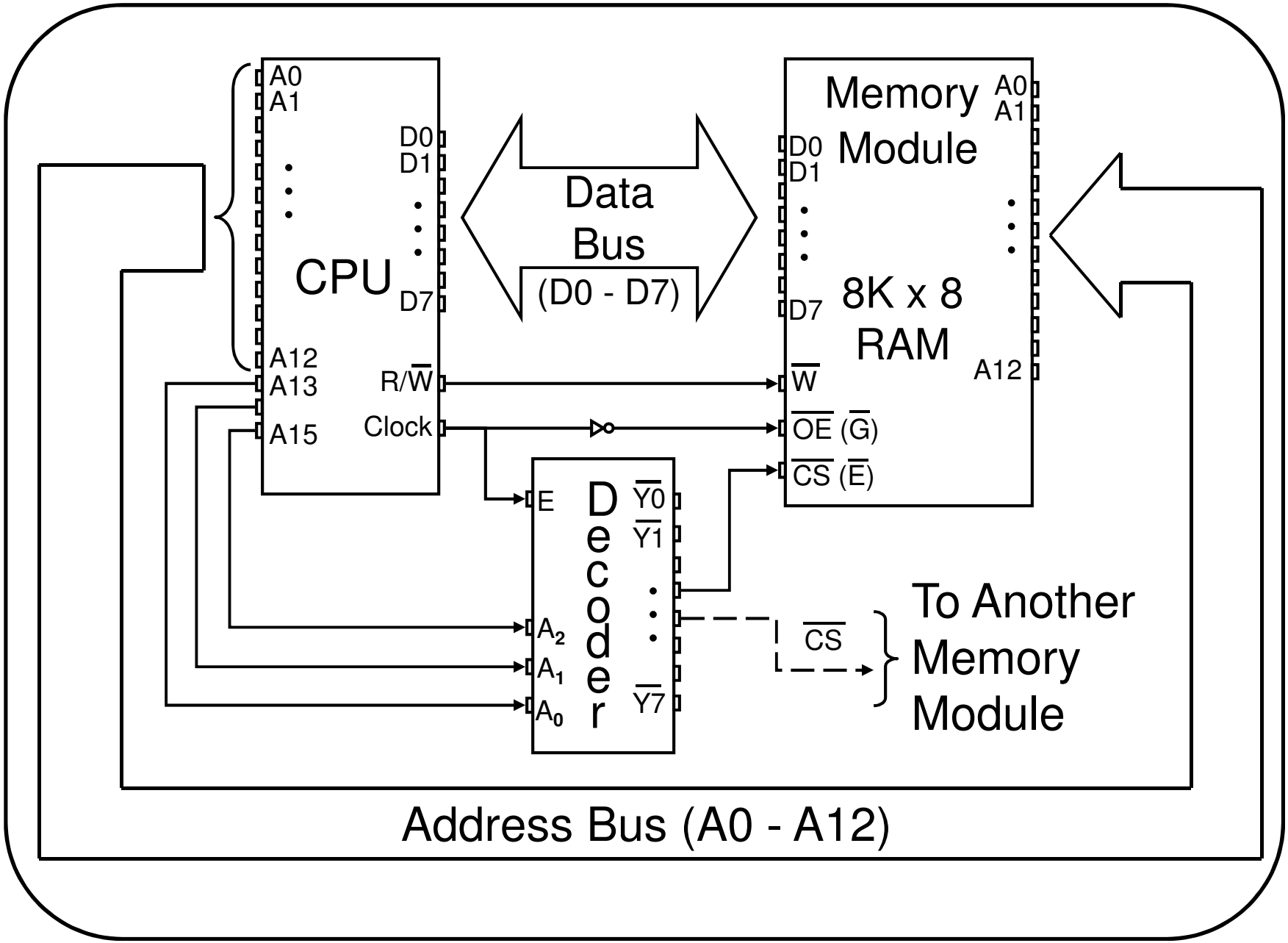
Memory consists of addressable locations

A memory location has 2 components: address and contents



Data transfer between CPU and memory



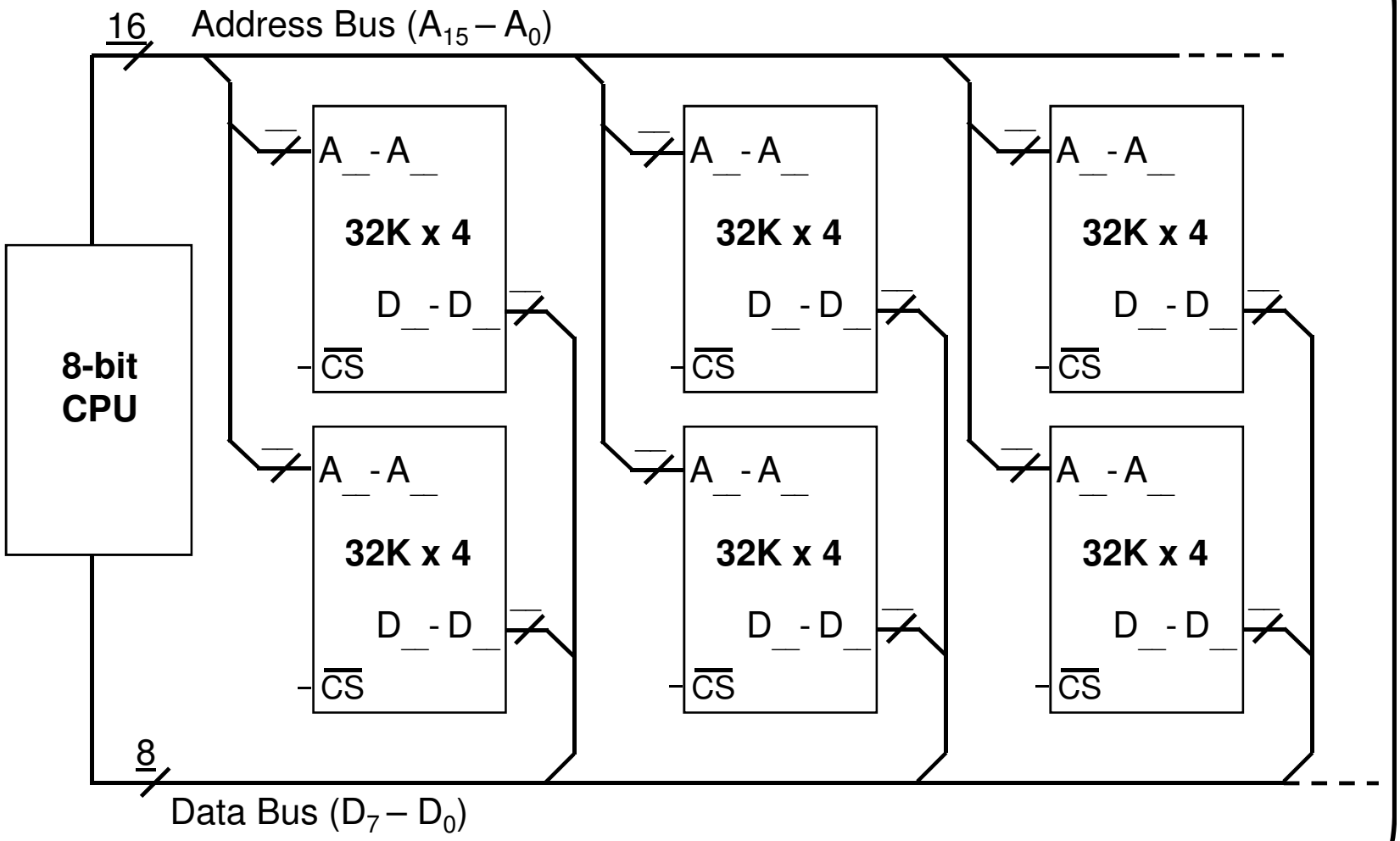

A0
A1
D0
D1
CPU
D7
A12
A13
A15
R/W
Clock
Data
Bus
(D0 - D7)
Memory
Module
8K x 8
RAM
W
OE (G)
CS (E)
E
Decoder
Y0
Y1
Y7
A2
A1
A0
CS
To Another
Memory
Module
Address Bus (A0 - A12)

Busses

- Address Bus
  - Set of parallel lines used to specify a memory location
  - Unidirectional (CPU $\rightarrow$ Memory)
  - # lines = # bits required to address all memory locations.
    - e.g. For an 8K memory module, 8K = $2^3 \times 2^{10} = 2^{13}$ locations
    - Hence, 13 lines are required
- Data Bus
  - Set of parallel lines carrying data / instructions
    - e.g. An 8-bit CPU can transfer 8 bits (1byte) of data at a time
  - Bidirectional (CPU $\leftrightarrow$ Memory)
- Control Bus
  - Set of lines controlling data transfer
  - Example of lines
    - $\overline{CS}$: chip selection logic
    - Clock : synch signal,
    - R/$\overline{W}$: Read or Write

    } Unidirectional (CPU $\rightarrow$ Memory)

Exercise: Provide this 8-bit CPU with a 64Kbyte Memory Space



Note : The Clock (Ck) & R/$\overline{W}$ signals have been omitted for simplicity

I/O Schemes

1. Isolated I/O scheme

- The microprocessor has dedicated instructions for I/O operations
- The microprocessor has a separate address space for I/O devices

2. Memory-mapped I/O scheme

- The microprocessor uses the same instruction set for I/O operations
- The I/O devices and memory components are resident in the same memory space

Synchronizing the Microprocessor and the Interface Chip

*The **polling** method*

1. for input -- the ***microprocessor checks*** a ***status*** bit of the interface chip to find out if the interface chip has received new data from the input device.
2. for output -- the ***microprocessor checks*** a ***status*** bit of the interface chip to find out if it can send new data to the interface chip.

*The **interrupt**-driven method*

1. for input -- the ***interface chip interrupts*** the microprocessor whenever it has received new data from the input device.
2. for output -- the ***interface chip interrupts*** the microprocessor whenever it can accept new data from the microprocessor.

Motorola S-Records

- Files containing Machine Code (“*.s19”)
- ASCII files - portable (edited on any PC)
- Readable
  - Hex machine code
  - Memory addresses where code will be loaded

S-Record Format

S <Type> <Length> <Address> <Code/Data> <Checksum>

ONE printable char (0..9)

TWO printable chars.

Specify the record length in bytes, counting address + code/data + checksum fields

2-byte address

(FOUR printable chars)

Executable code and/or data.

(Up to 64 bytes)

TWO printable chars.

Least significant byte of the 1's complement of the sum of the values in the record length + address + code/data fields

S-Record Example

S0 0E 0000 53 52 45 43 4F 52 44 2E 42 41 4B E3 ⇐ Starting Record

	Address	Instruction / Data
S1 04 C000 FE 3D	$C000	LDX $C008
S1 05 C001 C0 08 71		
S1 04 C003 BD 7B	$C003	JSR $C00B
S1 05 C004 C0 0B 6B		
S1 09 C006 20 FE 00 0A 05 FF 04	$C006	BRA $FE
	$C008	$000A } Data
	$C00A	$05 } Data
S1 05 C00C C0 0F 5F	$C00B	STX $C00F
S1 04 C00E 39 F4	$C00E	RTS
S1 04 C010 00 2B	$C010	$00 } Data
S9 03 0000 FC		⇐ Termination Record

Type **Length** **Address** **Code / Data** **Checksum**

Loading Executable S-Records ("*.s19") into the EVB

- 1st - Establish communication with the EVB from the PC

  - Establish the serial connection between the COM1 Port on the PC side and the Terminal I/O Port on the EVB

- 2nd - On the BUFFALO window in MiniIDE type the command

  > load –t {hit Enter}

  EVB ready for S-record stream through (t)erminal port

- 3rd - Open a 2nd MiniIDE window and connect it as a Terminal window

  - Go to Menu Terminal -> Download File -> Browse for the .s19 file to download to the EVB board

Loading Executable S-Records into the QEVB11 Simulator

- (a) Use the Load command from the File pull-down menu, OR
- (b) Click on the button with the blue arrow on top of a stack of papers
- Locate the .s19 s-Record in the PC
- Machine code will be loaded into RAM memory of EVB model

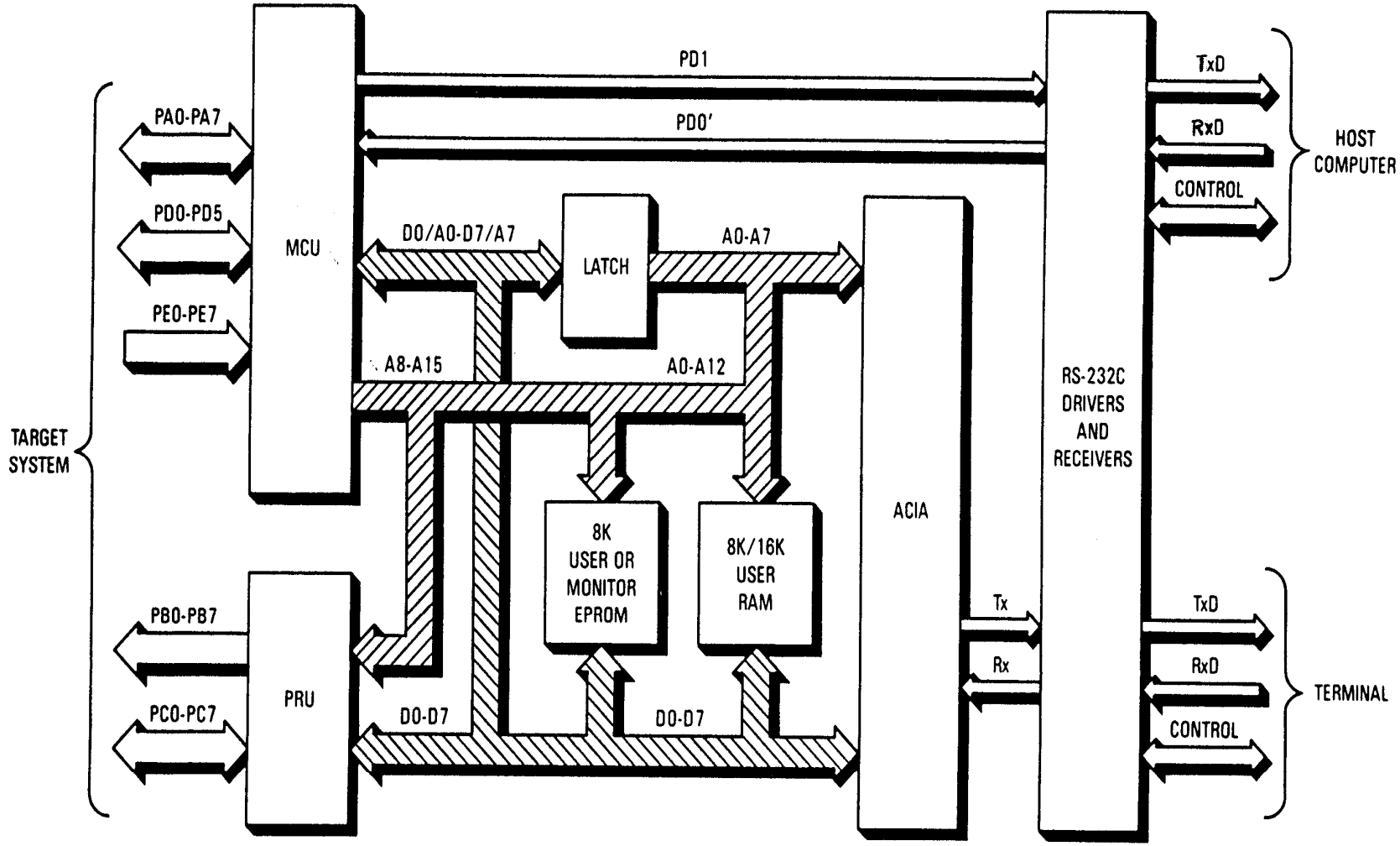


Figure 1-2. EVB Block Diagram

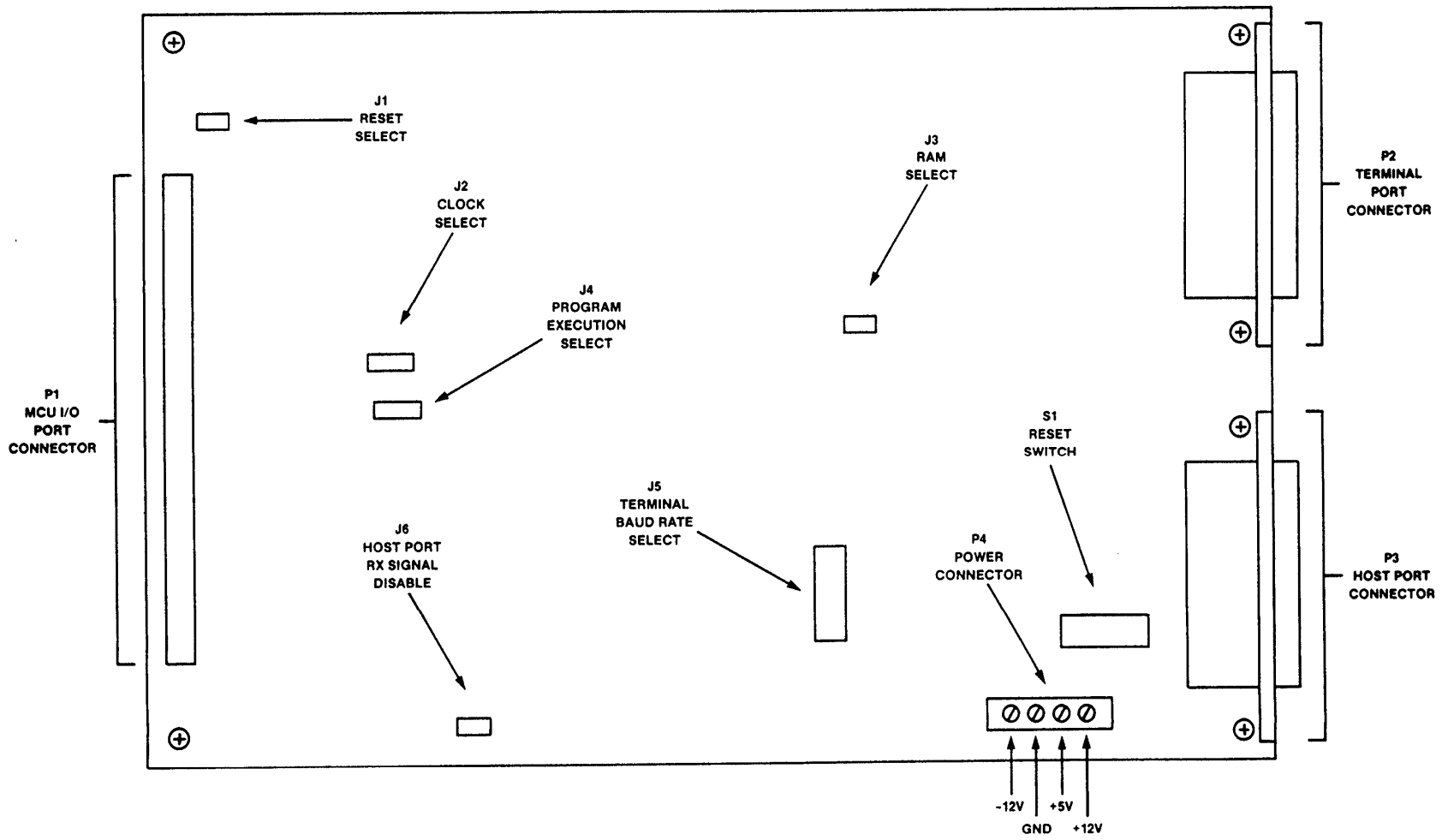


Figure 2-1. EVB Connector, Switch, and Jumper Header Location Diagram

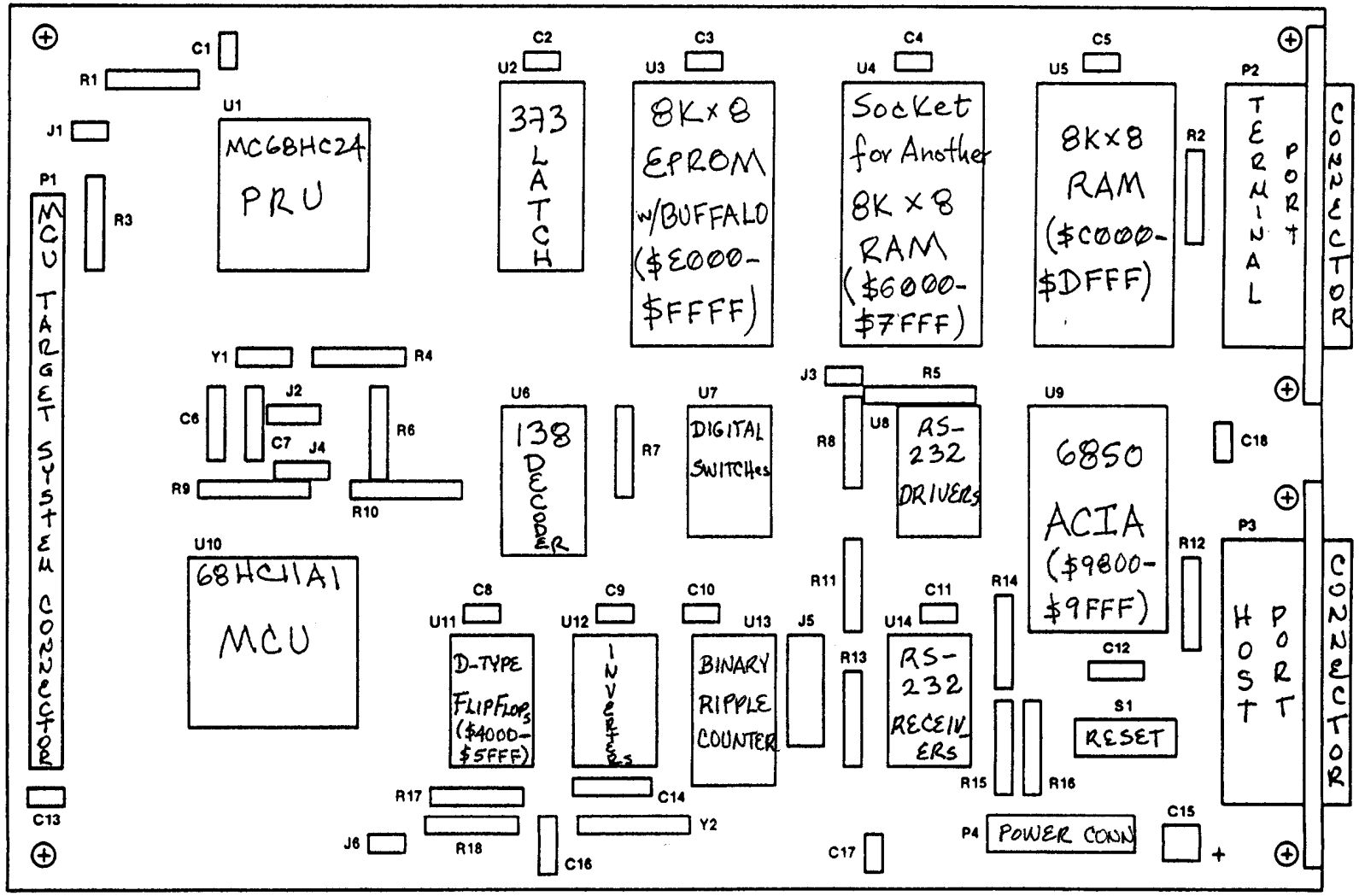


Figure 6-1. EVB Parts Location Diagram

Table 6-5. EVB Parts List

REFERENCE DESIGNATION	DESCRIPTION
C1-C5, C8-C11, C13, C14, C17, C18	Capacitor, 0.1 µF @ 50 Vdc, ±20%
C6, C7, C16	Capacitor, 24 pF @ 50 Vdc, ±20%
C12	Capacitor, 1.0 µF @ 50 Vdc, ±20%
C15	Capacitor, electrolytic, 100 µF @ 50 Vdc, ±20%
J1, J3, J6 J2, J4 J5	Header, jumper, single row post, 2 pin, Aptronics #929705-01-02 Header, jumper, single row post, 3 pin, Aptronics #929705-01-03 Header, jumper, double row post, 12 pin, Aptronics #929715-01-06.
P1 P2, P3 P4	Header, double row post, 60 pin, Aptronics #929715-01-30 (MCU I/O port connector) Connector, cable, 25-pin, ITT #DBP-25SAA (terminal/host I/O port connector) Terminal block, 4 position, Electrovert #25.112.0453 (power supply connector)
R1-R3, R5-R10 R4 R11, R13-R15 R12 R16 R17, R18	Resistor, 10k ohm, 5%, 1/4W Resistor, 10M ohm, 5%, 1/4W Resistor, 27k ohm, 5%, 1/4W Resistor, 620 ohm, 5%, 1/4W Resistor, 100k ohm, 5%, 1/4W Resistor, 2.2k ohm, 5%, 1/4W
S1	Switch, pushbutton, SPDT, C&K #8125-R2/7527
U1 U2 U3 U4 U5 U6 U7 U8 U9 U10 U11 U12 U13 U14	I.C., MC68HC24FN, PRU I.C., MC74HC373N, transparent latch I.C., 2764, 8k EPROM, 250 ns I.C., MCM6164, 8k RAM (user supplied) I.C., MCM6164, 8k RAM, 250 ns I.C., MC74HC138, decoder/demultiplexer I.C., MC74HC4066/MC14066B, digital switch I.C., MC1488P, RS-232C driver I.C., MC68B50P, ACIA I.C., MC68HC11A1FN, MCU (Note 1) I.C., MC74HC74, D-type flip-flop I.C., MC74HC14, inverter I.C., MC74HC4040, binary ripple counter I.C., MC1489P, RS-232C receiver
XU1 XU3-XU5 XU9 XU10	Socket, PC mount, 44 pin, PLCC, AMP #821-551-1 (use with U1) Socket, 28 pin, DIP, Robinson Nugent #ICL-286-S7-TG (use with U3-U5) Socket, 24 pin, DIP, Robinson Nugent #ICL-246-S7-TG (use with U9). Socket, PC mount, 52 pin, PLCC, AMP #821-575-1 (use with U10)
Y1 Y2 	Crystal, MCU, 8.0 MHz (Notes 2 & 3) Crystal, ACIA, 2.4576 MHz Fabricated jumper, Aptronics #929955-00 (use with jumper headers J1-J6)

NOTES:
(1) MCU supplied with the EVB have the configuration (CONFIG) register ROMON bit cleared to disable MCU internal ROM, thereby allowing external EPROM containing the BUFFALO program to control EVB operations.
(2) Crystal frequencies from 4 to 8 MHz (up to 8.4 MHz) can be used without any changes to the 24 pF capacitors (C6 and C7) and 10M ohm resistor (R5) values.
(3) 8 MHZ crystal obtains 2 MHz E-clock/9600 baud MCU SCI operation. 4 MHZ crystal obtains 1 MHz E-clock/4800 baud MCU SCI operation.

BUFFALO Commands

asm	–	assemble / disassemble memory locations
br(eak)	–	set / clear breakpoints
g(o)	–	execute instructions
load	–	load S-records via serial ports
md	–	display memory contents
mm	–	view / modify memory contents
p	–	proceed / continue execution
rm	–	view / modify contents of CPU registers
t(race)	–	trace execution of instructions
h(elp)	–	offers commands' syntax & brief description

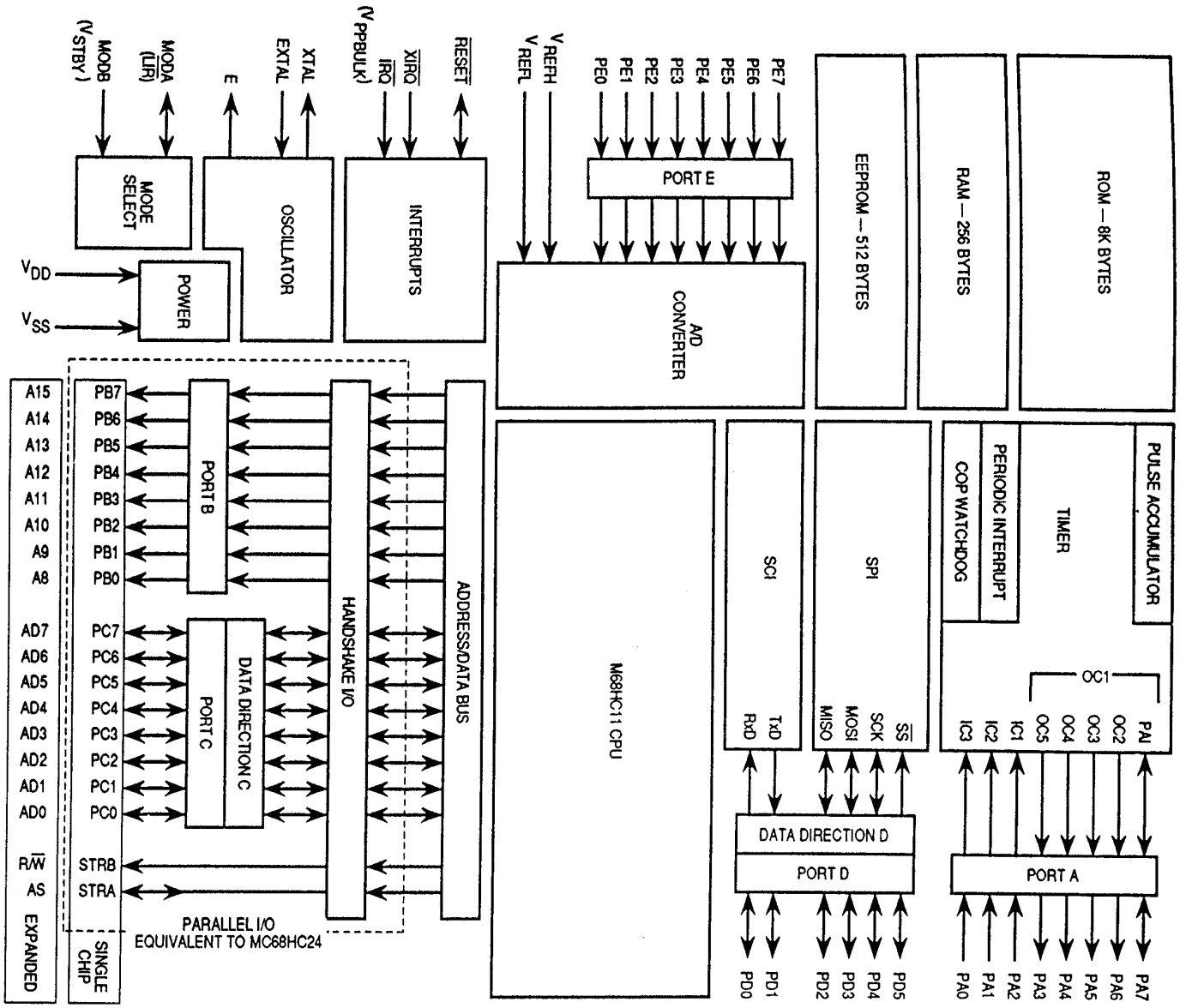
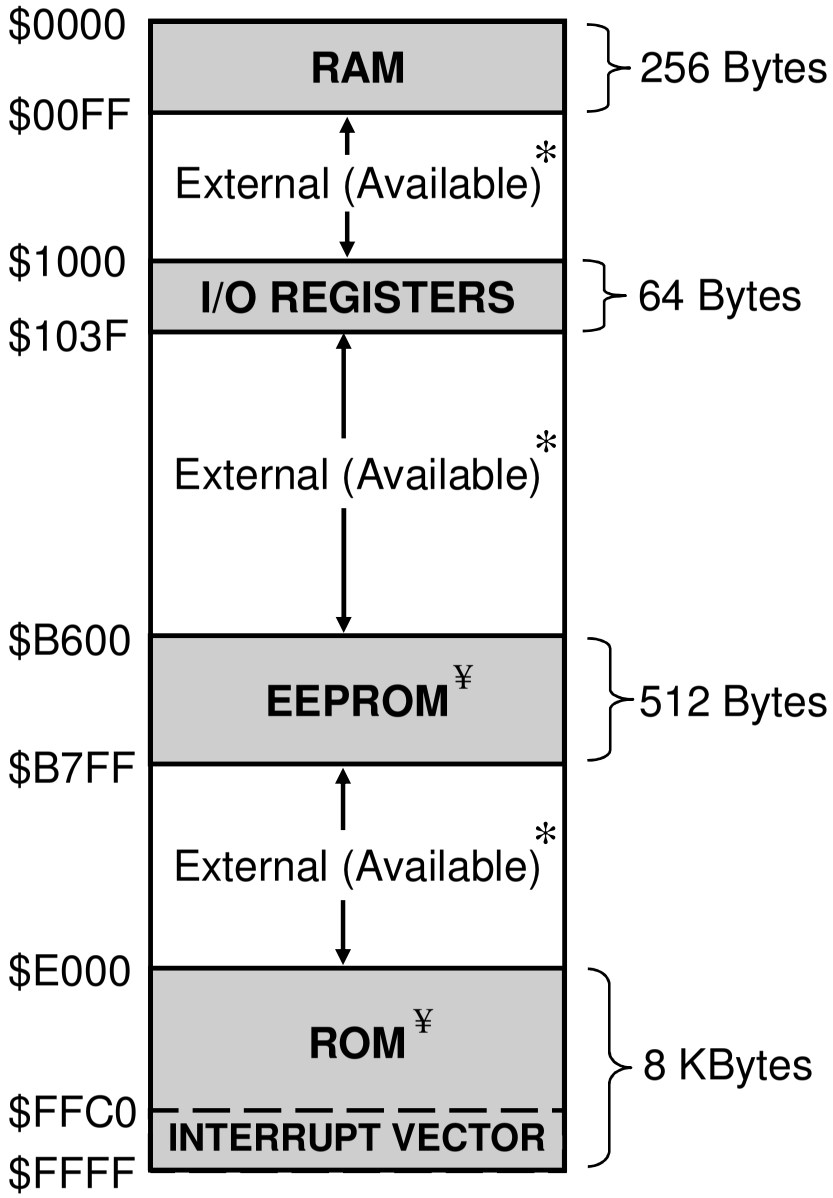


Figure 1-1. Block Diagram

HC11 Memory Map



* External Memory has a meaning only for Expanded Mode

¥ Either or both internal ROM & EEPROM can be disabled

$0000-$0035 USER RAM
$0036-$004A USER STACK POINTER
$004B-$00C3 MONITOR VARIABLES
$00C4-$00FF VECTOR JUMP TABLE

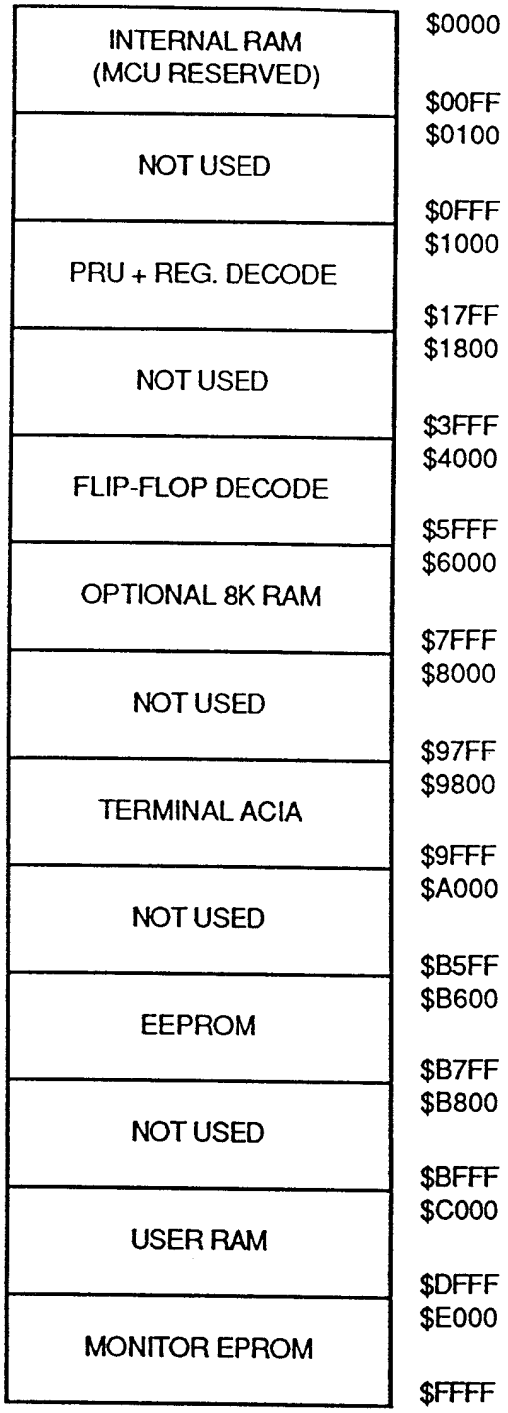
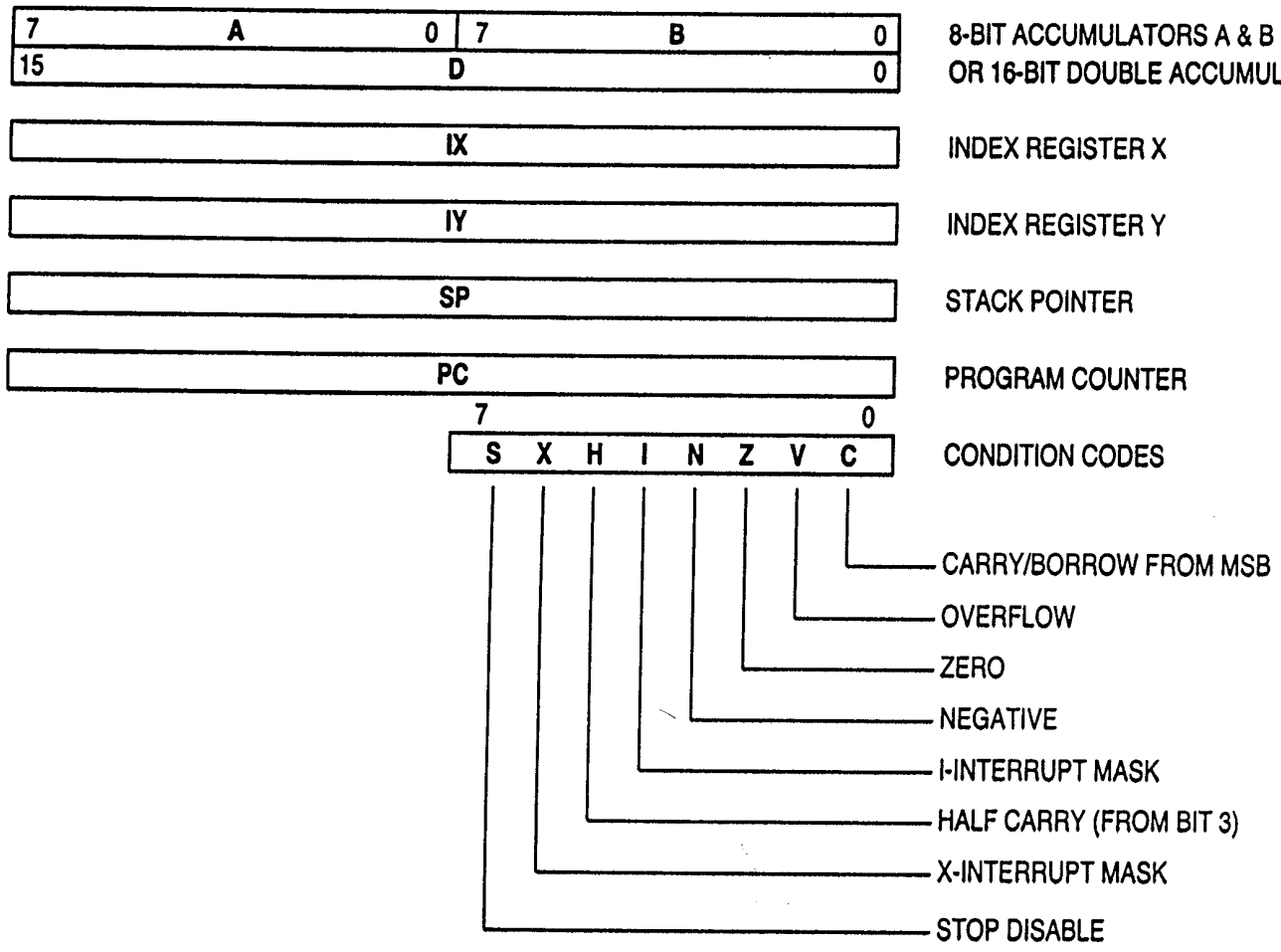


Figure 5-2. EVB Memory Map Diagram



M68HC11 Programmer's Model

Branch Instructions

Function	Mnemonic	REL	DIR	INDX	INDY	Comments
Branch if Carry Clear	BCC	X				C = 0 ?
Branch if Carry Set	BCS	X				C = 1 ?
Branch if Equal Zero	BEQ	X				Z = 1 ?
Branch if Greater Than or Equal	BGE	X				Signed ≥
Branch if Greater Than	BGT	X				Signed >
Branch if Higher	BHI	X				Unsigned >
Branch if Higher or Same (same as BCC)	BHS	X				Unsigned ≥
Branch if Less Than or Equal	BLE	X				Signed ≤
Branch if Lower (same as BCS)	BLO	X				Unsigned <
Branch if Lower or Same	BLS	X				Unsigned ≤
Branch if Less Than	BLT	X				Signed <
Branch if Minus	BMI	X				N = 1 ?
Branch if Not Equal	BNE	X				Z = 0 ?
Branch if Plus	BPL	X				N = 0 ?
Branch if Bit(s) Clear in Memory Byte	BRCLR		X	X	X	Bit Manipulation
Branch Never	BRN	X				3-cycle NOP
Branch if Bit(s) Set in Memory Byte	BRSET		X	X	X	Bit Manipulation
Branch if Overflow Clear	BVC	X				V = 0 ?
Branch if Overflow Set	BVS	X				V = 1 ?

HC11 CPU Addressing Modes

- Describe the primary operand involved in an instruction
- Operands can be
  - CPU registers, and/or
  - Bytes from memory

1. Inherent (INH)
2. Immediate (IMM)
3. Direct (DIR)
4. Extended (EXT)
5. Indexed (IND)
6. Relative (REL)

Inherent (INH)

Only CPU registers are involved in the instruction

Examples

Machine Code	Instruction	Description
1B	ABA	ACCA $\leftarrow$ ACCA + ACCB
5C	INCB	ACCB $\leftarrow$ ACCB + 1
08	INX	IX $\leftarrow$ IX + 1
16	TAB	ACCB $\leftarrow$ ACCA

Immediate (IMM)

– The operand value is part of the instruction

– It follows the OpCode

Examples

Machine Code	Instruction	Description
86 25	LDAA #$25	ACCA ← $25
81 24	CMPA #%100100	ACCA – %00100100
CC 07 D2	LDD #!2002	ACCA:ACCB ← $07D2
		Same as ACCD ← $07D2

Direct (DIR)

– The operand is stored in initial 256 bytes ($0000 – $00FF)

Examples

Machine Code	Instruction	Description
90 1F	SUBA $1F	ACCA ← ACCA – <$001F>
96 A8	LDAA $A8	ACCA ← <$00A8>

Extended (EXT)

– The operand's absolute address appears explicitly in the 2 bytes following the OpCode (any in $0000 – $FFFF)

Examples

Machine Code	Instruction	Description
B7 C0 20	STAA $C020	<$C020> ← ACCA
F0 C0 1C	SUBB $C01C	ACCB ← ACCB – <$C01C>

Indexed (IND)

– Index registers IX & IY are used to calculate the effective address (EA). It can be any in $0000 – $FFFF.

– EA = Base Address + Unsigned 8-bit Offset

IX or IY

Examples

Machine Code	Instruction	Description
E3 22	ADDD $22,X	EA = IX + $22 ACCD ← ACCD + <EA:EA+1>
18 AB 0D	ADDA $0D,Y	EA = IY + $0D ACCA ← ACCA + <EA>

Relative (REL)

- It is used only by the branch instructions
- EA = Next Instruction's Address + Signed 8-bit Offset, OR

$$PC_{NEW} = EA = PC_{OLD} + Offset$$

- Offset range: $[\ -128_D\ ,\ 127_D\ ]$

Relative (REL)

– PC_{NEW} = EA = PC_{OLD} + Offset

Example: Fill in the spaces in the machine code below

Address	Mach Code	Label	Operation	Operand	Description
C000	20 __	there	BRA	where	branch always
C002	22 __	where	BHI	there	branch if higher
C004	24 __		BCC	lbcc	branch if carry clear
C006	27 __	hang	BEQ	hang	branch if Z = 1
C008	25 __	here	BLO	here	branch if lower
C00A	8D __	lbcc	BSR	subr1	branch to subroutine
	⋮ }	10 bytes of code			
C016	4F	subr1	CLRA		
	⋮				

The 68HC11 Machine Code

A 68HC11 instruction consists of

(1 to 2 bytes) of opcode + (0 to 3 bytes) of operand information

Examples

Assembly instruction		Machine instruction (in hex format always)
INCB		5C
LDAA	#!29	86 1D
ADDA	$002F	9B 2F (assembler encodes using direct addressing mode)
STAA	$C01E	B7 C0 1E
CPD	#$00FF	1A 83 00 FF
loop BRCLR	0, Y, $80, loop	18 1F 00 80 FB

Decoding machine language instructions

Procedure

Step 1 Compare the first one or two bytes with the opcode table to identify the corresponding assembly mnemonic and addressing mode.

Step 2 Identify the operand bytes after the opcode field.

Step 3 Write down the corresponding assembly instruction.

Step 4 Repeat step 1 to 3 until the machine code file is exhausted.

Sample lookup table to be used in decoding the program segment of the next example into assembly instructions

machine code	assembly instruction format
01	NOP
86	LDAA IMM
8B	ADDA IMM
96	LDAA DIR
97	STAA DIR
9B	ADDA DIR
C3	ADDD IMM
C6	LDAB IMM
CB	ADDB IMM
CC	LDD IMM
D3	ADDD DIR
D6	LDAB DIR
D7	STAB DIR
DB	ADDB DIR
DC	LDD DIR
DD	STD DIR

Example. Disassemble the following machine code to its corresponding assembly instructions.

96 30 8B 17 97 30 CC 02 F0

Solution:

The disassembly process starts from the leftmost byte. We next look up the machine code table to see which instruction it corresponds to.

Instruction 1.
Step 1. The first byte 96 corresponds to the instruction LDAA DIR.
Step 2. The second byte, 30_h, is the direct address.
Step 3. Therefore, the first instruction is LDAA $30.

Instruction 2.
Step 1. The third byte (8B) corresponds to the instruction ADDA IMM.
Step 2. The immediate value is 17_h.
Step 3. Therefore, the second instruction is ADDA #$17.

Instruction 3.
Step 1. The fifth byte (97) corresponds to the instruction STAA DIR.
Step 2. The DIR address is the next byte 30.
Step 3. Therefore, the third instruction is STAA $30.

Instruction 4.
Step 1. The seventh byte (CC) corresponds to the instruction LDD IMM.
Step 2. The IMM 16-bit value is given by the next 2 bytes 02 F0.
Step 3. Therefore, the fourth instruction is LDD #$02F0.

Development Process
People work here
Editor
.ASM
Assembler
.LIS
human readable results (including assembly errors)
.OBJ
may link multiple OBJ files
Linker
.EXE
loader is part of operating system (or possibly part of a debugger)
Loader
memory
processor PC
Computer System

Assembler Line Statement Format

Label Field	Operation Field	Operand Field	Comment Field
	ldab	#26	; Initializing Delay Counter
DelayLoop	decb		; Decrement Counter Value
	bne	DelayLoop	; If Counter not Zero stay in the Loop

Assembler Directives

The ORG directive

Example

```
            org     $C000       ; Code to follow starts at $C000
            lds     #StkTop     ; Initializing Stack Pointer Register
             .
             .
             .
            org     $DFFF       ; Base of stack identified with label StkTop
StkTop                          ; Address $DFFF is assigned to this label
            end
```

The END directive.-

Instruct the assembler to stop the assembly process for this module. Any directive or code following it is ignored.

Assembler Directives

The AORG directive (absolute ORG) .-

Instruct the linker not to relocate the code segment following it.

Example

```
        aorg    $C300       ; Code to follow starts at $C300
        ldaa    Counter     ; No matter what memory was
        inca                ; assigned to the last instruction of the
         :                  ; previous module
```

Assembler Directives

The PUBLIC directive .-

Allows a module to share a label (e.g. a subroutine) with other modules by making its name public or known to others

Example

```
                public    ASCII2Dec
;
; Function      ASCII2Dec
;               < Description of what it does and
;               parameters or variables involved >
;
                org       $C500       ; Code to follow starts at $C500
ASCII2Dec       psha
                 .
                 .
                 .
                rts

                end
```

Body of the subroutine

Assembler Directives

The EXTERN directive .-

Allows a module to have access to a public label external to this module, i.e. not defined in this module

Example

```
extern   ASCII2Dec
  .
  .
  .
jsr      ASCII2Dec    ; Call to subroutine ASCII2Dec
  .
  .
  .
end
```

The EQU directive .- Constants

Unnamed Constants

Examples:

```
        lds   #$DFFF ; Initializing stack pointer
        cmpb  #'A'   ; Compare ACCB with ASCII 'A' = $41
        staa  $1004  ; Store <ACCA> to PORTB data register ($1004)
```

Named Constants

Examples:

```
STK_TOP  equ  $DFFF  ; Top of Stack at start of program
CAP_A    equ  'A'    ; ASCII for uppercase A
PORTB    equ  $1004  ; PORTB data register address
```

Constant Definition Part

```
         lds   #STK_TOP  ; Initializing stack pointer
         cmpb  #CAP_A    ; Checking contents of ACCB w.r.t. 'A'
         staa  PORTB     ; Writing contents of ACCA to PORTB
```

Program Instructions

Advantages of Named Constants

1. Their value only needs to be changed once (in the Definition Part)
2. Improves readability of Assembly Code

Other Examples:

```
; Constants
DELAY     equ   2000        ; Delay value to initialize counter with
BITMASK1  equ   %00000001   ; Mask used for parity, bit 0 (B0)
BITMASK2  equ   %00110000   ; Mask used to toggle bits 4 & 5 (B4 & B5)

; Instructions (These are just isolated examples, NOT part of a program)
          ldx   #DELAY      ; Initializing delay counter IX
          bita  #BITMASK1   ; Checking if B0 is 0 or 1, <ACCA> even or odd
          anda  #BITMASK1   ; Does the same as bita but also modify ACCA
                            ; In this case ACCA B1-B7 are cleared
          eora  #BITMASK2   ; Toggles ACCA bits 4 & 5 (B4 & B5)
```

Using Variables in Assembly Programming.- Example

```
; Instructions (These are just isolated examples, NOT part of a program)
            ldaa    Counter     ; ACCA ← Counter
            inca                ; ACCA ← ACCA + 1
            staa    Counter     ; Counter ← ACCA, updating Counter
; (or)
            inc     Counter     ; Counter ← Counter + 1, Equivalent to above

            adda    Counter     ; ACCA ← ACCA + Counter
              :
Counter     db      0
```

Defining a Known String.- Example

```
; Constants
CR          equ   $0D
LF          equ   $0A
NUL         equ   $00
              :
; Variables
str1        db    “This is string 1”
str3        db    “ABCDEFG”
NameStr     db    “JOSE”, CR, LF, NUL   ; null-terminated string including
                                        ; the format control characters
                                        ; Carriage Return & Line Feed
```

Reserving arbitrary amounts of storage .- Example

```
; Program segment that stores string "ABC…Z" to variable alphabet

; Constants

CAP_A       equ    'A'      ; First letter, 'A' has the lowest ASCII value in the set
CAP_Z       equ    'Z'      ; Last letter, 'Z' has the highest ASCII value in the set

; Instructions

            org    $C000        ; Code below to be loaded starting at $C000
            ldx    #alphabet    ; IX pointing to alphabet (loaded with its address)
            ldaa   #CAP_A       ; ACCA = 'A'
AlphaLoop   staa   0,X          ; Store value in ACCA to address held by IX
            inx                 ; Increment IX, IX points to next byte in alphabet
            inca                ; ACCA holds ASCII value for next character
            cmpa   #CAP_Z       ; Is next char lower or same as 'Z' ?
            bls    AlphaLoop    ; If YES go back to store it and repeat cycle

; Variables

alphabet    ds     26           ; Allocates 26 bytes of memory for variable
                                ; alphabet . Its values are undefined initially.
```

Reserving arbitrary storage .- List File Example

```
 1: ; Program segment that stores string "ABC...Z" to variable alphabet
 2:
 3: ; Constants
 4:       =00000041          CAP_A        equ     'A'
 5:       =0000005A          CAP_Z        equ     'Z'
 6:
 7: ; Instructions
 8:       =0000C000                       org     $C000
 9:   C000 CE C00D                        ldx     #alphabet
10:  C003 86 41                          ldaa    #CAP_A
11:  C005 A7 00 AlphaLoop     staa    0,X
12:  C007 08                             inx
13:  C008 4C                             inca
14:  C009 81 5A                          cmpa  #CAP_Z
15:  C00B 23 F8                          bls     AlphaLoop
16:
17: ; Variables
18:  C00D +001A          alphabet        ds      !26
```

```
;
; HtoD - Subroutine to convert a 16-bit hex number to a 5 digit decimal number
;
; Decimal ASCII result is stored in external 5 byte variable DBUFR
; On entry IX points to hex value to be converted
; All registers are unchanged upon return
;

            public  HtoD        ; Subroutine label other module(s)
                                ; can have access to

            extern  DBUFR       ; Variable label defined in other module

HtoD        pshy
            pshx                ; Save registers
            pshb
            psha
            ldy     #DBUFR      ; IY points to DBUFR variable
            ldd     0,X         ; ACCD = hex value to be converted
            ldx     #!10000
            idiv                ; IX = hex/10,000, ACCD = remainder (r)
            :
```

Subroutine Example (Part 1)

```
        :
        :
        :
ldx     #!100
idiv                    ; IX = r1/100, ACCD = new r (r2)
xgdx                    ; IX = r2, ACCA:ACCB = 100s digit
addb    #$30            ; Convert to ASCII
stab    2,Y             ; Store to 100s digit in decimal buffer
xgdx                    ; ACCD = r2
ldx     #!10
idiv                    ; IX = r2/10, ACCD = new r (ACCB = 1s digit)
addb    #$30            ; Convert to ASCII
stab    4,Y             ; Store to units digit in decimal buffer
xgdx                    ; ACCA:ACCB = 10s digit
addb    #$30            ; Convert to ASCII
stab    3,Y             ; Store to 10s digit in decimal buffer
pula
pulb                    ; Restore registers
pulx
puly
rts                     ; Return

end
```

Subroutine Example (Part 2)

```
; Labels shared with other module(s)

                public  DBUFR       ; Variable label other module(s)
                                    ; can have access to
                extern  HtoD        ; Subroutine label defined in other module

; Addresses

BAUD            equ     $102B       ; Address for the SCI line speed register
SCCR1           equ     $102C       ; Address for the SCI control register 1
SCSR            equ     $102E       ; Address for the SCI status register
SCDR            equ     $102F       ; Address for the SCI data register
FFLOP           equ     $4000       ; FlipFlop Address
STK_TOP         equ     $DFFF       ; Address of top-of-stack

; Constants

JMPOpCode       equ     $7E         ; OpCode for JMP instruction
TIE_TE          equ     $88         ; Control byte for SCCR2, flags TE = 1, TIE = 1
TDRE            equ     $80         ; TDRE bit mask for SCSR
DELAY           equ     !10667      ; Value used to create a 32ms delay,
   .
   .
   .
```

Main Program Example (Part 1)

```
; Program Code
                org     $C000           ; To be loaded at $C000

                lds     #STK_TOP        ; Initializing stack pointer
MainLoop        ldx     #DELAY          ; Loading delay counter
DelayLoop       dex                     ; Decrementing counter
                bne     DelayLoop       ; If counter > 0 keep decrementing
                jsr     UpdatePeriod    ; Goto Update Signal Period
                bra     MainLoop        ; Repeat Loop
   :
   :
UpdatePeriod    :
                :
                ldx     #Periodhex      ; Load IX with address of hex period
                jsr     HtoD            ; Subroutine call
                ldx     #DBUFR          ; Reading subroutine output
                :
                :
                rts                     ; End of subroutine UpdatePeriod

NewLineString   db      LF, CR, ' ', NUL
DBUFR           ds      5               } Variables (Part of them)
Periodhexdw     0

                end                     ; End of Program Code
```

Main Program Example (Part 2)

M68HC11 Instruction Set & Assembly Programming

Bibliography:

- 68HC11 Reference Manual, Section 6.5 and Appendix A.
- 68HC11 Programming Reference Guide, Section 3.
- Textbook, Chapter 3.
- Huang's book, Chapter 2.

M68HC11 Instruction Set

Load & Store Instructions .- Examples

```
ldaa   #$2C     ; ACCA ← $2C
ldab   $C007    ; ACCB ← <$C007>
staa   $C00A    ; <$C00A> ← ACCA
```

Register Transfer & Exchange Instructions

```
tab     ; ACCB ← ACCA
tba     ; ACCA ← ACCB

xgdx    ; ACCD ↔ IX
xgdy    ; ACCD ↔ IY
```

M68HC11 Instruction Set

Arithmetic Instructions .- Examples

```
inc     Counter   ; Counter ← Counter + 1
deca              ; ACCA ← ACCA – 1
adda    alpha     ; ACCA ← ACCA  + alpha
suba    beta      ; ACCA ← ACCA – beta
aba               ; ACCA ← ACCA  + ACCB
nega              ; ACCA ← – ACCA (2's complement)
mul               ; ACCD ← ACCA * ACCB
```

```
;
; Assembly Code for Laboratory 1, Part 5, May 2003
;
            ORG     $D000

            LDD     sum         ; Load variable sum into ACCD
            ABA                 ; Add lower byte to higher byte
            STAA    sum         ; Store it back to sum
            SWI                 ; Return control to BUFFALO

sum         DW      $0804

            END
```

Simple Program Example

```
;
; Assembly Code for Laboratory 1, Part 5, May 2004
;
            ORG     $C500

            LDD     diff        ; Load variable diff into ACCD
            SBA                 ; ACCA ← ACCA – ACCB
            STAA    diff        ; Store it back to diff
            NOP                 ; Do nothing

diff        DB      $1E, $04

            END
```

Simple Program Example

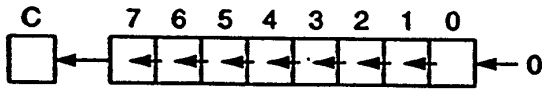
M68HC11 Instruction Set

Logical Operations .- Examples

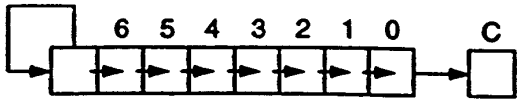
```
andb   #$F0          ; Clears Least Significant Nibble of ACCB
oraa   #$03          ; Sets Bits 0 & 1 of ACCA
eora   #$0C          ; Toggles Bits 2 & 3 of ACCA
bitb   $C01C         ; Implicit AND, ACCB • <$C01C>
                     ; Flags modified, ACCB not altered
bitb   #%00000011    ; Is ACCB multiple of 4?
       ; If as a result of this instruction Z is set
       ; (Z = 1), ACCB is multiple of 4.
ldx    #$1004
bset   0, X, $55     ; Sets bits 0,2,4 & 6 of PORTB
bclr   0, X, $AA     ; Clears bits 1,3,5,7 of PORTB
```

M68HC11 Instruction Set

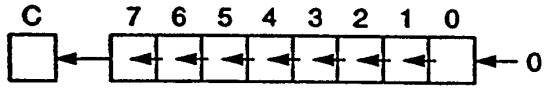
Shift and Rotate Instructions



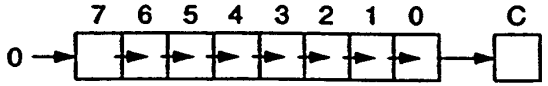
ARITHMETIC SHIFT LEFT
(ASL)



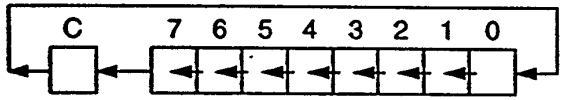
ARITHMETIC SHIFT RIGHT
(ASR)



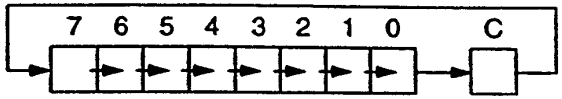
LOGICAL SHIFT LEFT
(LSL)



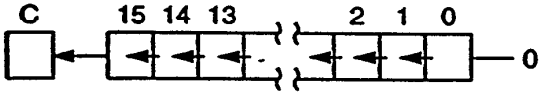
LOGICAL SHIFT RIGHT
(LSR)



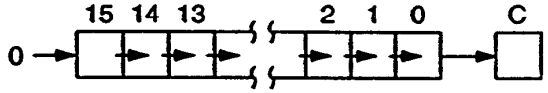
ROTATE LEFT
(ROL)



ROTATE RIGHT
(ROR)



ARITHMETIC OR LOGICAL
SHIFT LEFT DOUBLE
(ASLD OR LSLD)



LOGICAL SHIFT
RIGHT DOUBLE
(LSRD)

M68HC11 Instruction Set

Shift and Rotate Instructions .- Examples

```
ldaa   #$01   ;                ACCA = 1 = 00000001bin
asla          ; ACCA = ACCA * 2 = 2 = 00000010bin, Carry flag = 0
asla          ; ACCA = ACCA * 2 = 4 = 00000100bin, Carry flag = 0
asra          ; ACCA = ACCA / 2  = 2 = 00000010bin, Carry flag = 0

ldaa   #$F6   ;              ACCA = -10dec = $F6 = 11110110bin
asra          ; ACCA = ACCA / 2 = -5dec = $FB = 11111011bin,
              ; Carry Flag = 0
asra          ; ACCA = ACCA / 2 = -3dec = $FD = 11111101bin,
              ; Carry Flag = 1
```

M68HC11 Instruction Set

Program Control Instructions

1. Conditional branches

- Modify value of PC within $[-128,+127]_{dec}$ (1 byte signed)

(a) Testing a single CCR bit

```
beq     <label>         ; Z = 1 ?
bne     <label>         ; Z = 0 ?
bcs     <label>         ; C = 1 ?
bcc     <label>         ; C = 0 ?
```

M68HC11 Instruction Set

Program Control Instructions

1. Conditional branches

(b) Comparison of unsigned numbers

```
bhi     <label>     ; Unsigned > ,  C + Z = 0 ?
bhs     <label>     ; Unsigned ≥ ,  C = 0 ?
blo     <label>     ; Unsigned < ,  C = 1 ?
bls     <label>     ; Unsigned ≤ ,  C + Z = 1 ?
```

Example:

```
cmpa  #$25
bhi     Higher  ; Program control will be transferred to the
                ; instruction at label 'Higher' IF ACCA > $25,
                ; otherwise the instruction following bhi is
                ; executed

                ; If ACCA = $F3 = 243dec > $25 execution
                ; continues at 'Higher'
```

M68HC11 Instruction Set

Program Control Instructions

1. Conditional branches

(c) Comparison of signed numbers

```
bgt   <label>   ; Signed > ,  Z + (N ⊕ V) = 0 ?
bge   <label>   ; Signed ≥ ,  N ⊕ V = 0 ?
blt   <label>   ; Signed < ,  N ⊕ V = 1 ?
ble   <label>   ; Signed ≤ ,  Z + (N ⊕ V) = 1 ?
```

Example:

```
cmpa  #$25
bgt   Greater   ; Program control is transferred to the instruction
                ; at label 'Greater' IF the 2's complement value in
                ; ACCA > $25, otherwise the instruction following
                ; bgt is executed

                ; If ACCA = $F3 = -13dec < $25 = 37dec execution
                ; continues with the instruction following bgt
```

M68HC11 Instruction Set

Program Control Instructions

2. Unconditional branches

```
jmp  <label or address>    ; Jump always to a label / address
                           ; in the 64KB address space
bra  <label>    ; Branch always to an address in the range
                ; [ PC - 128dec , PC + 127dec ]
```

3. Subroutine calls

```
jsr    <label or address>  ; Jump to a subroutine starting with
                           ; a label / address anywhere in the
                           ; 64KB address space

bsr    <label>   ; Branch to a subroutine starting with a label
                 ; associated with an address in the range
                 ; [ PC - 128dec , PC + 127dec ],
                 ; PC is the address of the instruction following bsr

rts              ; Return from subroutine
```

M68HC11 Instruction Set

Stack Instructions

1. Saving contents of CPU registers

```
psha
pshb            ; Storing register values to stack
pshx
pshy
```

2. Retrieving contents of CPU registers

```
puly
pulx            ; Restoring register values from stack
pulb
pula
```

Internal CPU Registers	Description
BAR	**B**us **A**ddress **R**egister – 16 bits
BDR	**B**us **D**ata **R**egister – 8 bits
IR	**I**nstruction **R**egister – 8 bits
ATMP	**T**e**mp**oral **A**ddress **R**egister – 16 bits
DTMP	**T**e**mp**oral **D**ata **R**egister – 16 bits

Cycle Code	Description
FOP	**F**etch instruction **Op**code
FOFF	**F**etch 8-bit address **Off**set
FAHI	**F**etch **Hi**gh half of 16-bit **A**ddress
FALO	**F**etch **Lo**w half of 16-bit **A**ddress
ODHI	transfer **Hi**gh half of 16-bit **O**perand **D**ata
ODLO	transfer **Lo**w half of 16-bit **O**perand **D**ata
OD	transfer 8-bit **O**perand **D**ata
CA	**C**ompute operand **A**ddress (uses ALU)
EXEC	**exec**ute ("do" the instruction)

; **Assembly code for Cycle-by-Cycle Execution example**

; **Execution starts at $D500 (Reset vector is set to $D500)**

```
; Instructions
; ************
                org     $C000       ; Instruction execution is analyzed for
Code            ldaa    Data        ;       THIS,
                clrb                ;       THIS,
                jsr     Target      ;       THIS, and
                org     $C080
Target          inc     2,X         ;       THIS instruction

; Data
; ****
                org     $D000
Data            db      $19

                org     $D400
List            ds      2
                db      $39

; Initialization
; **************
                org     $D500
                lds     #STK_TOP
                ldx     #List
                ldab    #$5F
                jmp     Code

; Stack Area
; ***** ****
                org     $DFFF
STK_TOP
                end
```

The 68HC11 Instruction Execution Cycle

- Perform a sequence of read cycles to fetch instruction opcode byte(s) and address byte(s) if required.
- Optionally perform read cycle(s) required to fetch memory operand(s).
- Perform the operation specified by the opcode.
- Optionally write results back to a register or memory location(s).

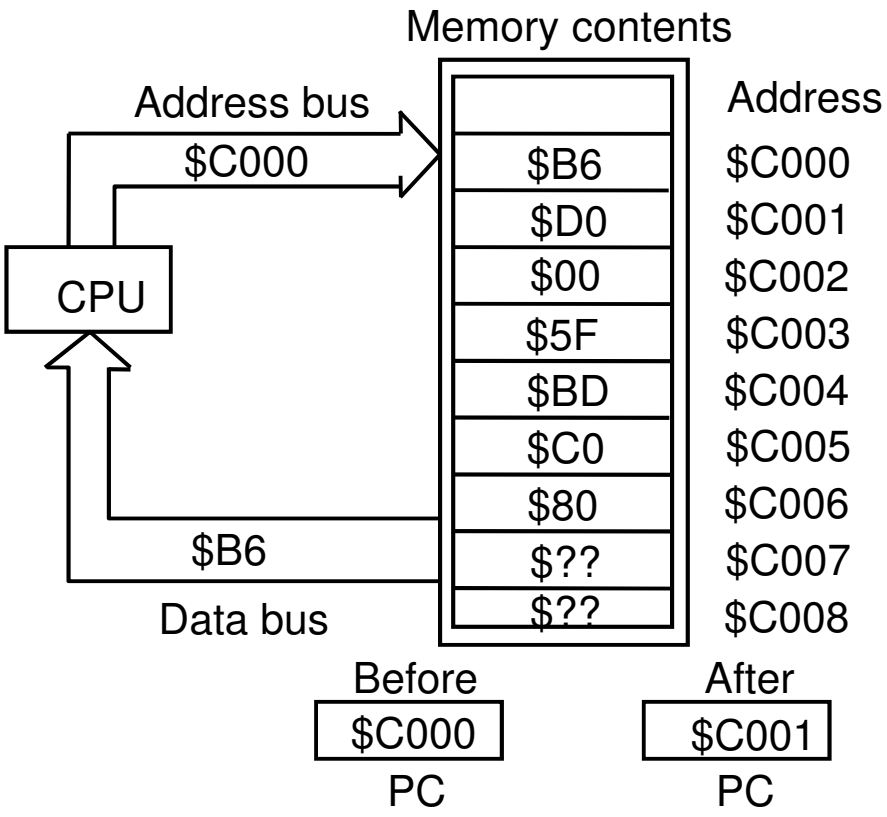
Example: Consider the following 4 instructions

Assembly instruction	***Memory location***	***Machine Code***
LDAA $D000	$C000	B6 D0 00
CLRB	$C003	5F
JSR $C080	$C004	BD C0 80
⋮	⋮	⋮
INC 2,X	$C080	6C 02

Instruction **LDAA $D000**

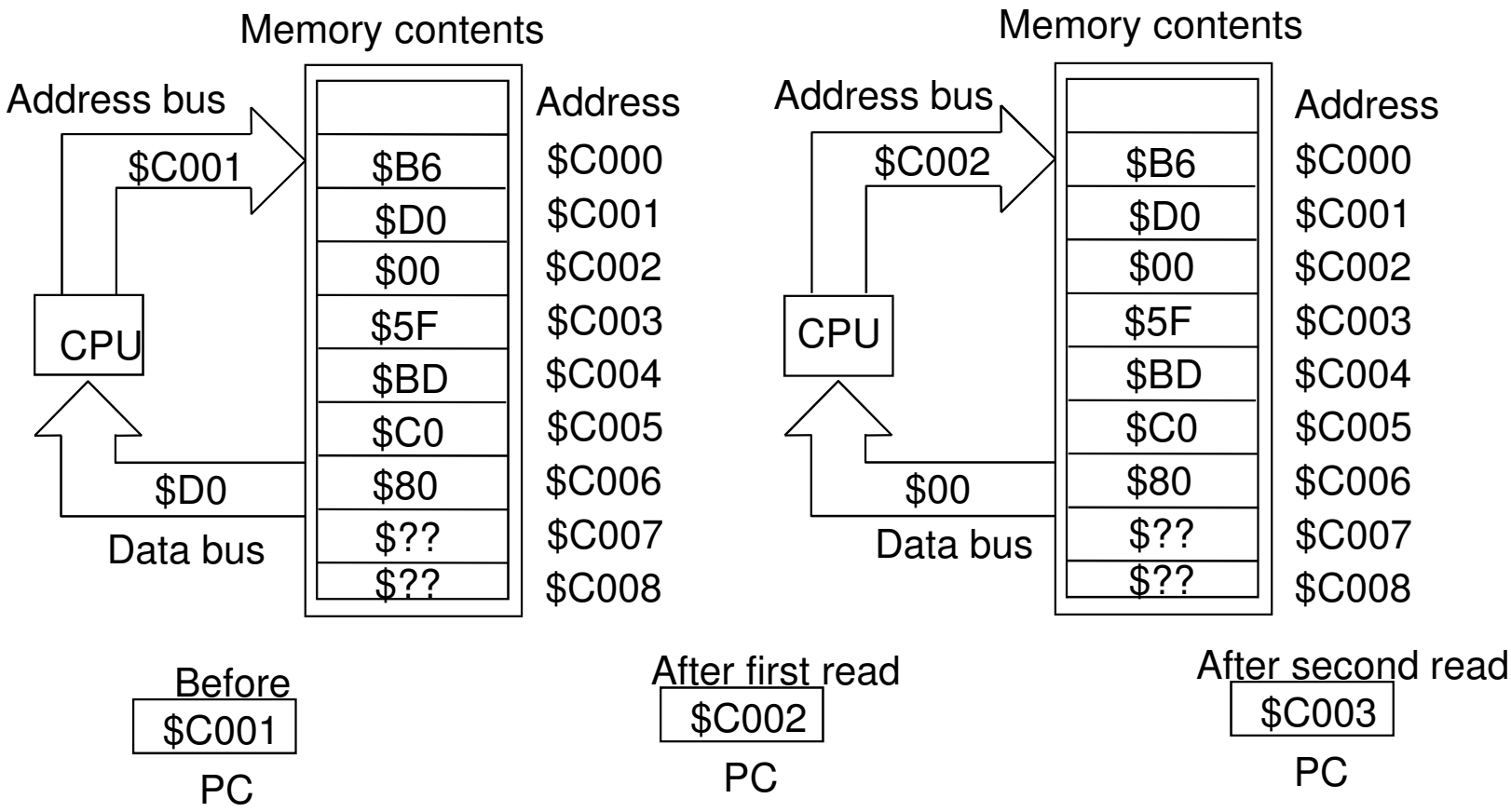
Step 1. Place the value in PC on the address bus with a request to read the contents of that location.

Step 2. The opcode byte **$B6** at $C000 is returned to the CPU and PC is incremented by 1.



Instruction 1 -- Opcode read cycle

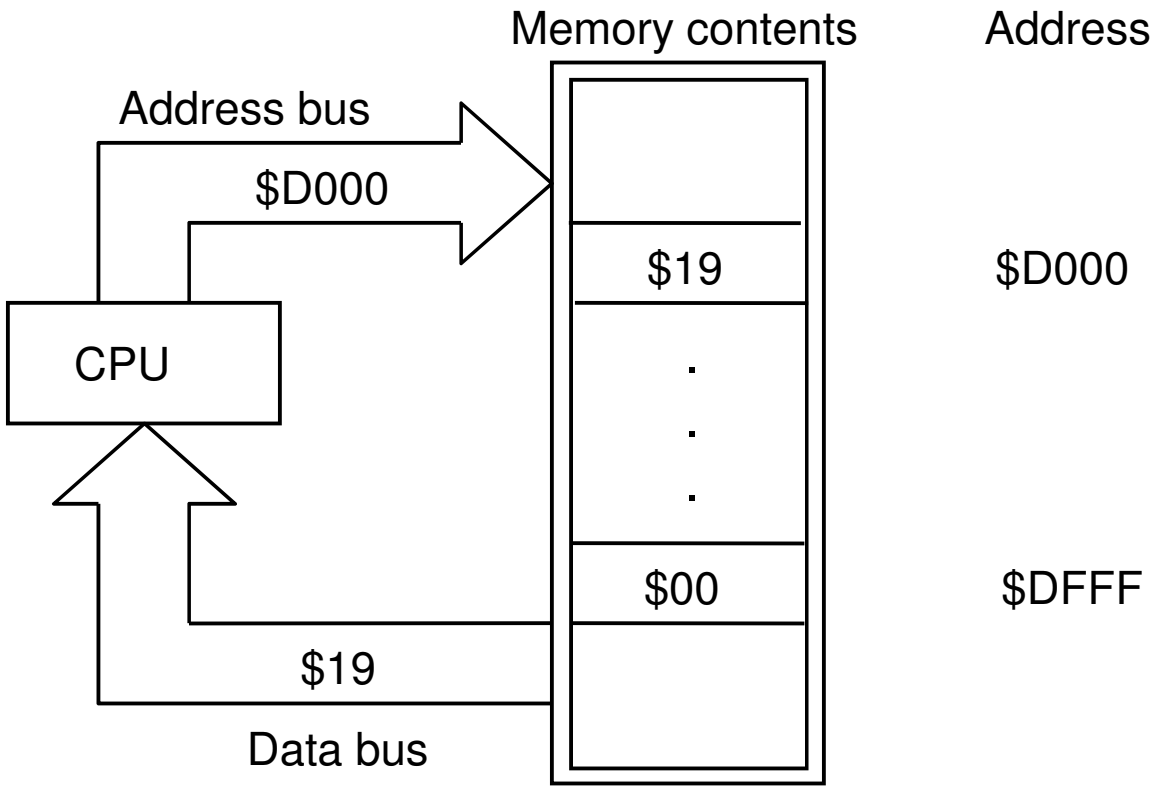
Step 3. CPU performs two read cycles to obtain the extended address $D000 from locations $C001 and $C002. At the end the value of PC is incremented to $C003



Instruction 1 -- Read cycles for address bytes

Step 4. The CPU performs another read to get the contents of the memory location at $D000, which is $19. The value $19 will be loaded into ACCA.

i.e. ACCA ← $19



Instruction 1 -- Operand read cycle

Cycle-by-Cycle Execution:

Cycle	LDAA (EXT)		
	Addr	Data	R/W
1	OP	B6	1
2	OP + 1	hh	1
3	OP + 2	ll	1
4	hhll	(hhll)	1

Cycle-by-cycle Execution – Register Transfer Notation

LDAA $D000

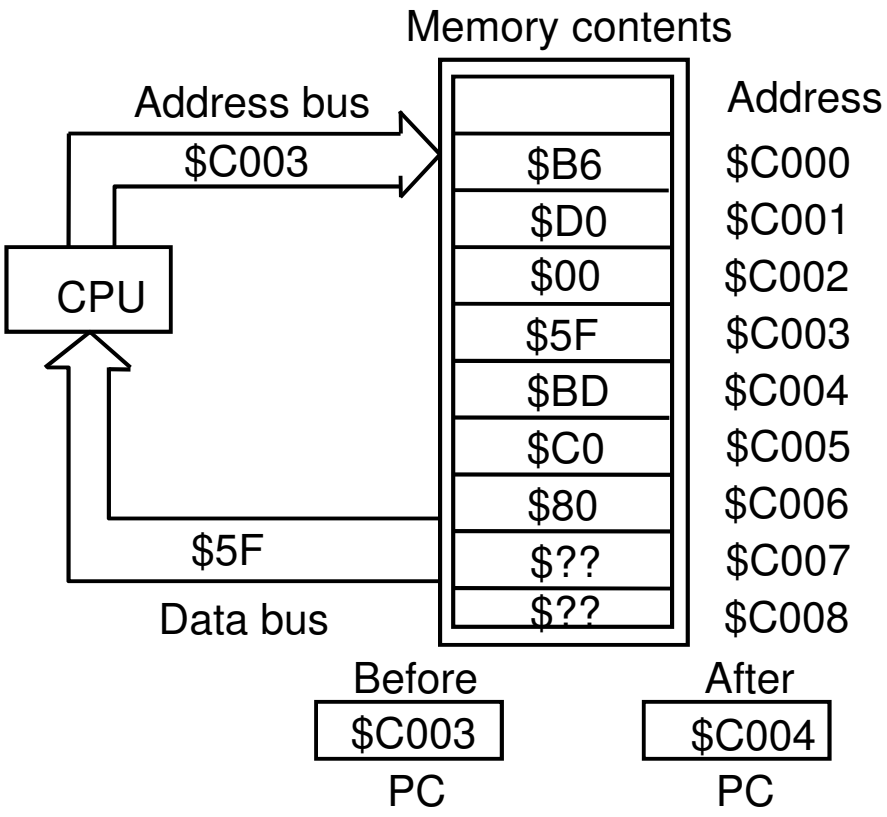
		cycle first half	cycle second half
E #	1	BAR ← <PC> = $C000	PC ← <PC> + 1 = $C001
Code	FOP		IR ← <BDR> = $B6
R/$\overline{W}$	1		
E #	2	BAR ← <PC> = $C001	PC ← <PC> + 1 = $C002
Code	FAHI		$ATMP_{HI}$ ← <BDR> = $D0
R/$\overline{W}$	1		
E #	3	BAR ← <PC> = $C002	PC ← <PC> + 1 = $C003
Code	FALO		$ATMP_{LO}$ ← <BDR> = $00
R/$\overline{W}$	1		
E #	4	BAR ← <ATMP> = $D000	ACCA ← <BDR> = $19
Code	OD		
R/$\overline{W}$	1		

< *reg* > – stands for the contents of register *reg*

Instruction **CLRB**

Step 1. Place the value in PC on the address bus with a request to read the contents of that location.

Step 2. The opcode byte **$5F** at $C003 is returned to the CPU and PC is incremented by 1.



Instruction 2 -- Opcode read cycle

Instruction 2, CLRB -- Execution

Step 3. Once decoded the corresponding action is taken, i.e. ACCB $\leftarrow$ 0.

No operands are read in this Instruction, just a single OpCode byte.
Neither any memory location is written as result of the operation.

Cycle-by-Cycle Execution:

Cycle	CLRB (INH)		
	Addr	Data	R/W
1	OP	5F	1
2	OP + 1	-	1

Cycle-by-cycle Execution – Register Transfer Notation

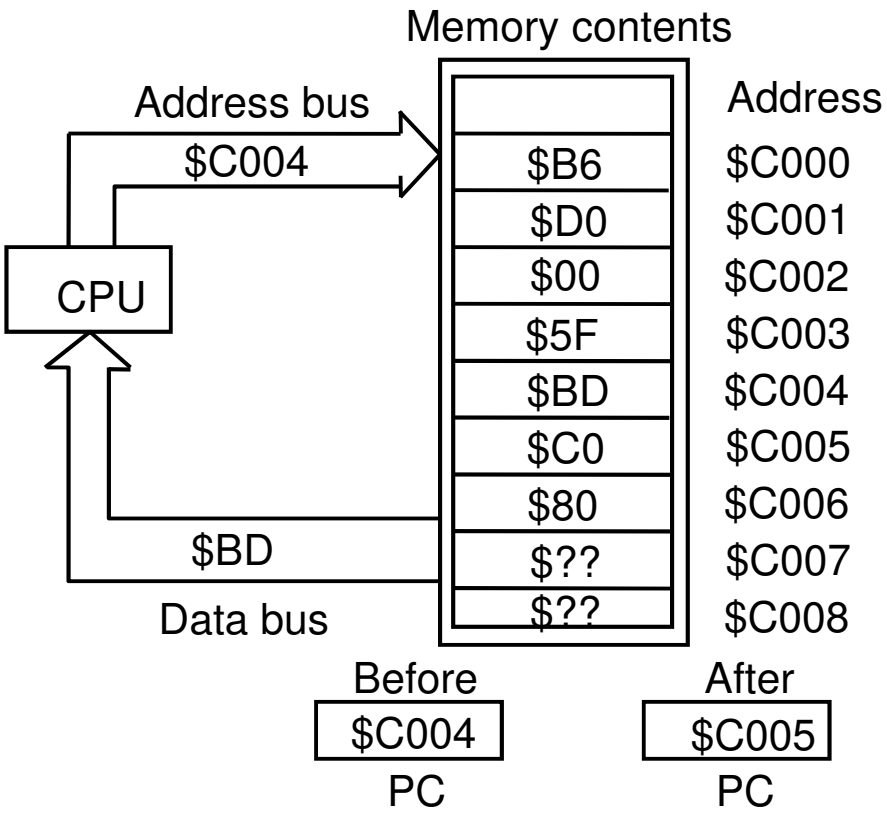
CLRB

E #	1	cycle first half	cycle second half
Code	FOP	BAR ← <PC> = $C003	PC ← <PC> + 1 = $C004
R/$\overline{W}$	1		IR ← <BDR> = $5F
E #	2	cycle first half	cycle second half
Code	EXEC	BAR ← <PC> = $C004	ACCB ← $00
R/$\overline{W}$	1		discard ← <BDR> = $BD

Instruction **JSR $C080**

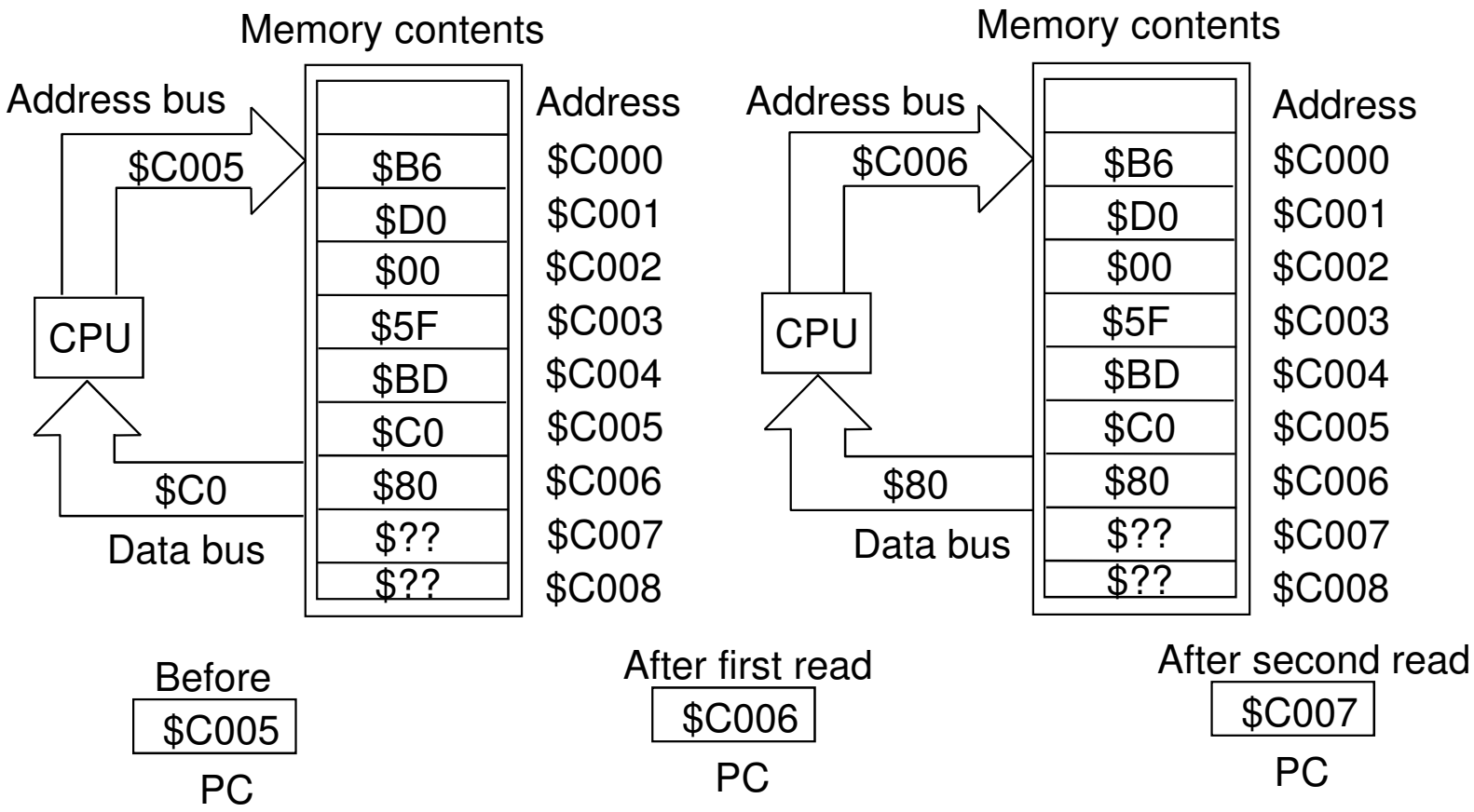
Step 1. Place the value in PC on the address bus with a request to read the contents of that location.

Step 2. The opcode byte **$BD** at $C004 is returned to the CPU and PC is incremented by 1.



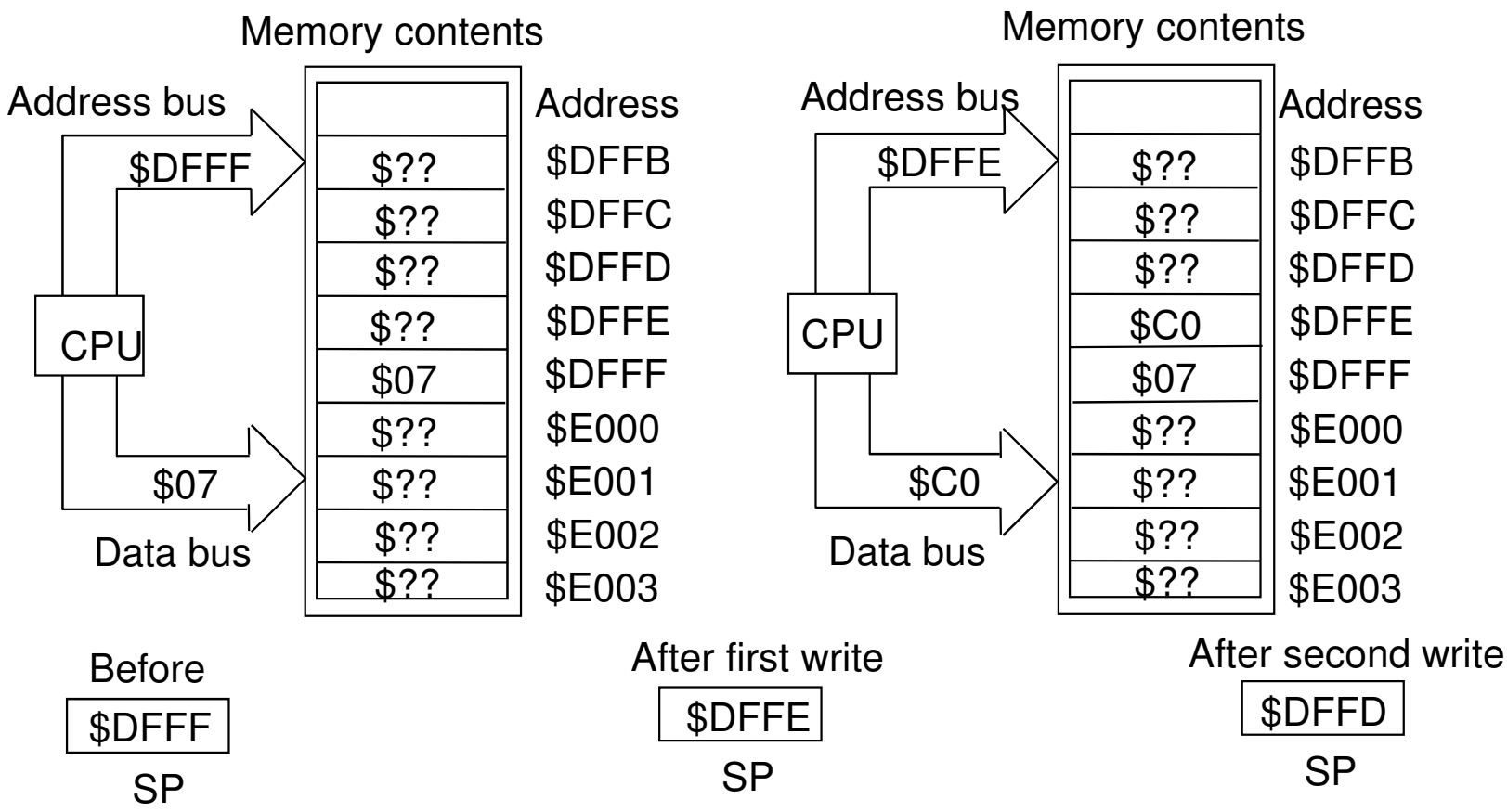
Instruction 3 -- Opcode read cycle

Step 3. CPU performs two read cycles to obtain the extended jump address $C080 from locations $C005 and $C006. At the end the value of PC is incremented to $C007



Instruction 3 -- Address byte read cycles

Step 4. The CPU stores (pushes) the current value of the PC, or return address $C007, onto the Stack. (Assume SP = $DFFF at the time)



Step 5. The CPU assigns the PC the jump address, i.e. PC ← $C080, where program execution continues by fetching an OpCode at that address.

Instruction 3 -- Execution

Cycle-by-Cycle Execution:

Cycle	JSR (EXT)		
	Addr	Data	R/W
1	OP	BD	1
2	OP + 1	hh	1
3	OP + 2	ll	1
4	hhll	(hhll)	1
5	SP	Rtn lo	0
6	SP - 1	Rtn hi	0

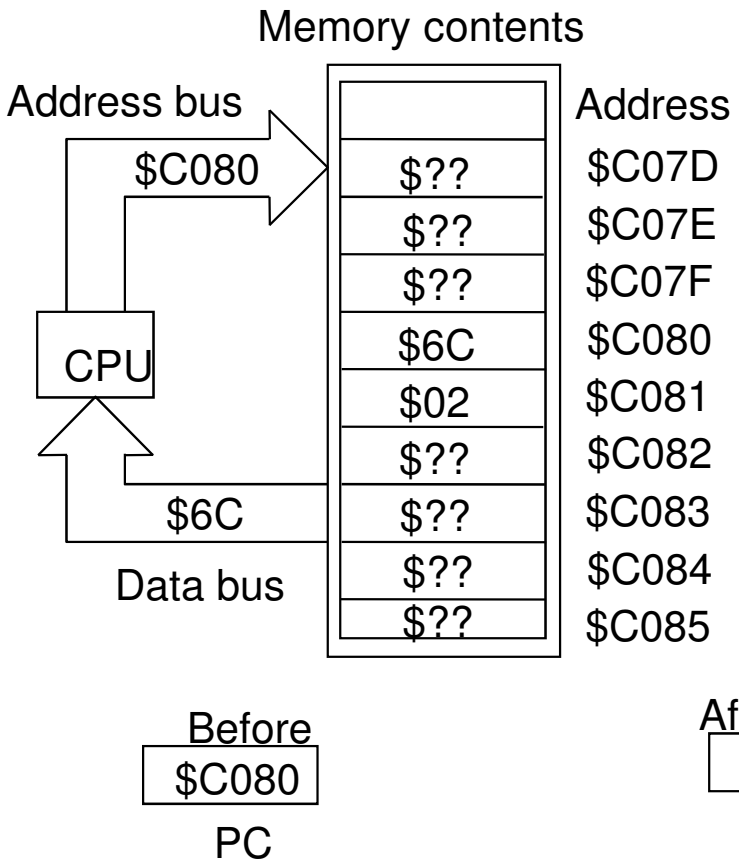
Cycle-by-cycle Execution – Register Transfer Notation

JSR $C080 (SP = $DFFF)

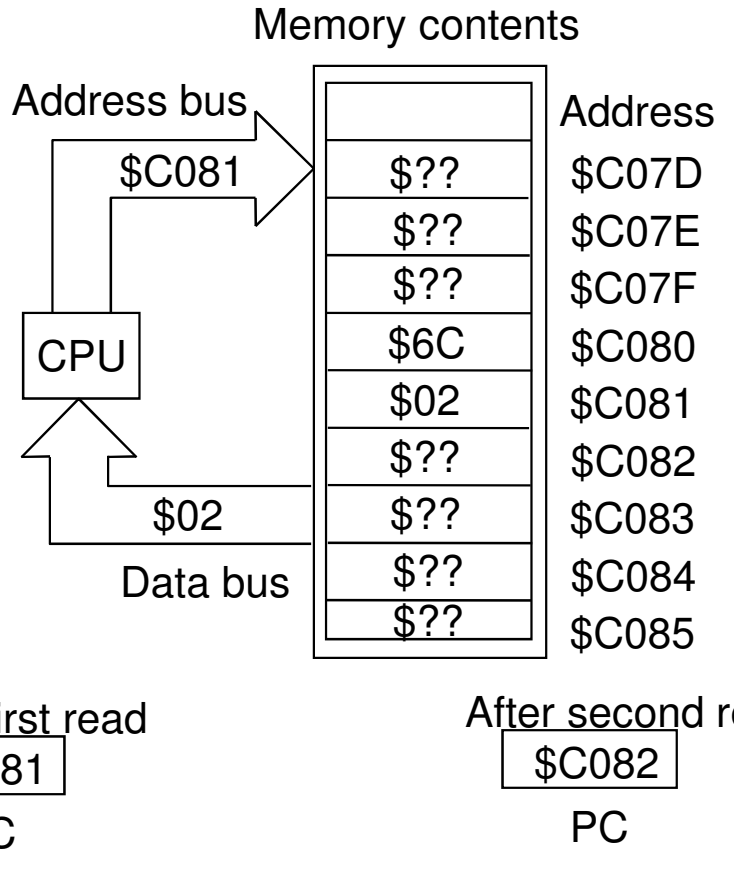
E #	1	cycle first half	cycle second half
Code	FOP	BAR ← <PC> = $C004	PC ← <PC> + 1 = $C005
R/$\overline{W}$	1		IR ← <BDR> = $BD
E #	2	cycle first half	cycle second half
Code	FAHI	BAR ← <PC> = $C005	PC ← <PC> + 1 = $C006
R/$\overline{W}$	1		$ATMP_{HI}$ ← <BDR> = $C0
E #	3	cycle first half	cycle second half
Code	FALO	BAR ← <PC> = $C006	PC ← <PC> + 1 = $C007
R/$\overline{W}$	1		$ATMP_{LO}$ ← <BDR> = $80
E #	4	cycle first half	cycle second half
Code	OD	BAR ← <ATMP> = $C080	$DTMP_{LO}$ ← <BDR> = $6C
R/$\overline{W}$	1	Note: This cycle is implemented for consistency with the EXTended addressing mode likely to make control logic simpler, contents of $C080 NOT USED by this Instruction)	
E #	5	cycle first half	cycle second half
Code	ODLO	BAR ← <SP> = $DFFF	BDR ← <PC_{LO}> = 07
R/$\overline{W}$	0	SP ← <SP> - 1 = $DFFE	
E #	6	cycle first half	cycle second half
Code	ODHI	BAR ← <SP> = $DFFE	BDR ← <PC_{HI}> = C0
R/$\overline{W}$	0	SP ← <SP> - 1 = $DFFD	PC ← <ATMP> = $C080

Instruction **INC 2,X**

Step 1. CPU fetches OpCode byte



Step 2. CPU fetches offset byte

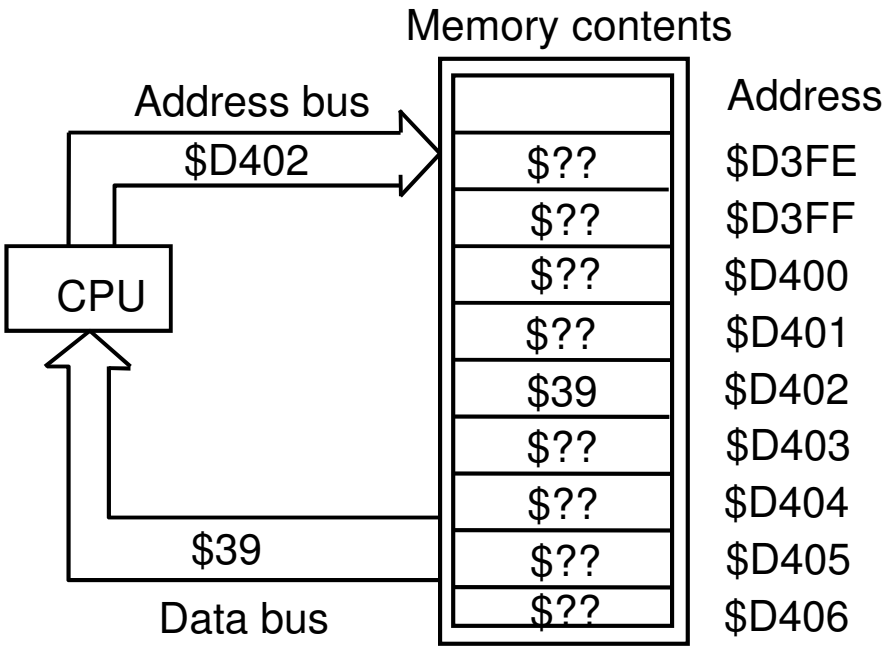


Instruction 4 -- Read cycles for Opcode & Offset

Instruction **INC 2,X**

Step 3. Effective operand address is computed using the ALU. Assuming IX = $D400:

Operand Address = IX + 2 = $D402

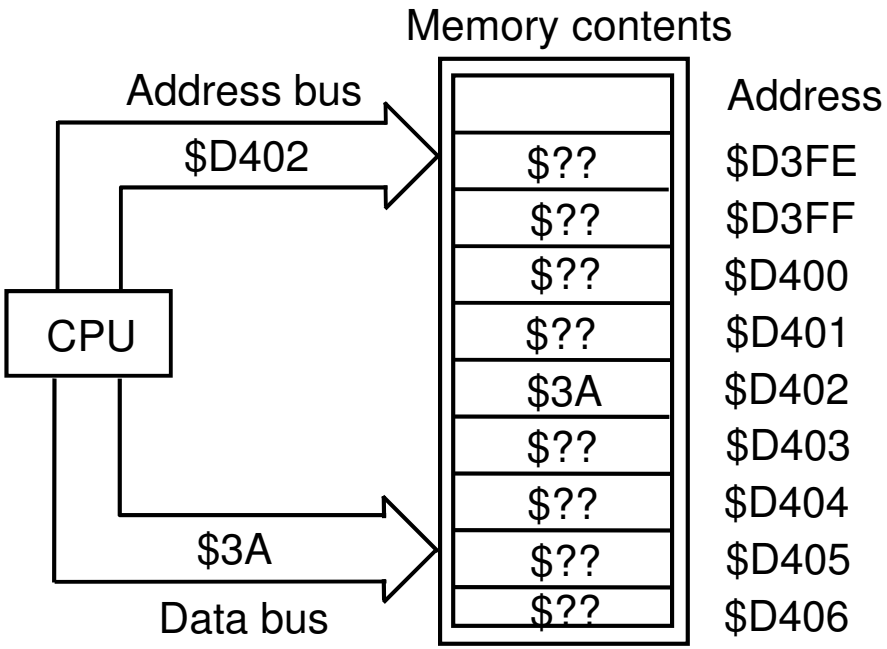


Step 4. The CPU puts out the effective address (**$D402**) of the operand to be read and incremented next. Its value is returned in the data bus(**$39**)

Instruction 4 -- Computing operand address & operand read cycle

Instruction **INC 2,X**

Step 5. Executing the increment operation:
Result = $39 + 1 = $3A



Step 6. Storing the result back to the memory address

Instruction 4 -- Increment & write cycles

Cycle-by-Cycle Execution:

Cycle	INC (IND,X)		
	Addr	Data	R/W
1	OP	6C	1
2	OP + 1	ff	1
3	FFFF	-	1
4	X + ff	(X + ff)	1
5	FFFF	-	1
6	X + ff	result	0

Cycle-by-cycle Execution – Register Transfer Notation

INC 2,X (IX = $D400)

		cycle first half	cycle second half
E #	1	BAR ← <PC> = $C080	PC ← <PC> + 1 = $C081
Code	FOP		IR ← <BDR> = $6C
R/$\overline{W}$	1		
E #	2	BAR ← <PC> = $C081	PC ← <PC> + 1 = $C082
Code	FOFF		$ATMP_{LO}$ ← <BDR> = $02
R/$\overline{W}$	1		$ATMP_{HI}$ ← $00 (*)
E #	3	BAR ← $FFFF	discard ← <BDR> = $xx
Code	CA	$ATMP_{LO}$ ← <IX_{LO}> + <$ATMP_{LO}$>	$ATMP_{HI}$ ← <IX_{HI}> + <$ATMP_{HI}$> + ©
R/$\overline{W}$	1	$ATMP_{LO}$ ← $00+$02 = $02, © = 0	$ATMP_{HI}$ ← $D4 + $00 + 0 = $D4
E #	4	BAR ← <ATMP> = $D402	$DTMP_{LO}$ ← <BDR> = $39
Code	OD		
R/$\overline{W}$	1		
E #	5	BAR ← $FFFF	discard ← <BDR> = $xx
Code	EXEC		$DTMP_{LO}$ ← <$DTMP_{LO}$> + 1
R/$\overline{W}$	1		$DTMP_{LO}$ ← $3A
E #	6	BAR ← <ATMP> = $D402	BDR ← <$DTMP_{LO}$> = $3A
Code	OD		
R/$\overline{W}$	0		

© – ALU carry bit, modified as a result of low half 16-bit addition.

(*) ATMP is sign-extended in preparation for next cycle address computation (CA), which involves a 16-bit addition.

In cases like this of a positive 8-bit offset a $00 byte is used for the high half 16-bit addition, in this case with IX_{HI}.

In cases of a signed 8-bit offset, such as in relative branch instructions, the ALU sign extends $ATMP_{LO}$ (using its MSB, B_7). For instance if the offset was -20_d, ie. $EC, the high byte to be added would be $FF, $ATMP_{HI}$ ← $FF, instead of $00. For $+20_d$, ie. $14, $ATMP_{HI}$ ← $00 is still used.

3-to-8 line decoder/demultiplexer; inverting 74HC/HCT138

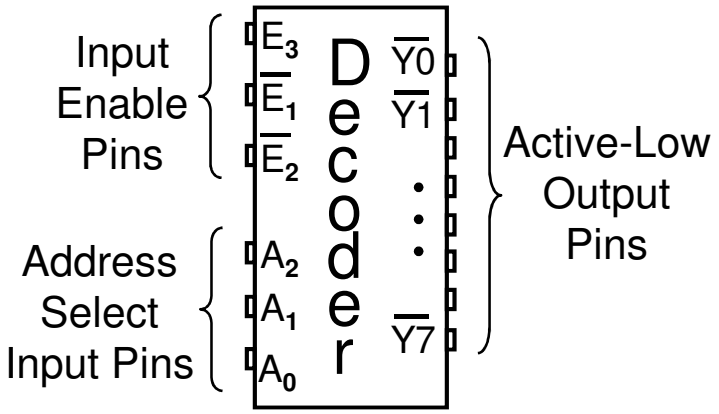
FUNCTION TABLE

INPUTS						OUTPUTS							
$\overline{E}_1$	$\overline{E}_2$	E_3	A_0	A_1	A_2	$\overline{Y}_0$	$\overline{Y}_1$	$\overline{Y}_2$	$\overline{Y}_3$	$\overline{Y}_4$	$\overline{Y}_5$	$\overline{Y}_6$	$\overline{Y}_7$
H	X	X	X	X	X	H	H	H	H	H	H	H	H
X	H	X	X	X	X	H	H	H	H	H	H	H	H
X	X	L	X	X	X	H	H	H	H	H	H	H	H
L	L	H	L	L	L	L	H	H	H	H	H	H	H
L	L	H	H	L	L	H	L	H	H	H	H	H	H
L	L	H	L	H	L	H	H	L	H	H	H	H	H
L	L	H	H	H	L	H	H	H	L	H	H	H	H
L	L	H	L	L	H	H	H	H	H	L	H	H	H
L	L	H	H	L	H	H	H	H	H	H	L	H	H
L	L	H	L	H	H	H	H	H	H	H	H	L	H
L	L	H	H	H	H	H	H	H	H	H	H	H	L

Notes

1. H = HIGH voltage level
   L = LOW voltage level
   X = don't care

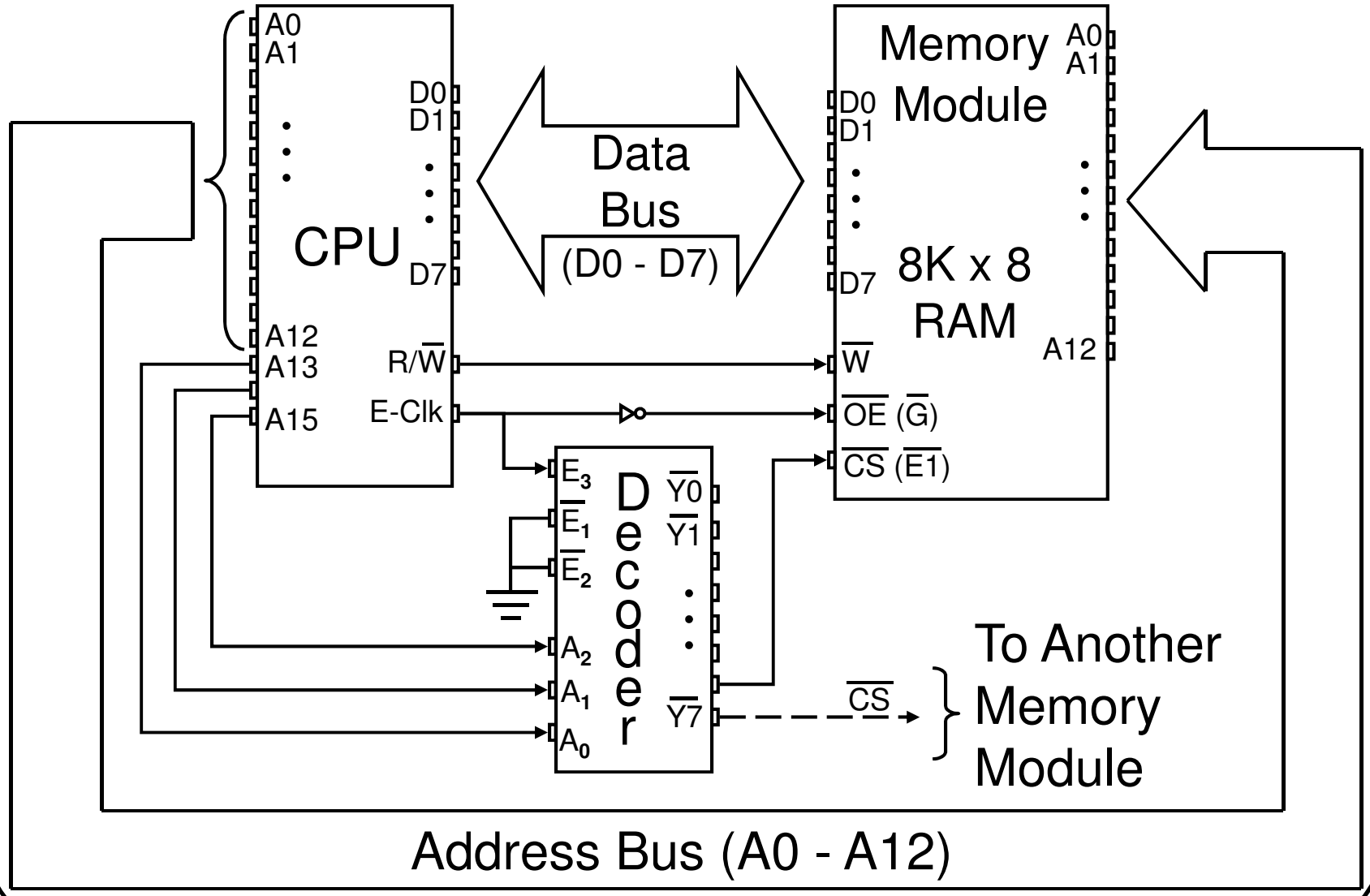
The 74HC138 Decoder



When the chip is enabled, i.e. $\overline{E}_1 = \overline{E}_2 = 0$ and $E_3 = 1$:

A_2	A_1	A_0	Activates Output
0	0	0	$\overline{Y}_0$
0	0	1	$\overline{Y}_1$
0	1	0	$\overline{Y}_2$
0	1	1	$\overline{Y}_3$
1	0	0	$\overline{Y}_4$
1	0	1	$\overline{Y}_5$
1	1	0	$\overline{Y}_6$
1	1	1	$\overline{Y}_7$

Address Decoding Example (EVB case)



Memory Module Allocation using the Decoder

- Lines A_{12} - A_0 are used to address the 8K = 2^{13} locations in the module
- Lines A_{15}-A_{13} specify location of memory module in the address space

A_{15}-A_{12} Min – Max	A_{15} 14 13 12 11 10 9 8 • • • 1 0	Address Range	Active Pin
0000 – 0001	0 0 0 x x x x x • • • x x	$0000 – $1FFF	$\overline{Y}_0$
0010 – 0011	0 0 1 x x x x x • • • x x	$2000 – $3FFF	$\overline{Y}_1$
0100 – 0101	0 1 0 x x x x x • • • x x	$4000 – $5FFF	$\overline{Y}_2$
0110 – 0111	0 1 1 x x x x x • • • x x	$6000 – $7FFF	$\overline{Y}_3$
1000 – 1001	1 0 0 x x x x x • • • x x	$8000 – $9FFF	$\overline{Y}_4$
1010 – 1011	1 0 1 x x x x x • • • x x	$A000 – $BFFF	$\overline{Y}_5$
1100 – 1101	1 1 0 x x x x x • • • x x	$C000 – $DFFF	$\overline{Y}_6$
1110 – 1111	1 1 1 x x x x x • • • x x	$E000 – $FFFF	$\overline{Y}_7$

Note: 8 memory modules 8K each could be allocated in the 64K address space

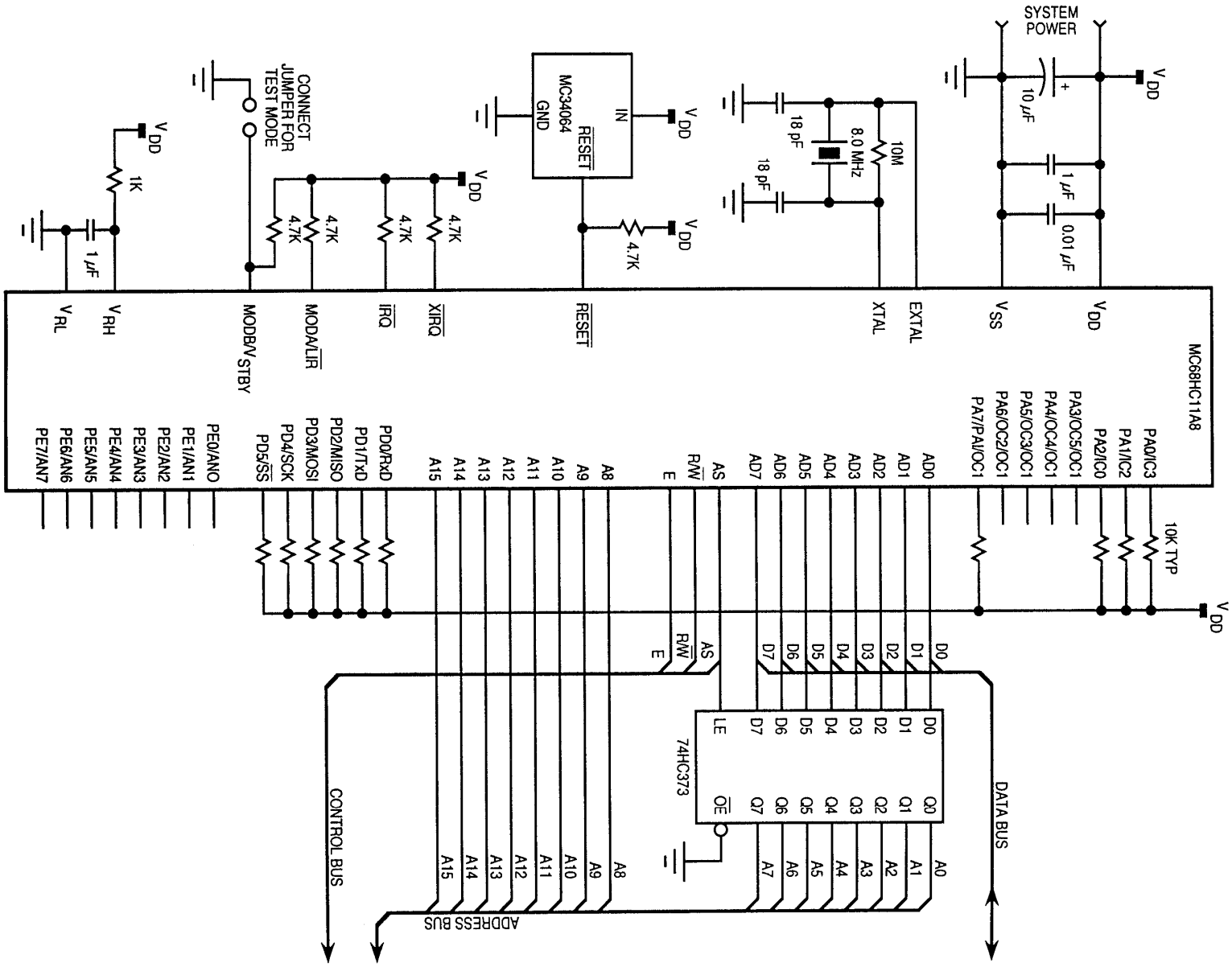


Figure 2-22. Basic Expanded Mode Connections (Sheet 1 of 2)

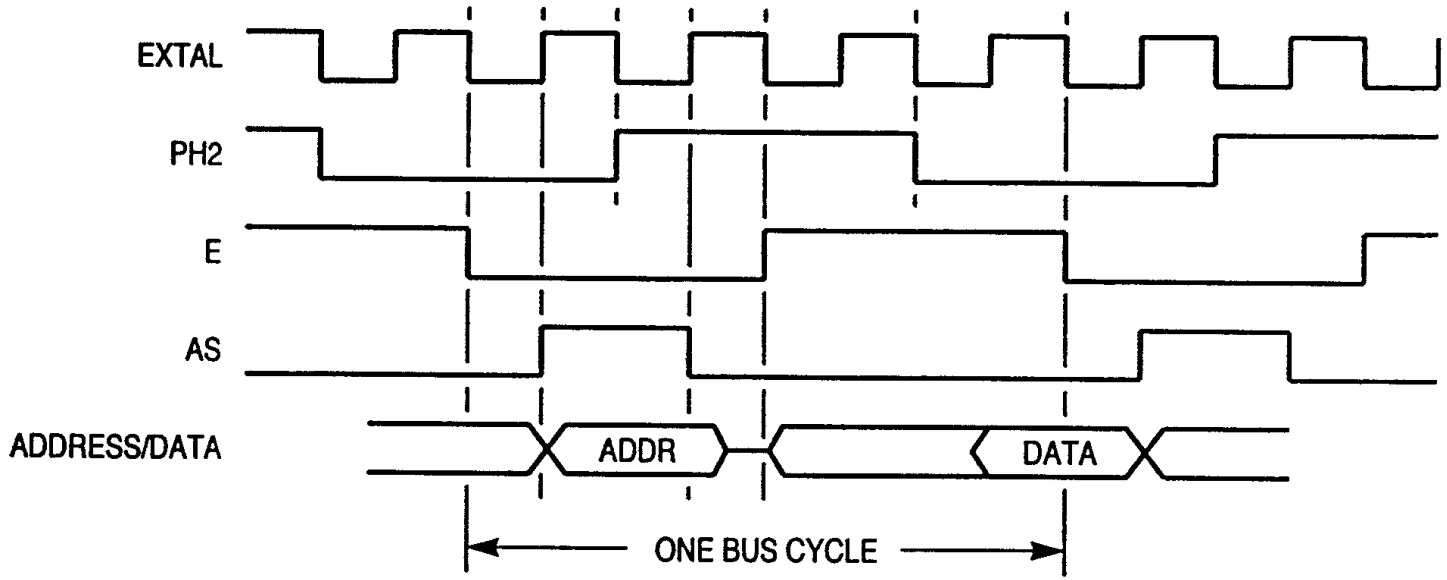
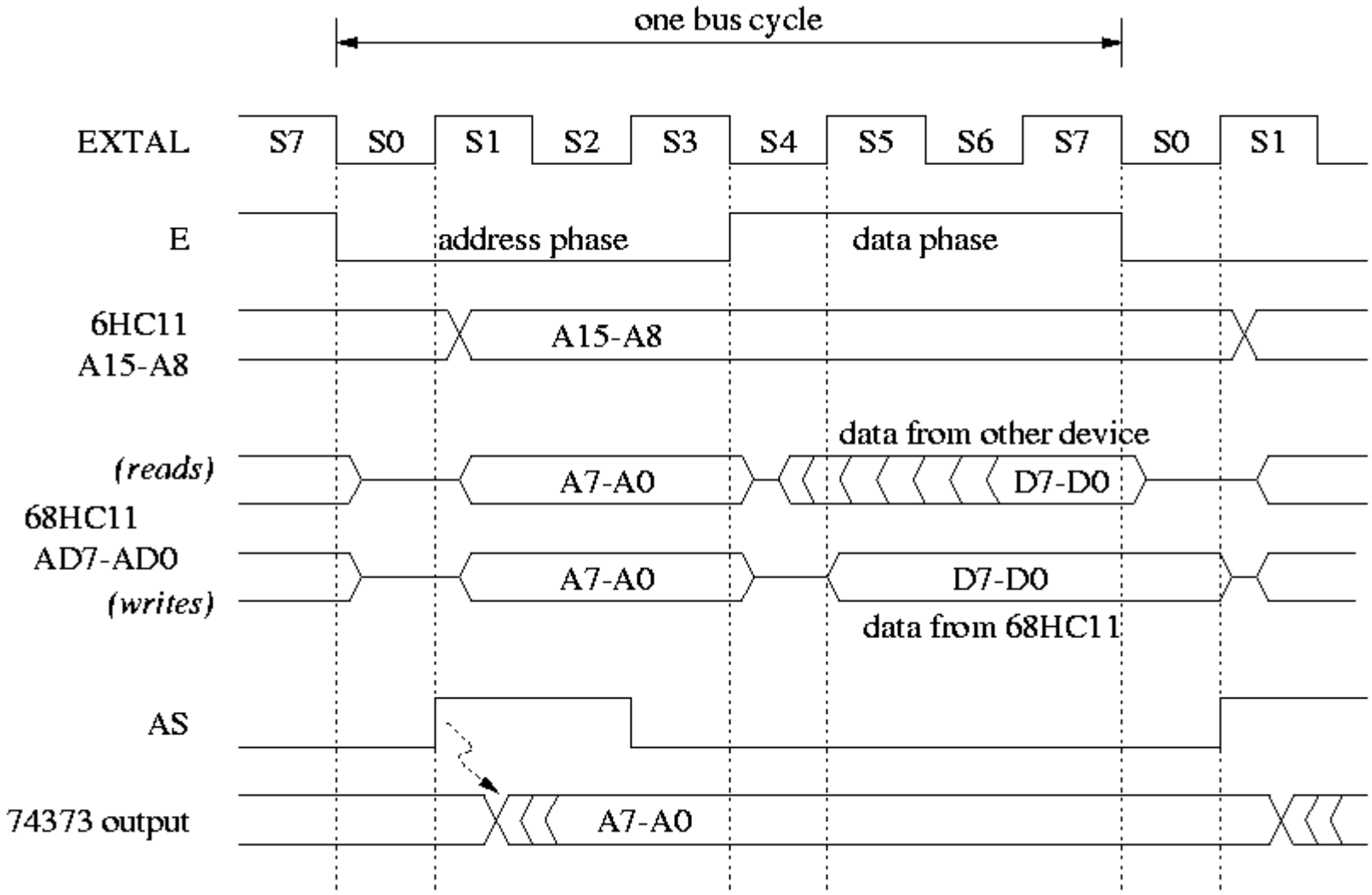
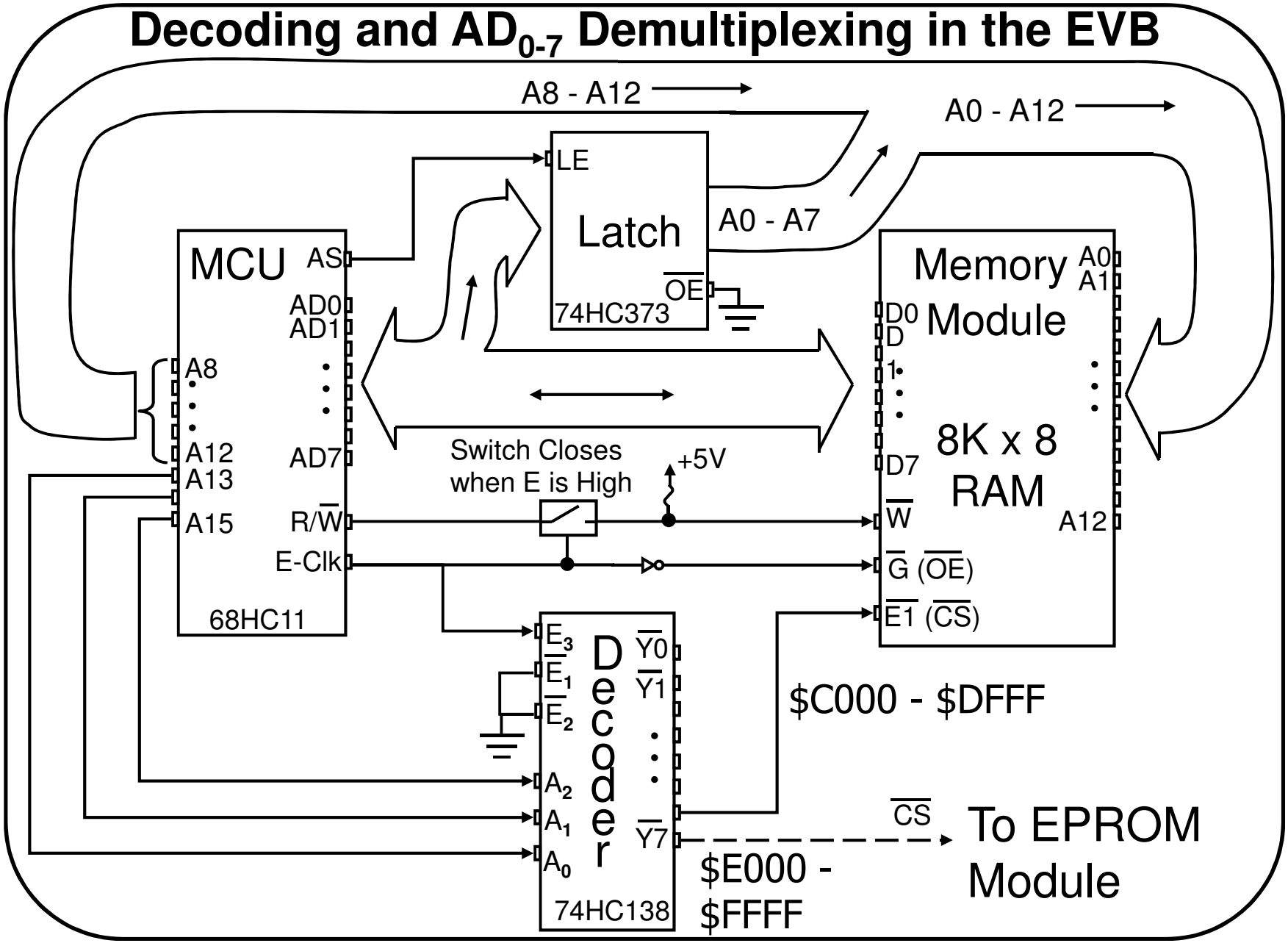


Figure 10-2 Timing Summary for Oscillator Divider Signals


one bus cycle
EXTAL
S7
S0
S1
S2
S3
S4
S5
S6
S7
S0
S1
E
address phase
data phase
6HC11
A15-A8
A15-A8
data from other device
(reads)
A7-A0
D7-D0
68HC11
AD7-AD0
(writes)
A7-A0
D7-D0
data from 68HC11
AS
74373 output
A7-A0

Decoding and AD_{0-7} Demultiplexing in the EVB



EVB Schematic Diagram (Sheet 1)

NOTES:

1. UNLESS OTHERWISE SPECIFIED:

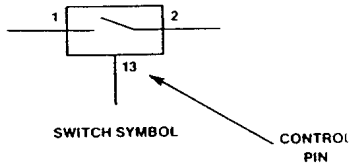
   ALL RESISTORS ARE IN OHMS, ±5%, 1/4W.
   ALL CAPACITORS ARE IN μF.
   ALL VOLTAGES ARE DC.

2. DEVICE TYPE NUMBERS LISTED BELOW IS FOR REFERENCE ONLY. DEVICE TYPE NUMBER VARIES WITH MANUFACTURER.

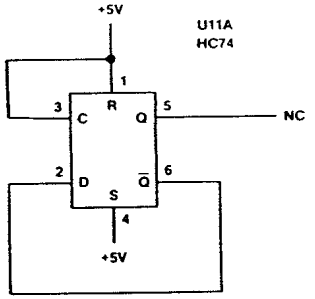
REF DES	DEVICE TYPE	NOTES	GND	+5V	+12V	-12V
U1	MC68HC24	PRU	29	17		
U2	MC74HC373	TRANSPARENT LATCH	10	20		
U3	2764	8K EPROM (250ns)	14	28		
U4	(USER SUPPLIED)	8K RAM (250ns)	14	28		
U5	MCM6164	8K RAM (250ns)	14	28		
U6	MC74HC138	DECODER/DEMUX	8	16		
U7	MC74HC4066	DIGITAL SWITCH	7	14		
U8	MC1488P	RS-232C DRIVER	7		14	1
U9	MC68B50P	ACIA	1	12		
U10	MC68HC11A1	MCU	1	26		
U11	MC74HC74	D-TYPE FLIP-FLOP	7	14		
U12	MC74HC14	INVERTER	7	14		
U13	MC74HC4040	RIPPLE COUNTER	8	16		
U14	MC1489P	RS-232C RECEIVER	7		14	

3. DIGITAL SWITCH U7 OPERATES AS FOLLOWS:

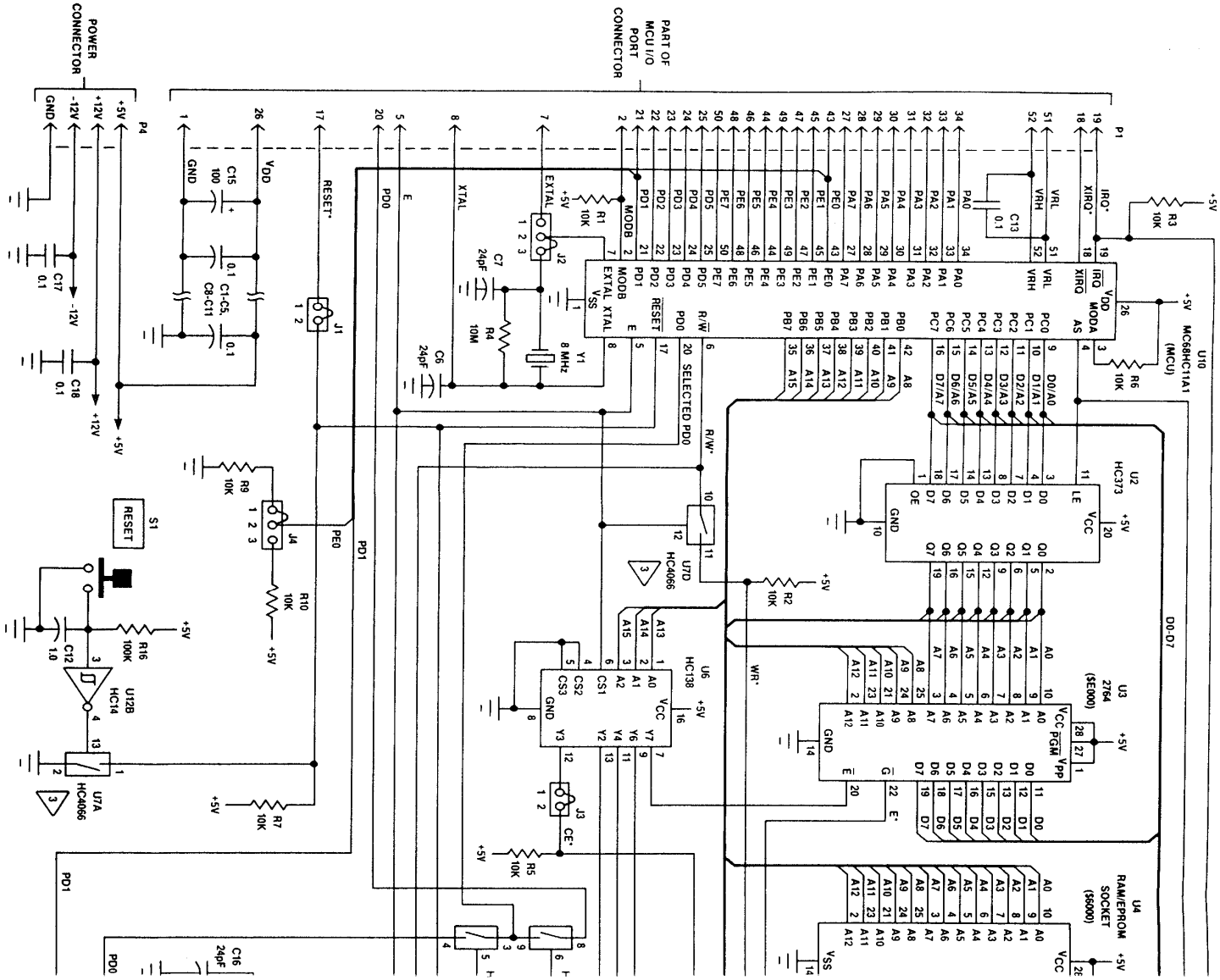
CONTROL PINS (5, 6, 12, OR 13)	SWITCH OPERATION
LOGIC ZERO (0)	OFF/OPEN
LOGIC ONE (1)	ON/CLOSED



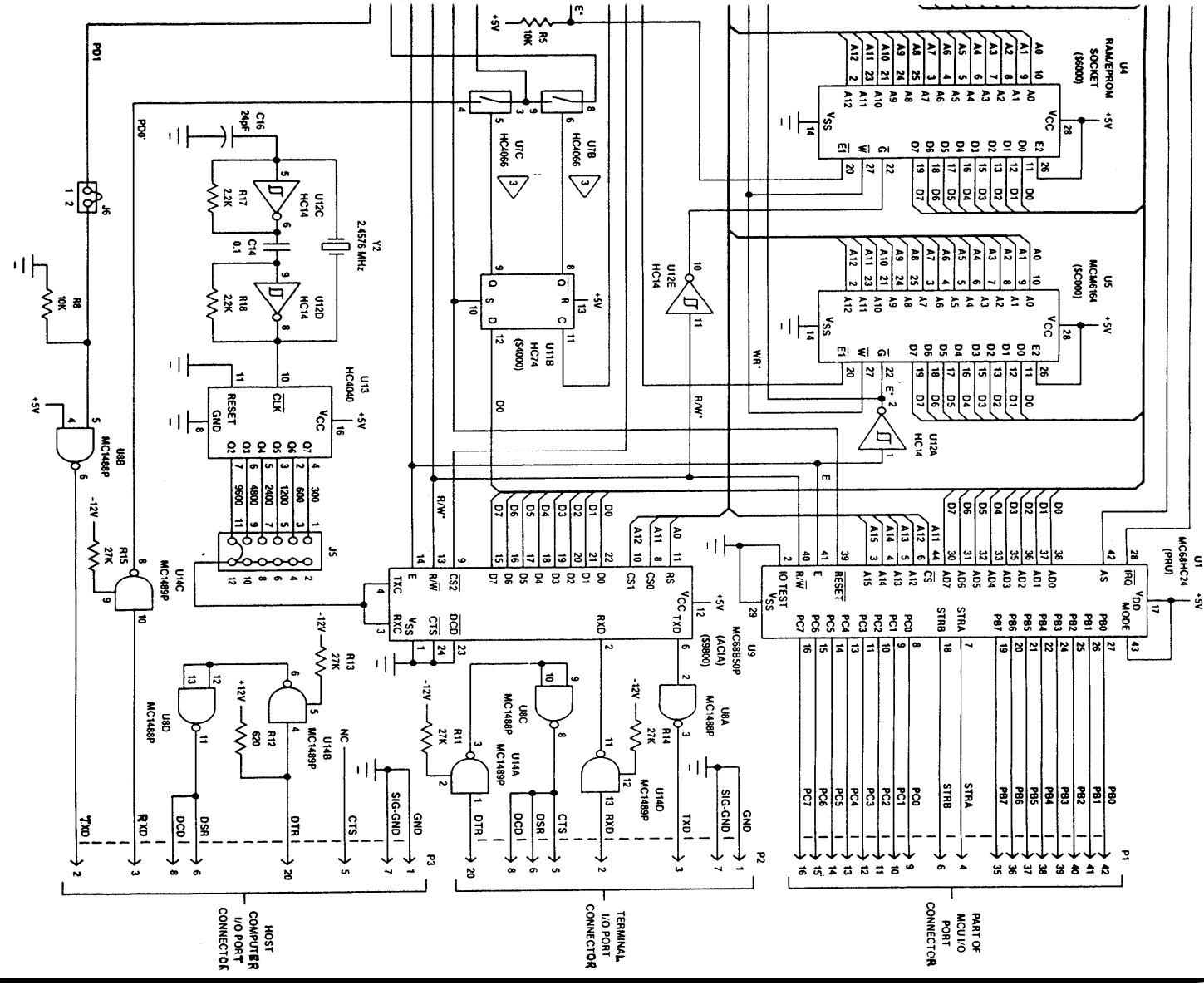
4. NOT USED DEVICE.

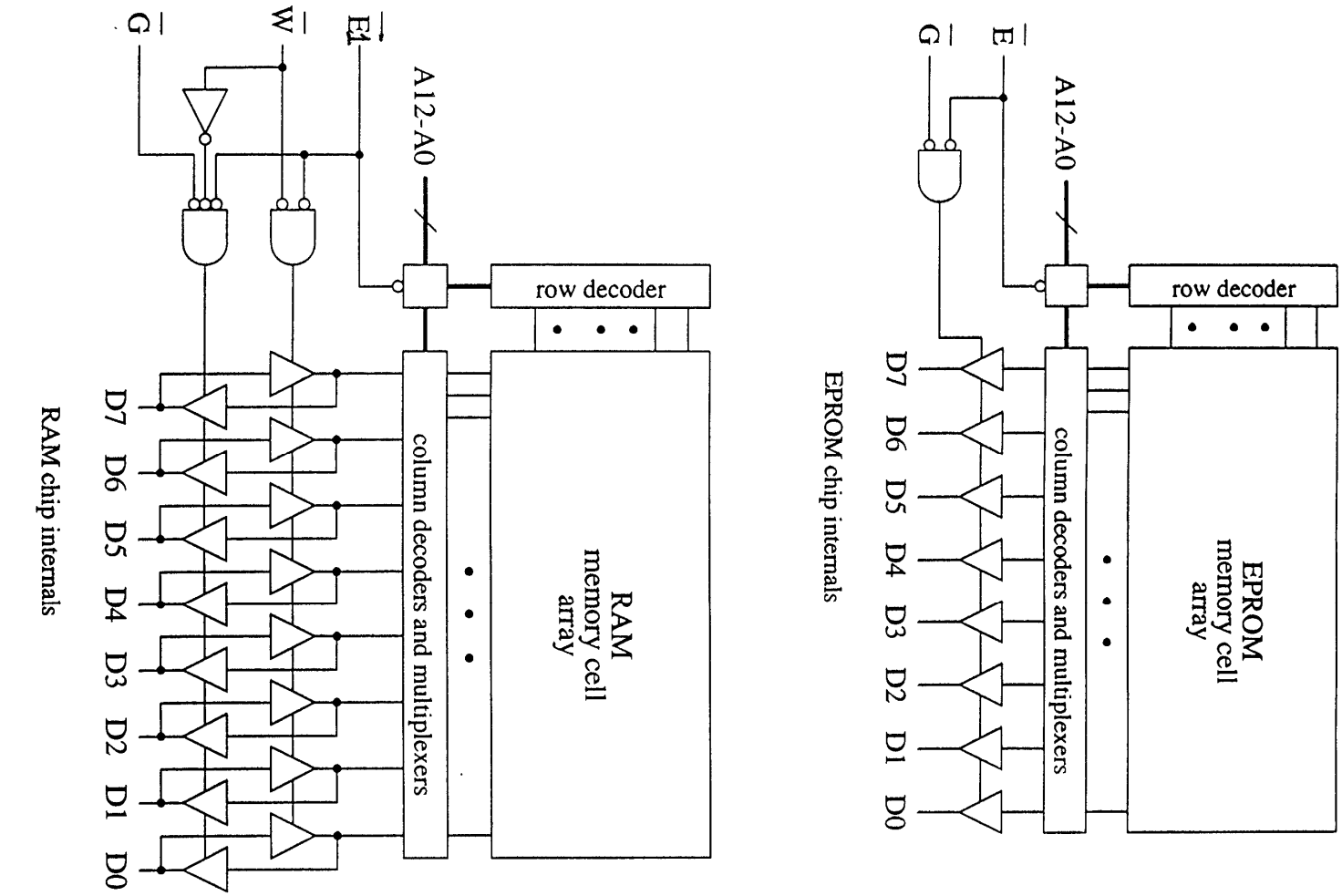


EVB Schematic Diagram (Sheet 2, Part 1)

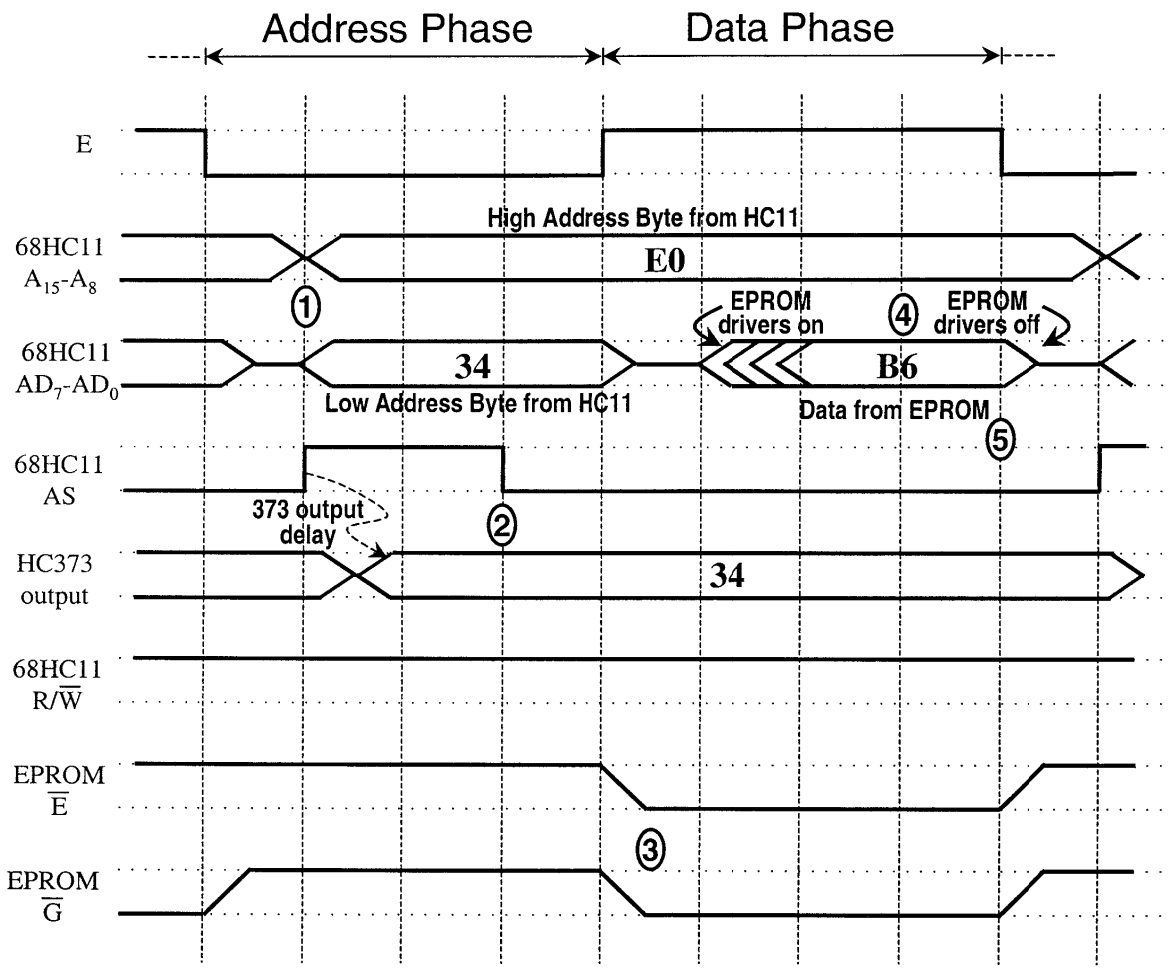


EVB Schematic Diagram (Sheet 2, Part 2)



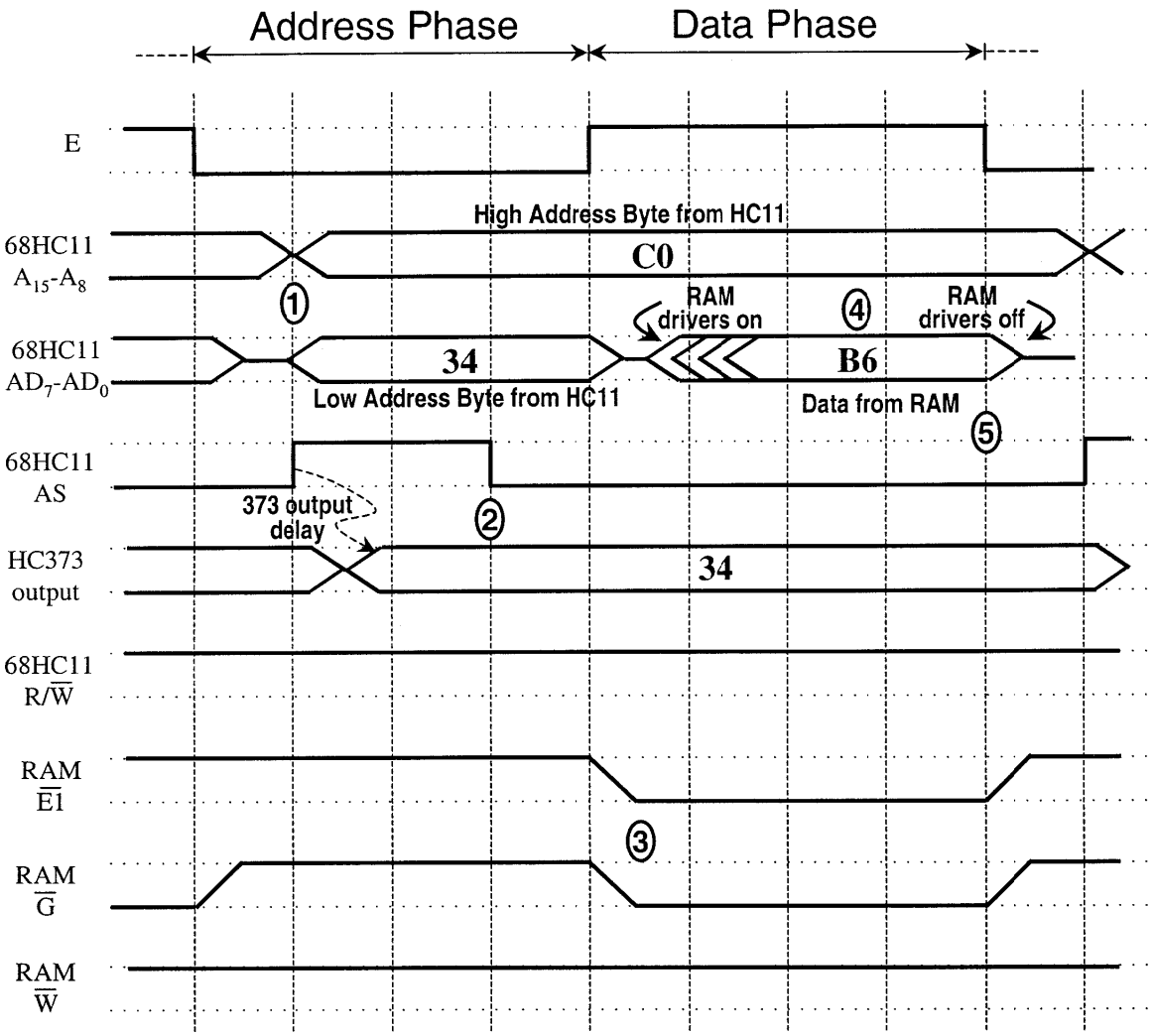

row decoder
EPROM
memory cell
array
column decoders and multiplexers
A12-A0
G
E
D7 D6 D5 D4 D3 D2 D1 D0
EPROM chip internals
row decoder
RAM
memory cell
array
column decoders and multiplexers
A12-A0
G
W
E1
D7 D6 D5 D4 D3 D2 D1 D0
RAM chip internals

EPROM Read Cycle



1. The 68HC11 issues the address, *e.g.* $E034. High address byte ($E0) in lines A_{15} - A_8 and low address byte ($34) in lines AD_7 - AD_0.
2. The low address byte is latched by the 74HC373 with the falling edge of AS before E goes high starting the data phase (A_7 - A_0 is now supplied by the 74HC373).
3. $\overline{Y7}$ is asserted by the decoder (the 74HC138) selecting the EPROM module. EPROM outputs are also enabled ($\overline{G}$ goes low as well) and EPROM drivers turn on.
4. Valid data ($B6) is driven by the EPROM into the AD_7 - AD_0 bus (serving as data bus in this second half).
5. The data bus value (*e.g.* $B6) is latched by the CPU into one its registers with the falling edge of the E clock.

RAM Read Cycle



① The 68HC11 issues the address, *e.g.* $C034. High address byte ($C0) in lines A_{15} - A_8 and low address byte ($34) in lines AD_7 - AD_0.

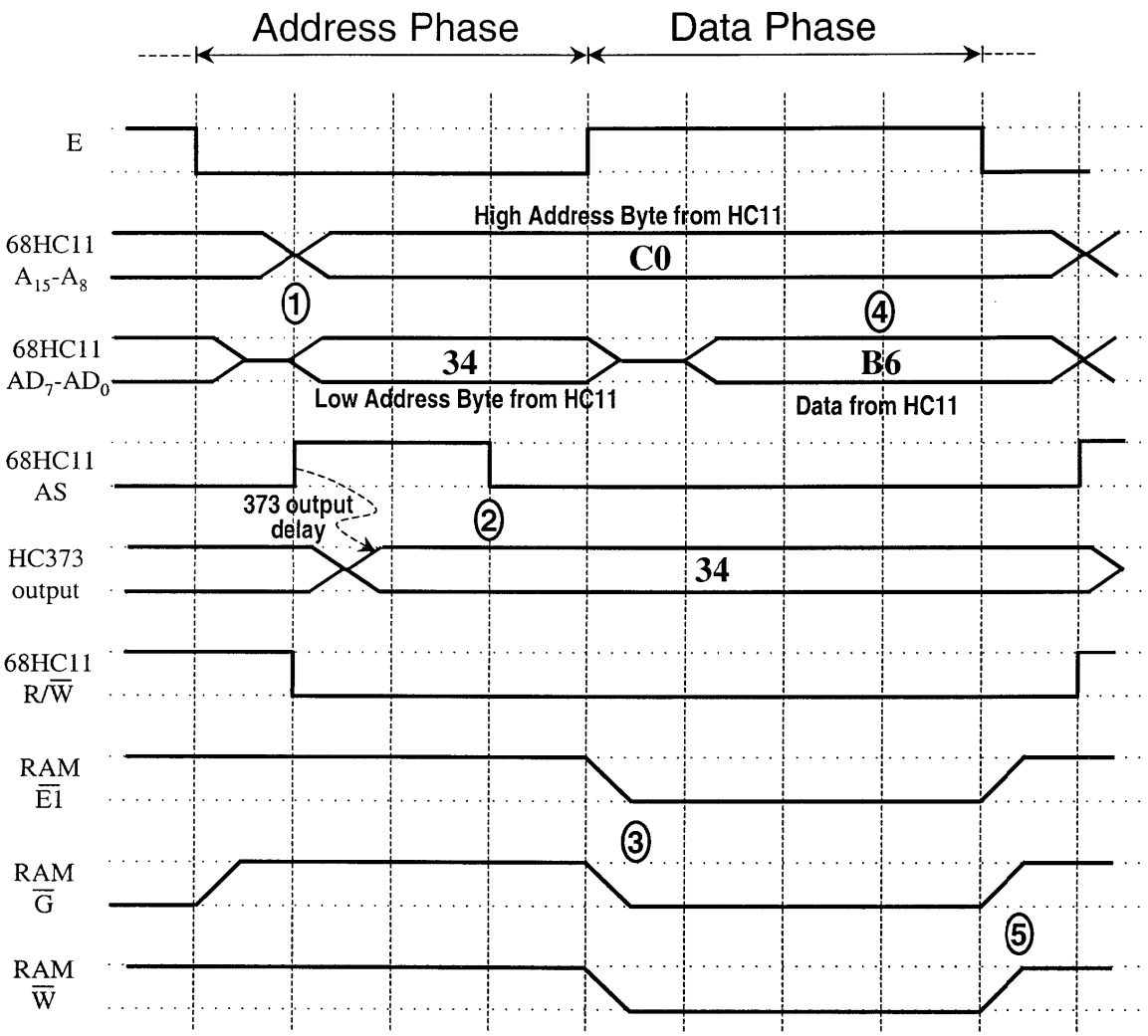
② The low address byte is latched by the 74HC373 with the falling edge of AS before E goes high starting the data phase.

③ $\overline{Y6}$ is asserted by the decoder (the HC138) selecting the RAM module. RAM outputs are also enabled ($\overline{G}$ goes low and $\overline{W}$ remains high since the 68HC11 R/$\overline{W}$ is high).

④ Valid data ($B6) is driven by the RAM into the AD_7 - AD_0 bus.

⑤ The data bus value (*e.g.* $B6) is latched by the CPU into one its registers with the falling edge of the E clock.

RAM Write Cycle



1. The 68HC11 issues the address, *e.g.* $C034. High address byte ($C0) in lines A_{15} - A_8 and low address byte ($34) in lines AD_7 - AD_0. At the same time, the 68HC11 R/$\overline{W}$ signal goes low indicating this is a write cycle since write (W) is active low.
2. The low address byte is latched by the 74HC373 with the falling edge of AS before E goes high starting the data phase.
3. $\overline{Y6}$ is asserted by the decoder (the 74HC138) selecting the RAM module. RAM input buffers are enabled since the HC11 R/$\overline{W}$ is gated to the RAM $\overline{W}$ with the rising of E.
4. Valid data ($B6) is driven by the HC11 into the AD_7 - AD_0 bus.
5. The writing of the data bus value ($B6) to the corresponding memory location ($C034) is completed once $\overline{Y6}$ (RAM $\overline{E1}$) and RAM $\overline{W}$ are deasserted (once they go back high).

Drawing a Timing Diagram

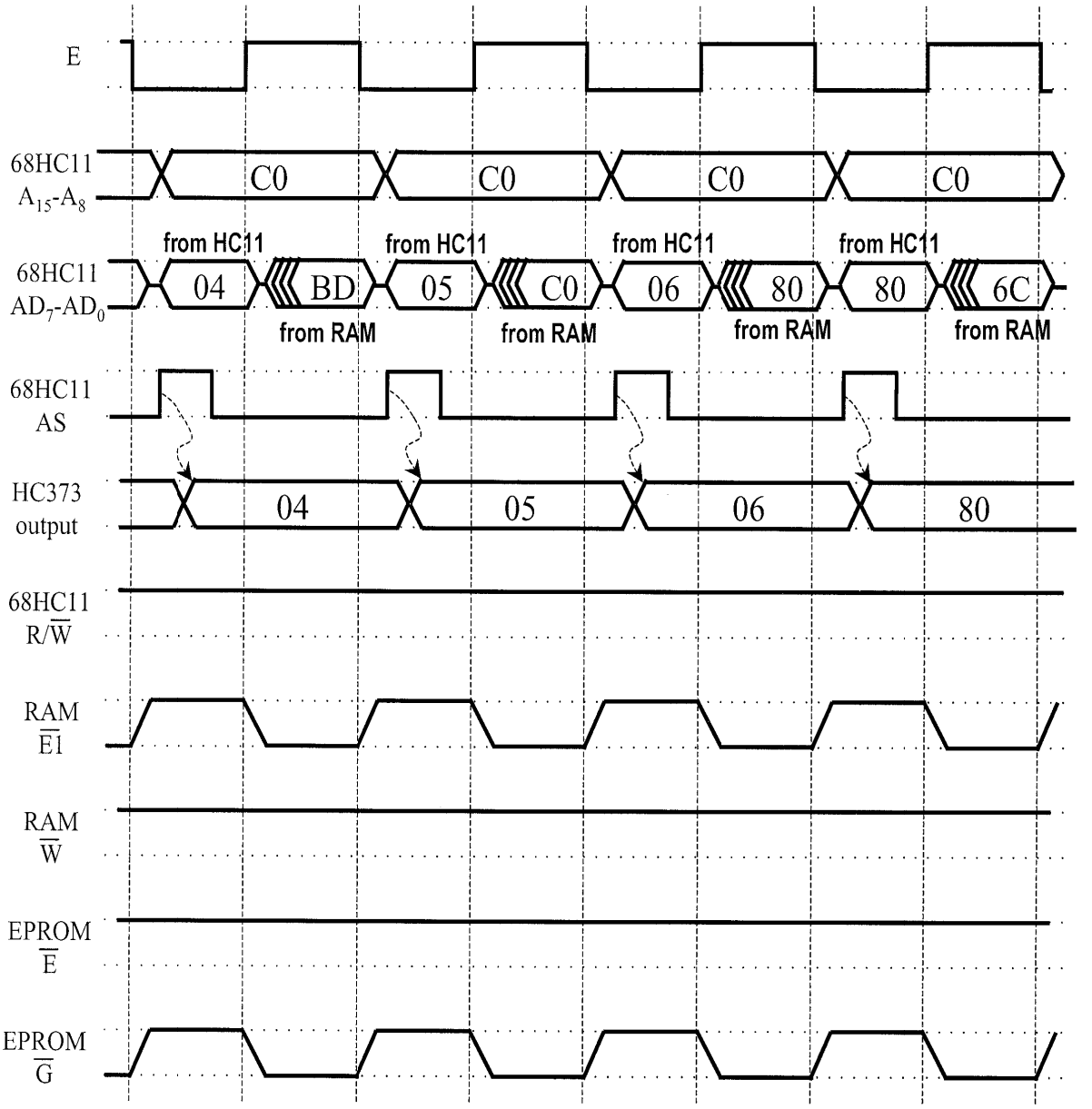
1. Identify the Cycle Type
   a) EPROM Read,
   b) RAM Read, OR
   c) RAM Write.
2. Find the Chip driving the AD_{7-0} lines
   a) Address phase : HC11,
   b) Data phase : HC11, EPROM OR RAM.
3. Include hex values for each bus

   i.e. • $A_{15} - A_8$,
   • $AD_7 - AD_0$, AND
   • The 373 Latch Output ($A_7 - A_0$).

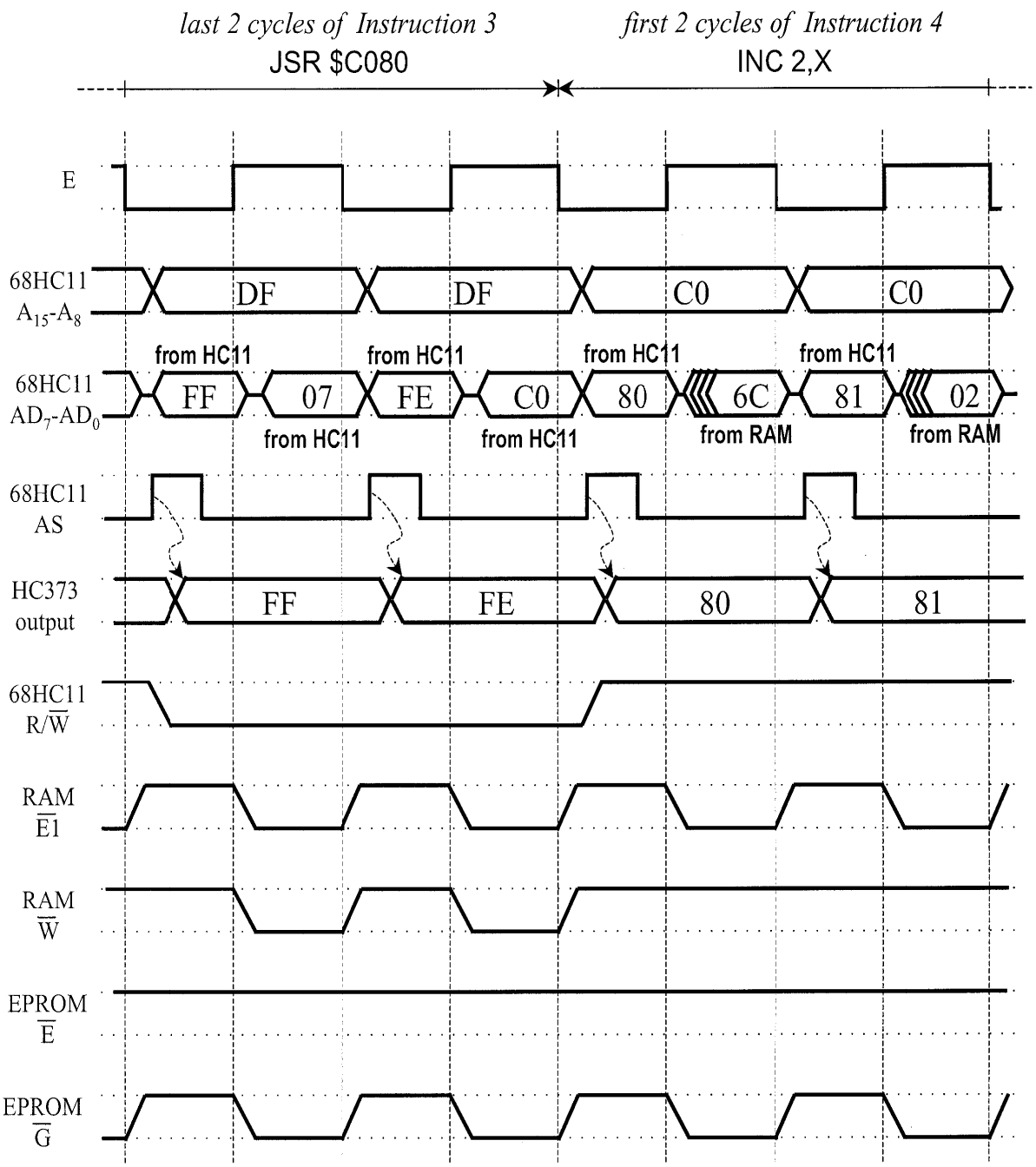
Timing Diagram Example

first 4 cycles of Instruction 3

JSR $C080



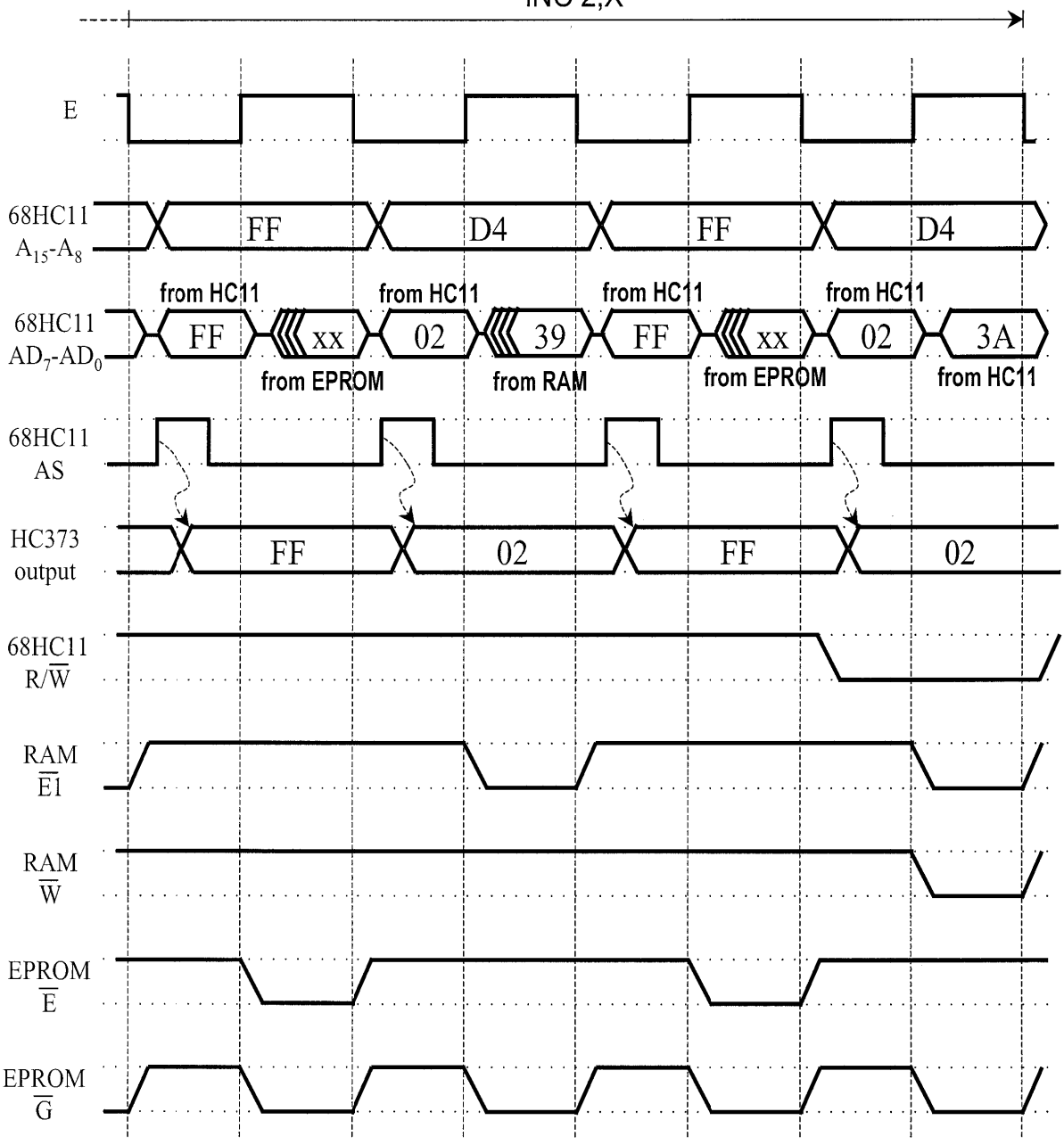
Timing Diagram Example



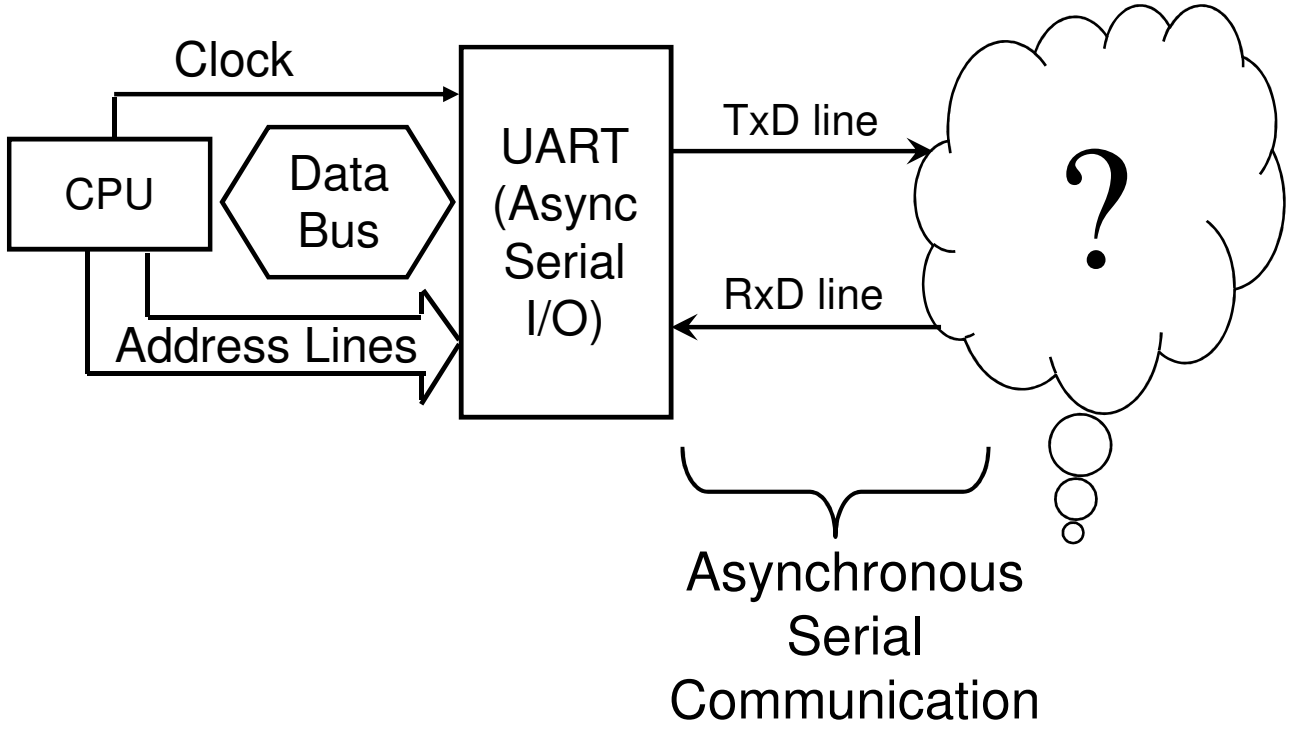
Timing Diagram Example

last 4 cycles of Instruction 4

INC 2,X



Definitions

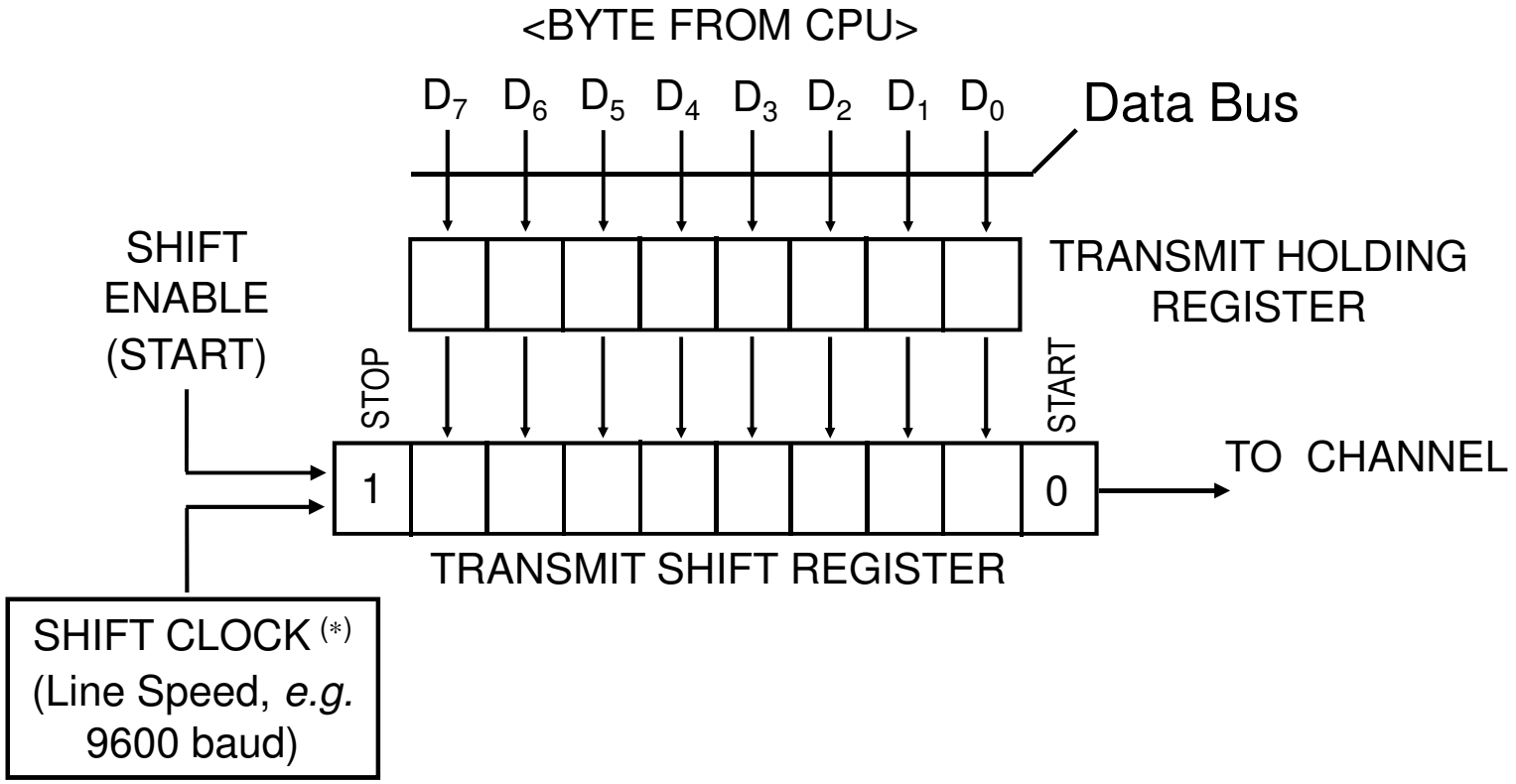


- Asynchronous: There is no clock to establish a time reference
- Serial: Data is carried over the channel one bit at a time,

  not in parallel as over the Data Bus with the CPU

UART .- Transmitter Unit

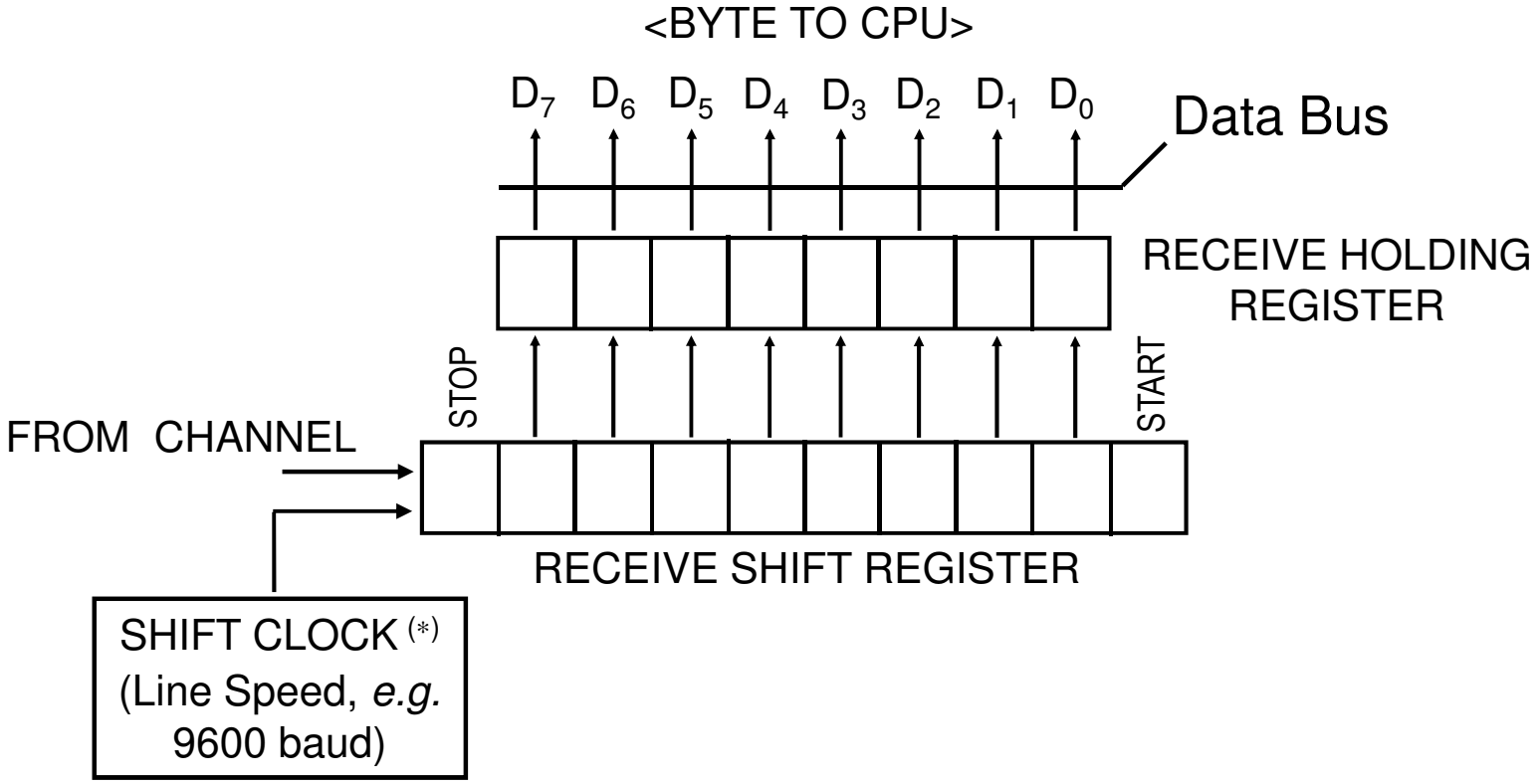
Parallel-to-Serial Conversion



(*) Also known as the BAUD RATE CLOCK

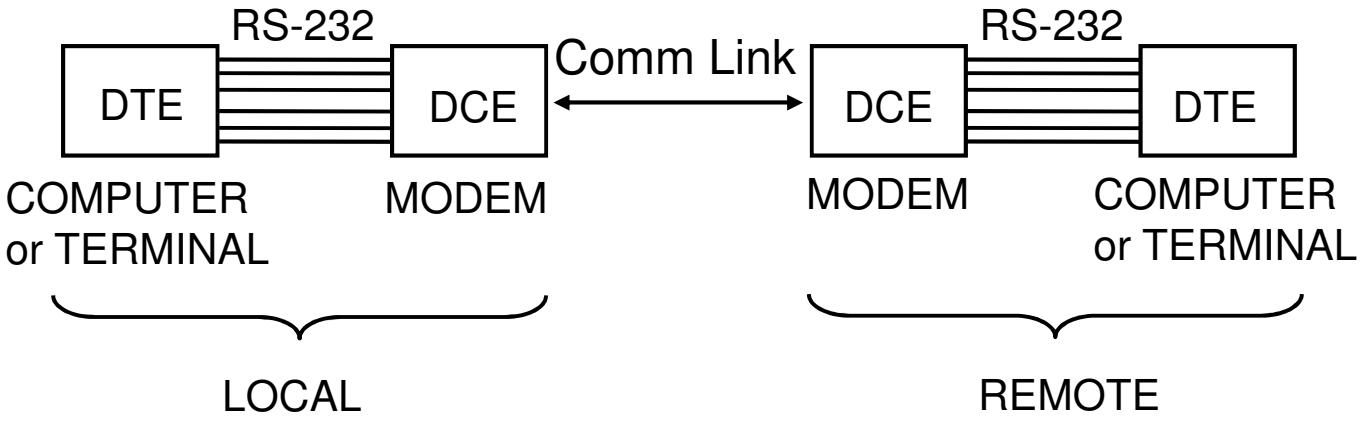
UART .- Receiver Unit

Serial-to-Parallel Conversion



(*) Also known as the BAUD RATE CLOCK

A data communication system



RS-232 Standard Establishes

- Electrical,
- Mechanical,
- Functional, and
- Procedural Specifications

for the communication interface between

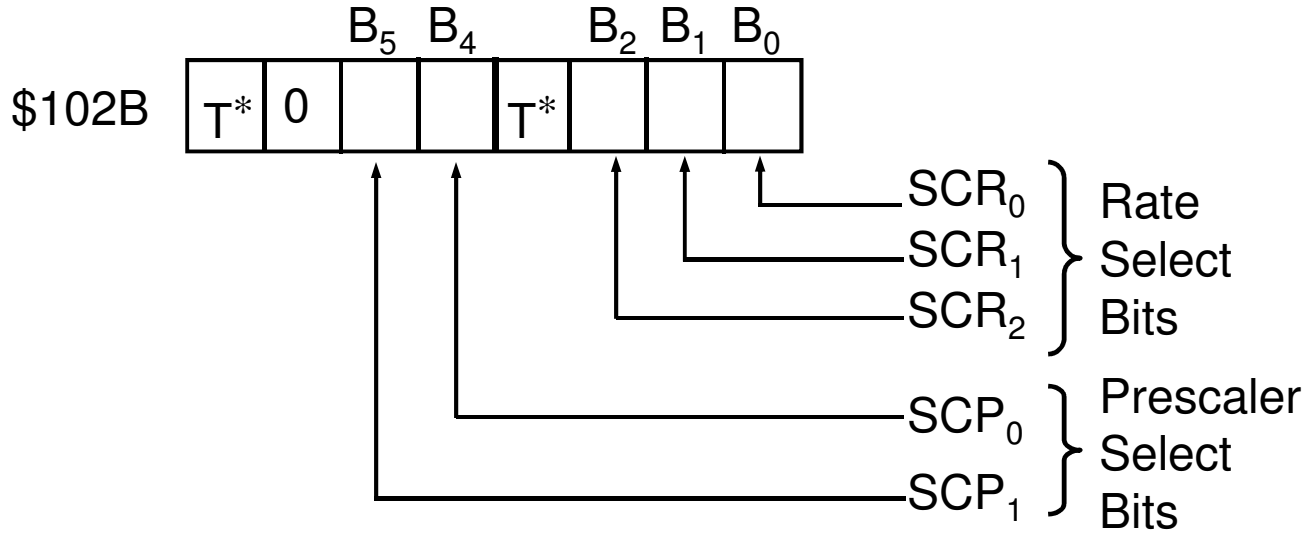
A Computer (or DTE [1]) and a Modem (or DCE [2]).

[1] Data Terminal Equipment, [2] Data Communication Equipment

SCI Unit Registers

Name	Address	Description
BAUD	$102B	Sets Line Speed (Baud Rate)
SCCR1	$102C	Control Register 1
SCCR2	$102D	Control Register 2
SCSR	$102E	Status Register
SCDR	$102F	Data Register (for both rx & tx)

BAUD Register



* T - Used only in test mode

For an E-clock frequency = 2MHz

SCP_1	SCP_0	Division Factor	Prescaler Output	Highest Baud Rate (Prescaler Output / 16)
0	0	1	2 MHz	125,000 Baud
0	1	3	2/3 MHz	41,667 Baud
1	0	4	0.5 MHz	31,250 Baud
1	**1**	13	2/13 MHz	**≈ 9,600 Baud**

BAUD Register (Continued)

For a Highest Baud Rate of 9,600 Baud ($SCP_1 = SCP_0 = 1$)

SCR_2	SCR_1	SCR_0	Division Factor	Selected Baud Rate
0	0	0	1	9,600 Baud
0	0	1	2	4,800 Baud
0	1	0	4	2,400 Baud
0	1	1	8	1,200 Baud
1	0	0	16	600 Baud
1	0	1	32	300 Baud
1	1	0	64	150 Baud
1	1	1	128	75 Baud

BAUD Register (Continued)

Example: Write a program segment in assembler to set the SCI unit baud rate equal to 2400 baud.

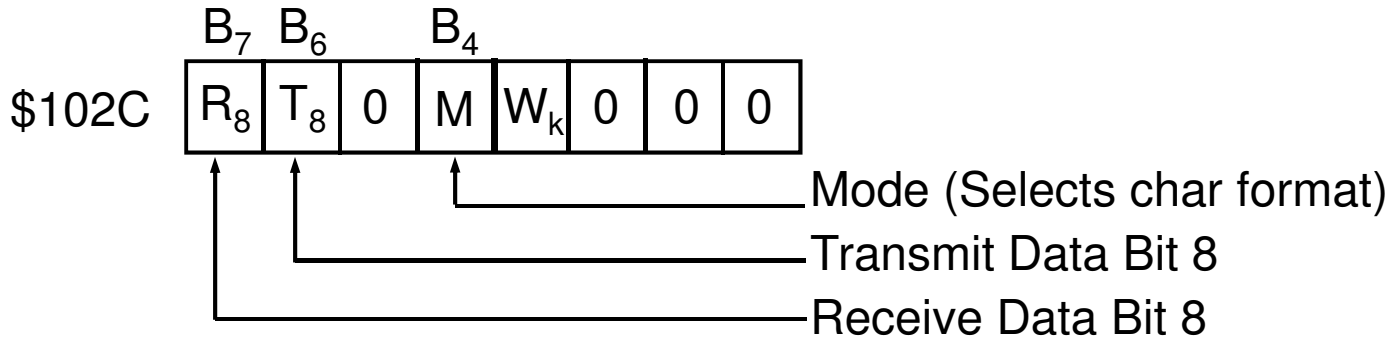
From previous tables we need

$B_5\ B_4 = 1\ 1$, and

$B_2\ B_1\ B_0 = 0\ 1\ 0$

```
BAUD        equ     $102B          ; Register address
BAUD2400    equ     %00110010      ; Control byte
            ldaa    #BAUD2400
            staa    BAUD           ; Setting line speed to 2400 baud
```

SCCR1 Register



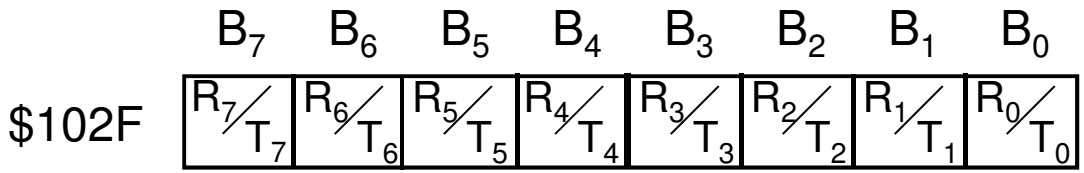
- M = 0, SCI rx & tx 8-bit data frames (Only SCDR is needed)
- M = 1, SCI rx & tx 9-bit data frames
  - In this case data bit B_8 is transferred through
    . T_8 during tx, and
    . R_8 during rx.

Frame Length

The SCI unit always uses

- 1 Start Bit, 8 or 9 Data Bits, and 1 Stop Bit = 10 or 11 bits/frame total

SCDR Register



(Receive and transmit double buffered)

Examples:

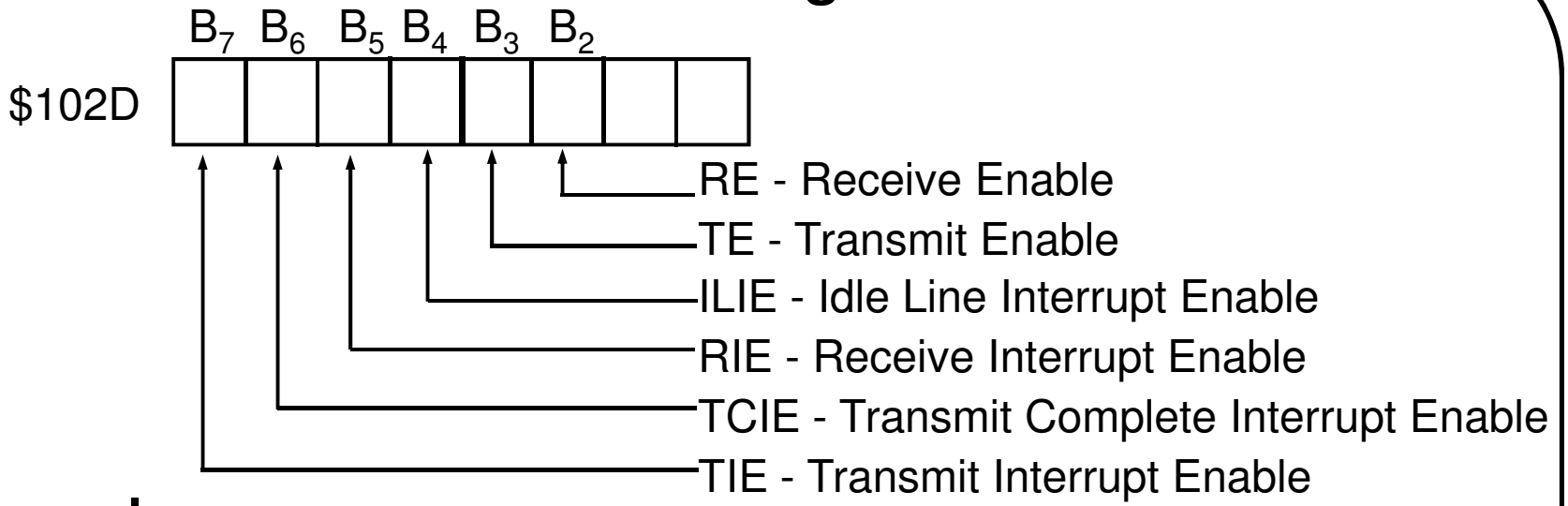
A) When the SCI unit has received new data and it is available for the CPU to read it (*i.e.*, RDRF condition)

```
ldaa    $102F    ; Brings the new data byte held by the SCDR
                 ; Rx buffer (RDR) into the CPU ACCA register
```

B) When the SCI unit is ready to accept a new byte from the CPU for transmission (*i.e.*, TDRE condition)

```
staa    $102F    ; Sends the data byte in ACCA to the SCI unit
                 ; SCDR Tx buffer (TDR) for transmission
```

SCCR2 Register

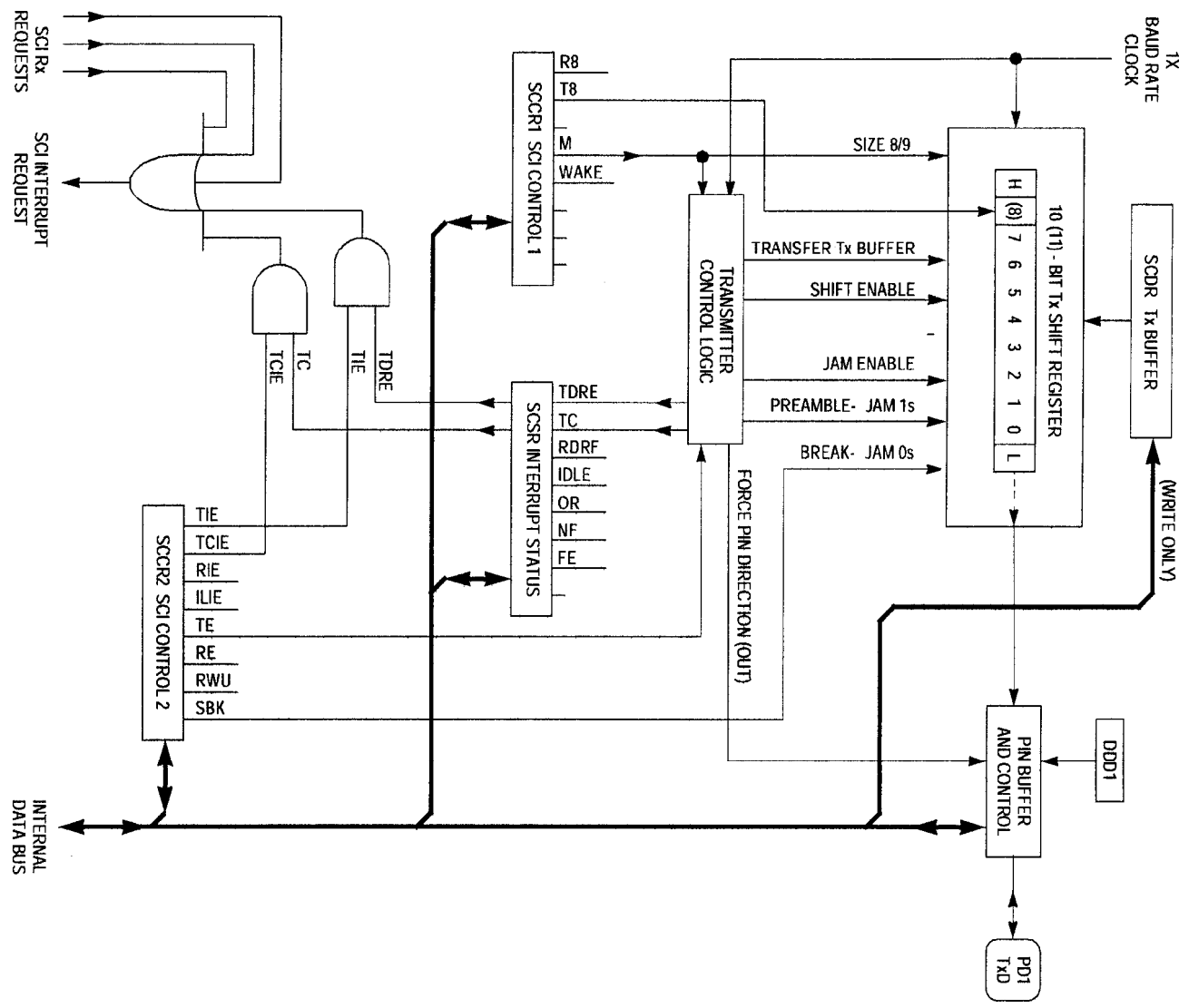


Example:

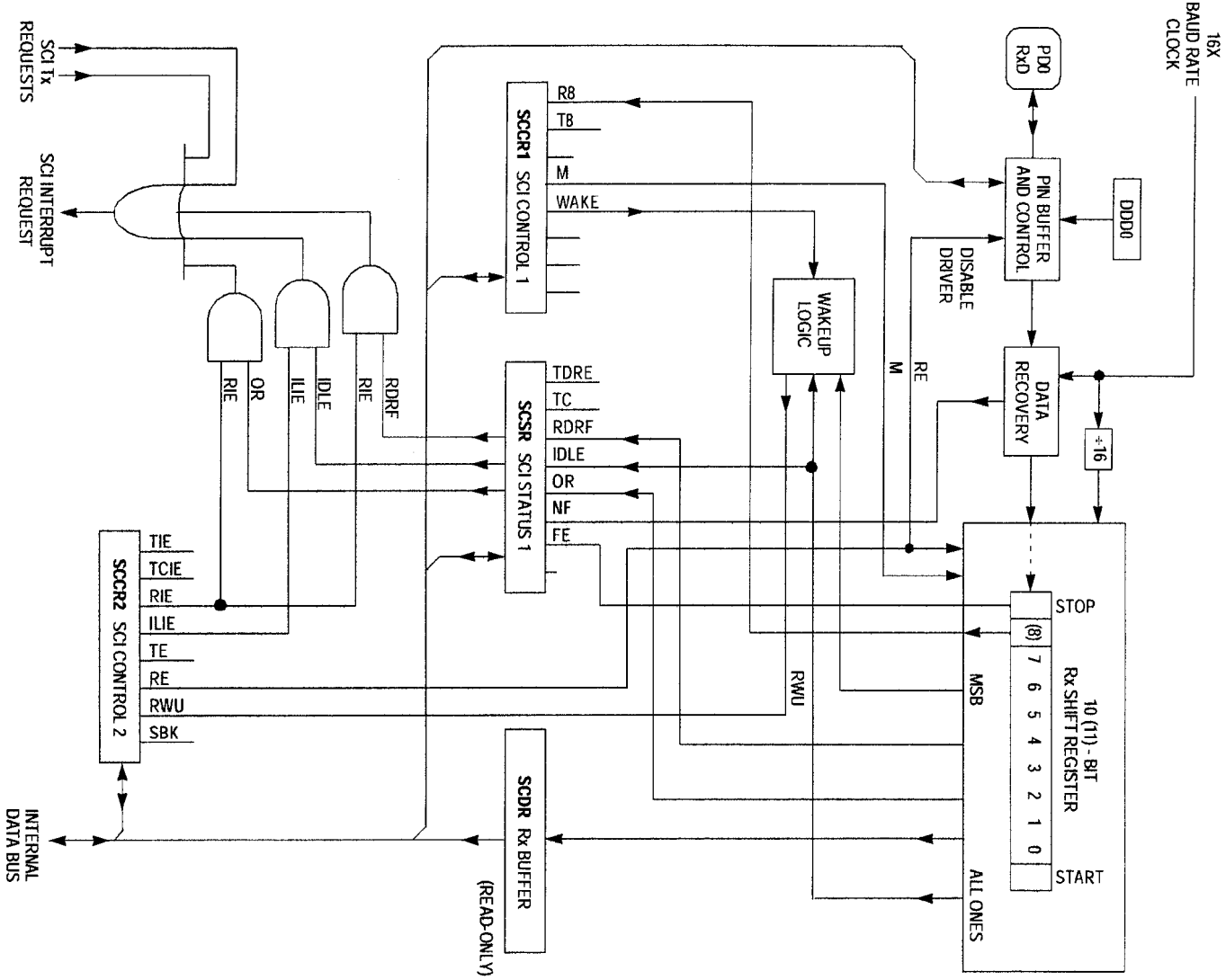
A) Enable the SCI unit for reception only (*i.e.*, need to **set** the RE bit)

```
; Address
SCCR2     equ    $102D

;Constant
RE        equ    $04

;Instructions
          ldaa   #RE       ; Load ACCA with control byte
          staa   SCCR2     ; Enables SCI for reception
```

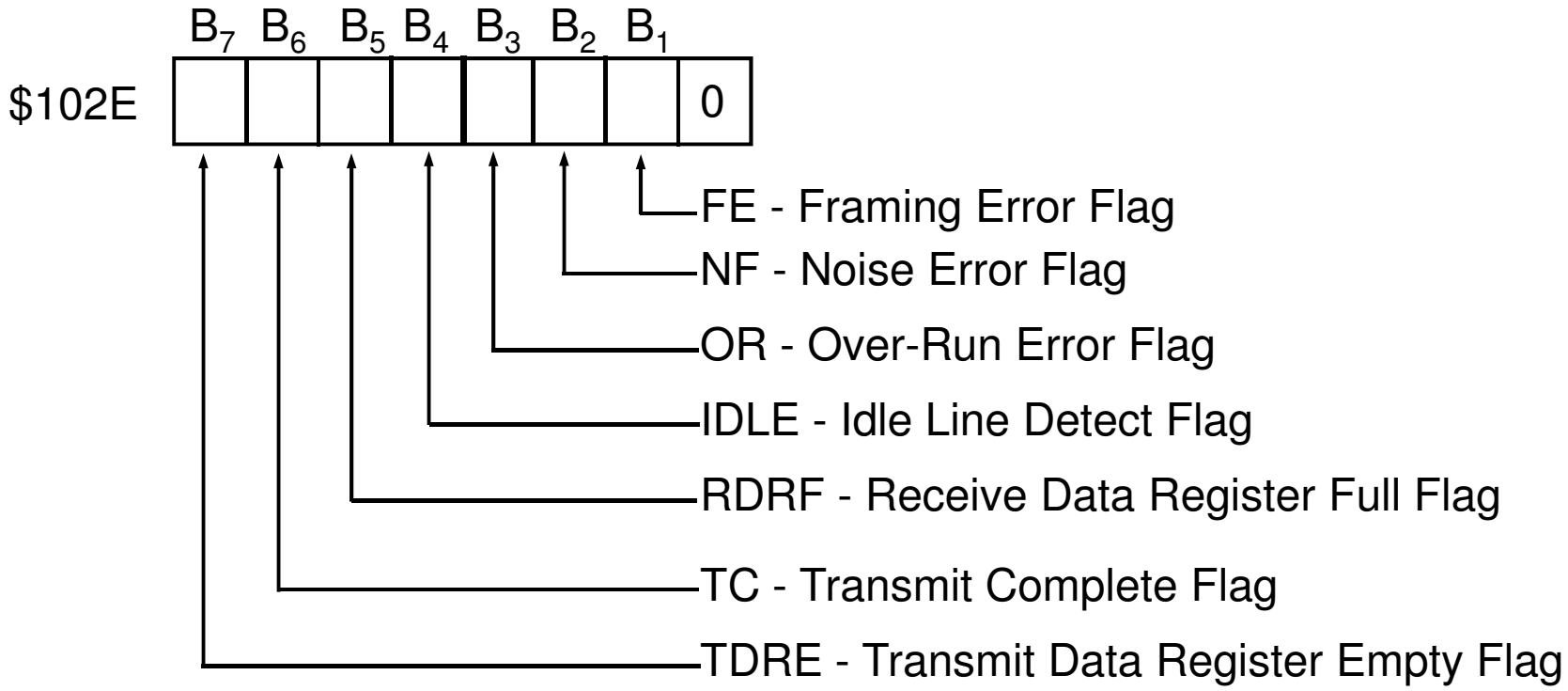
SCI Transmitter Block Diagram



SCI Receiver Block Diagram



SCSR Register



TDRE Flag

SET $\Rightarrow$ the SCI tx unit is ready to accept a new char from the CPU

CLEAR $\Rightarrow$ the TDR is still full,
the SCI unit needs time to transmit and avoid Over-Run

Example: Check whether the CPU can send a new char to the SCI without overwriting the last one sent

```
; Address Definitions
SCSR        equ     $102E       ; Status register address
; Constant Definitions
TDRE        equ     $80         ; Mask for TDRE flag in SCSR
; Instructions
            ldab    SCSR        ; ACCB ← SCSR
            andb    #TDRE       ; Is the TDRE flag SET?
            bne     SendChar    ; If YES goto send next char
```

RDRF Flag

SET $\Rightarrow$ the SCI rx unit has a new char ready for the CPU to read

CLEAR $\Rightarrow$ the RDR is empty, no char is available to be read from SCI

Example: Check whether a new char is ready at the SCI for the CPU to pick up

```
; Address Definitions
SCSR        equ     $102E       ; Status register address
; Constant Definitions
RDRF        equ     $20         ; Mask for RDRF flag in SCSR
; Instructions
            ldab    SCSR        ; ACCB ← SCSR
            andb    #RDRF       ; Is the RDRF flag SET?
            bne     ReadChar    ; If YES goto read next char
```

Polling Method .- Transmission

1.- Enable the SCI tx: Making TE = 1

Example:
```
ldaa  SCCR2
oraa  #$08
staa  SCCR2
```

2.- Read SCSR (hardware requisite)

```
ldab  SCSR
```

3.- Write data to be tx to the SCDR

Example:
```
ldaa  Data
staa  SCDR
```

Sending a stream of chars

4.- Check SCSR until the TDRE flag is Set

5.- When TDRE = ‘1’, next char is written to SCDR

6.- Back to Step 3, cycle repeats until last char is sent to SCI for tx.

Polling Method .- Reception

1.- Enable the SCI rx: Making RE = 1

Example:

```
ldaa  SCCR2
oraa  #$04
staa  SCCR2
```

2.- Check SCSR periodically until the RDRF flag is Set

3.- When RDRF = '1', next char is read from SCDR, *e.g.* `ldaa SCDR`

4.- Some processing is done if required

5.- Back to Step 2, cycle repeats forever or until reception is finished.

Dual D-type flip-flop with set and reset; positive-edge trigger

74HC/HCT74

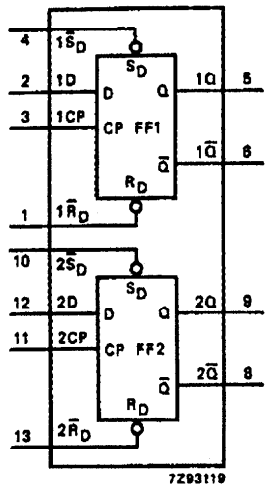


Fig.4 Functional diagram.

FUNCTION TABLE

INPUTS				OUTPUTS	
$\overline{S}_D$	$\overline{R}_D$	CP	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H	H

INPUTS				OUTPUTS	
$\overline{S}_D$	$\overline{R}_D$	CP	D	Q_{n+1}	$\overline{Q}_{n+1}$
H	H	↑	L	L	H
H	H	↑	H	H	L

Note

1. H = HIGH voltage level
   L = LOW voltage level
   X = don't care
   ↑ = LOW-to-HIGH CP transition
   Q_{n+1} = state after the next LOW-to-HIGH CP transition

Directing SCI RxD pin PD_0 to the MCU I/O port connector

```
; Address Definitions
FFLOP       equ     $5000           ; A decode address for the FlipFlop
; Constant Definitions
Byte2Write  equ     $00             ; Bit 0 must be clear, B0 = 0
; Instructions
            ldaa    #Byte2Write     ; ACCA ← 0, clra is an alternative
            staa     FFLOP          ; Switches PD0 to Target System
```

Directing SCI RxD pin PD_0 to the RS-232 compatible Host Computer I/O port

```
; Constant Definitions
Byte2Write  equ     $0F             ; Bit 0 must be set, B0 = 1
; Instructions
            ldaa    #Byte2Write     ; ACCA ← $0F
            staa    FFLOP           ; Switches PD0 to Host Comp I/O port
                                    ; to communicate over an RS-232
                                    ; protocol (with the PC in our case)
```

MC6850 (ACIA) Register Selection

Register Select Input (RS)	R/$\overline{W}$	Register Selected
1	0	Tx Data Register (TDR)
1	1	Rx Data Register (RDR)
0	0	Control Register (CR)
0	1	Status Register (SR)

Bibliography for Interrupts & the SCI unit

Textbook

- Section 2.1.3
- Section 2.2.2
- Section 2.3.1
- Section 3.9
- Sections 4.2.2, 4.2.3 & 4.3

HC11 Reference Manual *(*Pink Book*)*

- Section 5.5 Interrupt Process
- Section 9.5.2 Interrupts & Status Flags (SCI tx)
- Section 9.6.4 Receive Status Flags & Interrupts

Interrupt Acknowledgement Procedure

1. Main program execution is suspended
2. Program state is saved to the stack
3. PC $\leftarrow$ Interrupt Vector of highest priority interrupt pending
   - Execution continues at this address
4. ISR is concluded with an **RTI** instruction
   - Program state, *ie.* all CPU registers are restored
   - PC $\leftarrow$ Return Address,

     where execution of main program resumes

Example of SCI Interrupt Service Routine (ISR)

```
; Address Definitions
SCSR        equ   $102E        ; Status register address
SCDR        equ   $102F        ; Data register address
; Constant Definitions
RDRF        equ   $20          ; Mask for RDRF flag in SCSR
; Instructions
SCI_isr     ldaa  SCSR         ; ACCA ← SCSR (SCI status)
            bita  #RDRF        ; Is there a new available char?
            bne   SCI_rx_isr   ; If YES goto service rx
              :                }  Check / service causes other than
                                  RDRF, or just branch to rti at the end
SCI_rx_isr  ldaa  SCDR         ; Read available char from SCI
              :                }  Service reception
            rti                ; Return from ISR
```

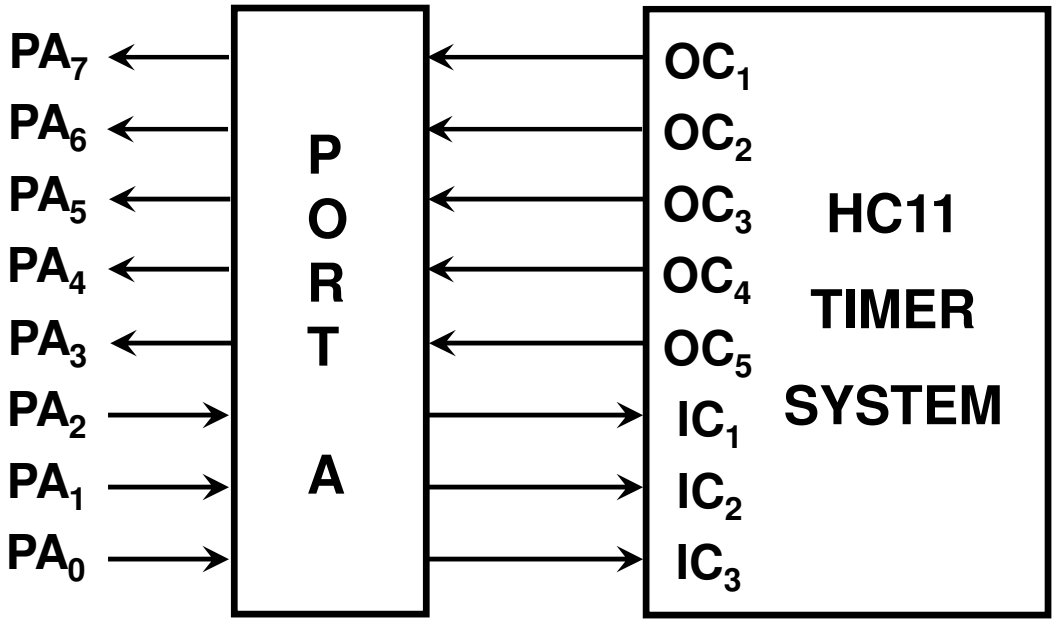
Interrupt Vector Jump Table

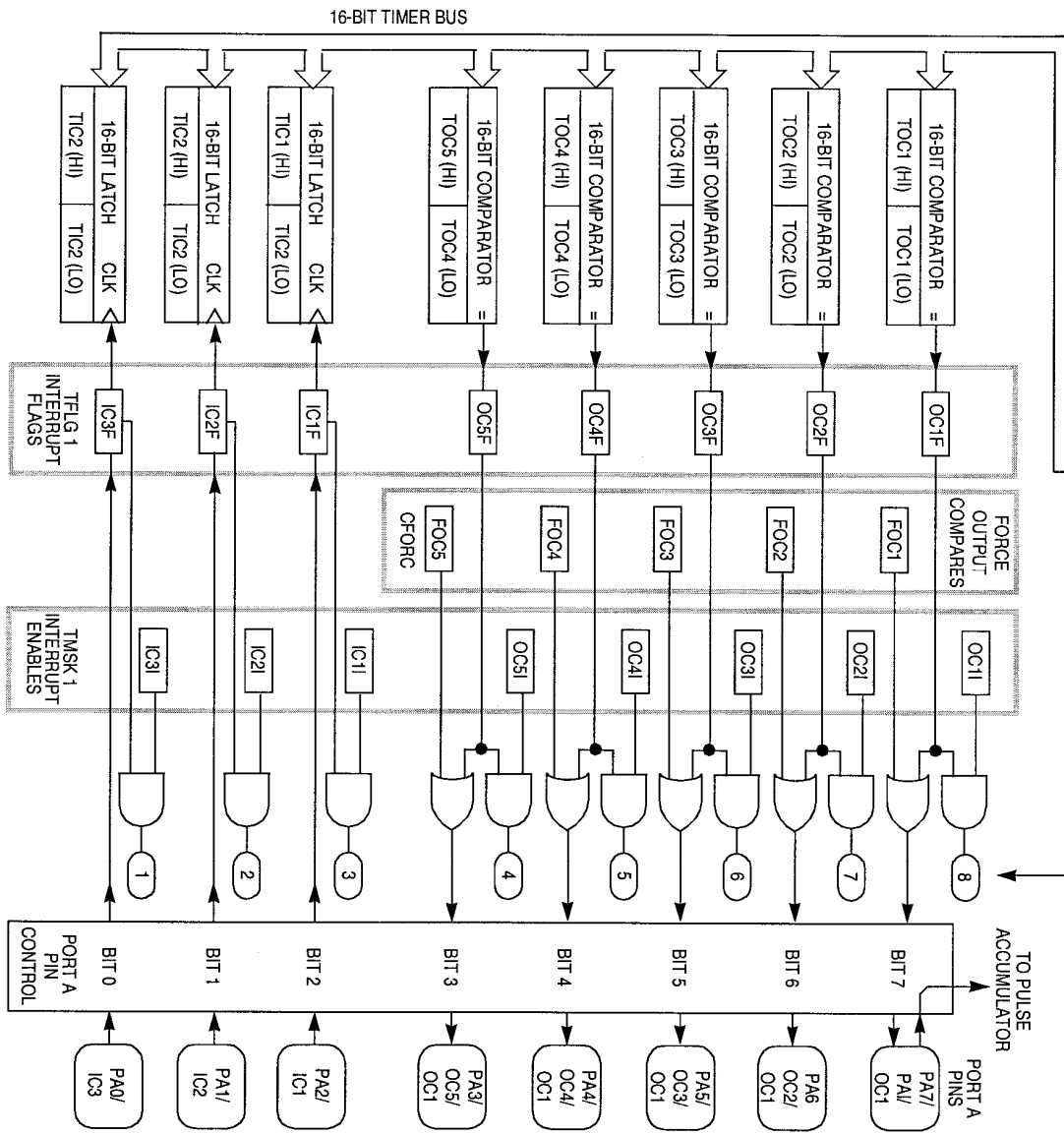
INTERRUPT VECTOR	FIELD
Serial Communications Interface (SCI)	$00C4-$00C6
Serial Peripheral Interface (SPI)	$00C7-$00C9
Pulse Accumulator Input Edge	$00CA-$00CC
Pulse Accumulator Overflow	$00CD-$00CF
Timer Overflow	$00D0-$00D2
Timer Output Compare 5	$00D3-$00D5
Timer Output Compare 4	$00D6-$00D8
Timer Output Compare 3	$00D9-$00DB
Timer Output Compare 2	$00DC-$00DE
Timer Output Compare 1	$00DF-$00E1
Timer Input Capture 3	$00E2-$00E4
Timer Input Capture 2	$00E5-$00E7
Timer Input Capture 1	$00E8-$00EA
Real Time Interrupt	$00EB-$00ED
IRQ	$00EE-$00F0
XIRQ	$00F1-$00F3
Software Interrupt (SWI)	$00F4-$00F6
Illegal Opcode	$00F7-$00F9
Computer Operating Properly (COP)	$00FA-$00FC
Clock Monitor	$00FD-$00FF

Example of EVB Jump Table Update for SCI

```
; Address Definitions
SCI_JMPTBL1   equ   $00C4    ; JMP OpCode Address in Table for SCI
SCI_JMPTBL2   equ   $00C5    ; Start of Jump Address in Table for SCI
; Constant Definitions
JMPOpCode     equ   $7E      ; OpCode for JMP instruction
; Instructions
              ldaa  #JMPOpCode     ; ACCA ← JMP OpCode
              staa  SCI_JMPTBL1    ; Storing OpCode to table
              ldx   #$C400         ; IX ← SCI ISR address
              stx   SCI_JMPTBL2    ; Storing Jump Address to table
               :                   } Other initialization instructions

              aorg  $C400    ; Code to follow loaded at $C400
SCI_isr       <1st Instruction of SCI ISR>
```

Timer System Functions & Port A Pins



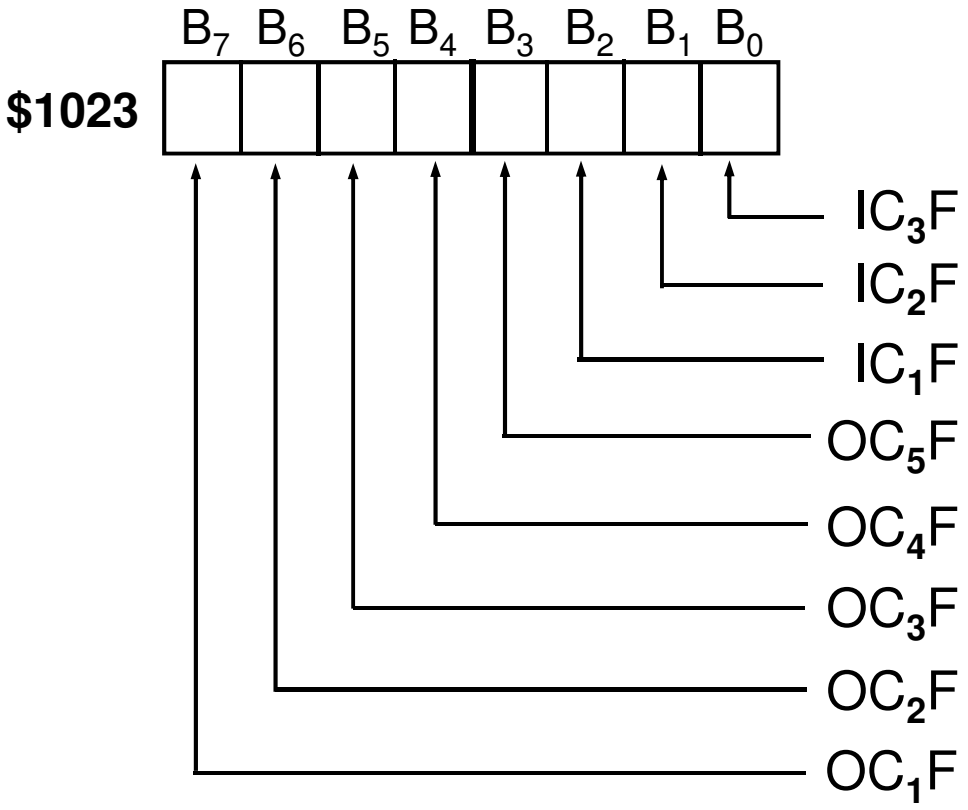


Main Timer System Block Diagram

Selecting the Free-running Counter Frequency (for an E-clock freq = 2 MHz)

PR_1	PR_0	Prescale Factor	Counter Updates Every	Counter Clock Freq
0	0	1	500 ns	2 MHz
0	1	4	2 μs	0.5 MHz
1	0	8	4 μs	250 KHz
1	1	16	8 μs	125 KHz

Flag Register TFLG1



Example:

Clearing OC_3F

```
OC3F    equ    %00100000
TFLG1   equ    $1023

        ldaa   #OC3F
        staa   TFLG1
```

Example .- Updating TOC_2 for a 1.5ms interval (Counter clock = 2MHz)

```
TCNT        equ   $100E      ; Address of TCNT register
TOC2        equ   $1018      ; Address of TOC2 register
Increment   equ   !3000      ; Increment in decimal

Initialization (1st time):
            ldd   TCNT
            addd  #Increment
            std   TOC2
            :

Part of ISR or Successful Polling:
            ldd   TOC2
            addd  #Increment
            std   TOC2       ; Hardware detail:
                             ; When TOC2HI is written
                             ; compares are suspended
                             ; for 1 E-clock cycle
```

Controlling Port A Pin State

Timer Control Register 1 (TCTL1)

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
$1020	**OM_2**	**OL_2**	**OM_3**	**OL_3**	**OM_4**	**OL_4**	**OM_5**	**OL_5**
RESET =	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

OM_X	OL_X	Action Taken
0	0	None (Disconnected)
0	1	Toggle OC_X line
1	0	Clear OC_X line (logic '0')
1	1	Set OC_X line (logic '1')

Example:

Setting PA_5 to toggle on each successful compare with TOC3

```
CTLBYTE  equ  %00010000
TCTL1    equ  $1020

         ldaa  #CTLBYTE
         staa  TCTL1
```

Local IC/OC Timer Interrupt Masks

Interrupt Enable Register 1 (TMSK1)

	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
$1022	**OC_1I**	**OC_2I**	**OC_3I**	**OC_4I**	**OC_5I**	**IC_1I**	**IC_2I**	**IC_3I**
RESET =	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

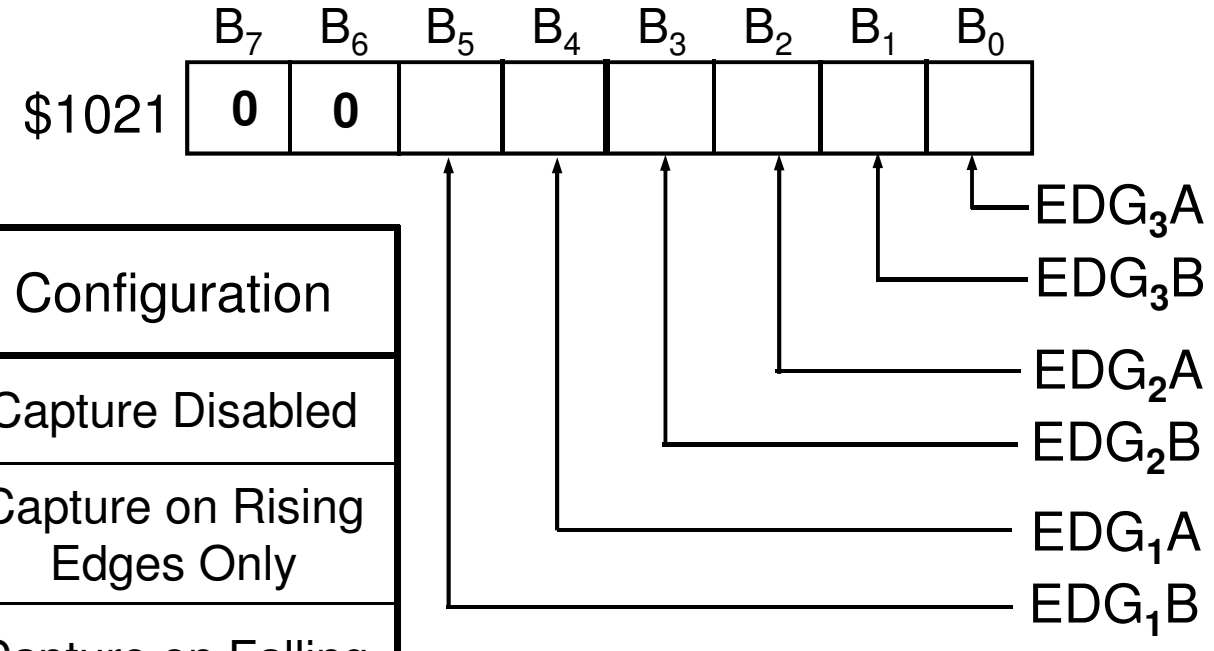
When '0' = Interrupt Inhibited
'1' = Interrupt Enabled

Example:
Enabling Interrupts from TOC4

```
OC4I    equ  %00010000
TMSK1   equ  $1022

        ldaa  TMSK1
        oraa  #OC4I
        staa  TMSK1
```

Selecting Port A Triggering Event

Timer Control Register 2 (TCTL2)



EDG_XB	EDG_XA	Configuration
0	0	Capture Disabled
0	1	Capture on Rising Edges Only
1	0	Capture on Falling Edges Only
1	1	Capture on Any Edge (Rising or Falling)

Example .- Period Measurement using TIC3 (PA_0)

```
TIC3            equ   $1014
TCTL2           equ   $1021
CTLBYTE         equ   %00000001        ; Rising edge

Initialization  ldaa  # CTLBYTE
                staa  TCTL2
                :

Part of ISR     ldd   TIC3
or              subd  PreviousReading  ; Period = TIC_new - TIC_old
Successful      std   Period
Polling         ldd   TIC3
                std   PreviousReading  ; Update TIC_old
                <clear IC3F>
                :
PreviousReading ds    2                ; TIC_old
Period          ds    2                ; Current period
```

Example .- Pulse Width Measurement using TIC3 (PA_0)

```
CTLBYTE1    equ   %00000001;      Rising edge
CTLBYTE2    equ   %00000010;      Falling edge
```

Initialization

```
            ldaa  # CTLBYTE1
            staa  TCTL2
            < EdgeFlag ← Rising >
            :
```

Part of ISR or Successful Polling

```
- If EdgeFlag == Rising ?
            ldd   TIC3
            std   FirstEdge
            < EdgeFlag ← Falling >
            ldaa  #CTLBYTE2          ; Enable falling
            staa  TCTL2              ; edge detection
            :
```

Example .- Pulse Width Measurement (cont'd)

Part of ISR or Successful Polling

```
        :
- If EdgeFlag == Falling ?
    ldd     TIC3
    subd    FirstEdge
    std     PWidth
    < EdgeFlag ← Rising >
    ldaa    #CTLBYTE1          ; Enable rising
    staa    TCTL2              ; edge detection
        :
FirstEdge   ds   2
PWidth      ds   2
EdgeFlag    ds   1
```

Using IC Timer Interrupts:

1) Set the EVB jump vector for the corresponding IC function

2) Enable TIC Function in TCTL2

3) Enable local interrupt mask IC_xI

4) Enable global interrupt mask (I-flag in CCR)

Example:

Enabling interrupts from TIC3 (Step 3 above)

```
TMSK1       equ     $1022           ; Address of TMSK1 register
IC3I        equ     %00000001       ; IC3I Mask
   :
   :
   :
```

Part of Initialization

```
            ldaa    TMSK1
            oraa    #IC3I
            staa    TMSK1
```

Example .- Generating a 1 sec Delay with TOC_x

1. Delay interval is triggered (1st time):

```
        ldaa    #!30        } Counter
        staa    Counter     } Initialization
        ldd     TOCx        }
        addd    #!33920     } TOCx Initialization [OCx successful compare is triggering the interval]
        std     TOCx        }
        bra     Return
```

2. With every SUCCESSFUL COMPARE (after 1st time above):

```
        dec     Counter         ; Decrement Counter
        bmi     ExecTask        ; Check if Counter < 0
        bra     Return          ; If NOT keep decrementing it
ExecTask jsr    ProcessX        ; If YES take action
        <Deactivate TOCx interrupt>  ; for a One-time Task
Return  <clear OCxF>
        rti
```

Example .- Generating a 2 sec Square Wave in PA_6 (TOC2)

```
TOC2_isr    ldaa    Counter
            bne     CountDown
            ldaa    #!30          } Counter
            staa    Counter       } Initialization
            ldd     TOC2          }
            addd    #!33920       } Update
            std     TOC2          } TOC2
            ldaa    TCTL1         }
            anda    #%00111111    } Disconnect TOC2
            staa    TCTL1         } from output pin
                                  } logic (PA6 )
            bra     Return
CountDown   dec     Counter
            bne     Return
            ldaa    TCTL1         }
            oraa    %01000000     } When Counter reaches 0,
            staa    TCTL1         } set PA6 line to toggle its
                                  } value with next TOC2
                                  } successful compare
Return      < clear OC2F >
            rti
```

If Counter is 0, Initialize TOC2 & Counter

Example .- Generating a 2 sec Square Wave (cont'd)

TOC2_init (1) < Counter ← 0 >

(2) Set PA_6 logic to toggle its value with the next TOC2 successful compare

(3) Set up TOC2 interrupt vector (e.g. EVB jump table)

(4) Clear OC2F

(5) Enable TOC2 interrupts
i.e. OC2I ← 1

Measuring Long Periods

Detecting & Solving a Missed Overflow

```
TIC_isr   « Find & Save: TIC_Δ = TIC_new – TIC_old »
          « If TIC_Δ < 0, Counter ← Counter – 1 »
          « If TIC_new < 5000_d
              « read TOF »
              « If TOF is SET,
                   inc  Counter
                   IncFlag ← 0 » »
          « Save Counter »   ; Preparing for measuring
          clr  Counter       ; next period
          « clear IC_xF flag »  ; Acknowledge interrupt service
          rti

OF_isr    « If IncFlag is SET,
               inc  Counter »
          « IncFlag ← 1 »
          « clear TOF flag »  ; Acknowledge interrupt service
          rti
```

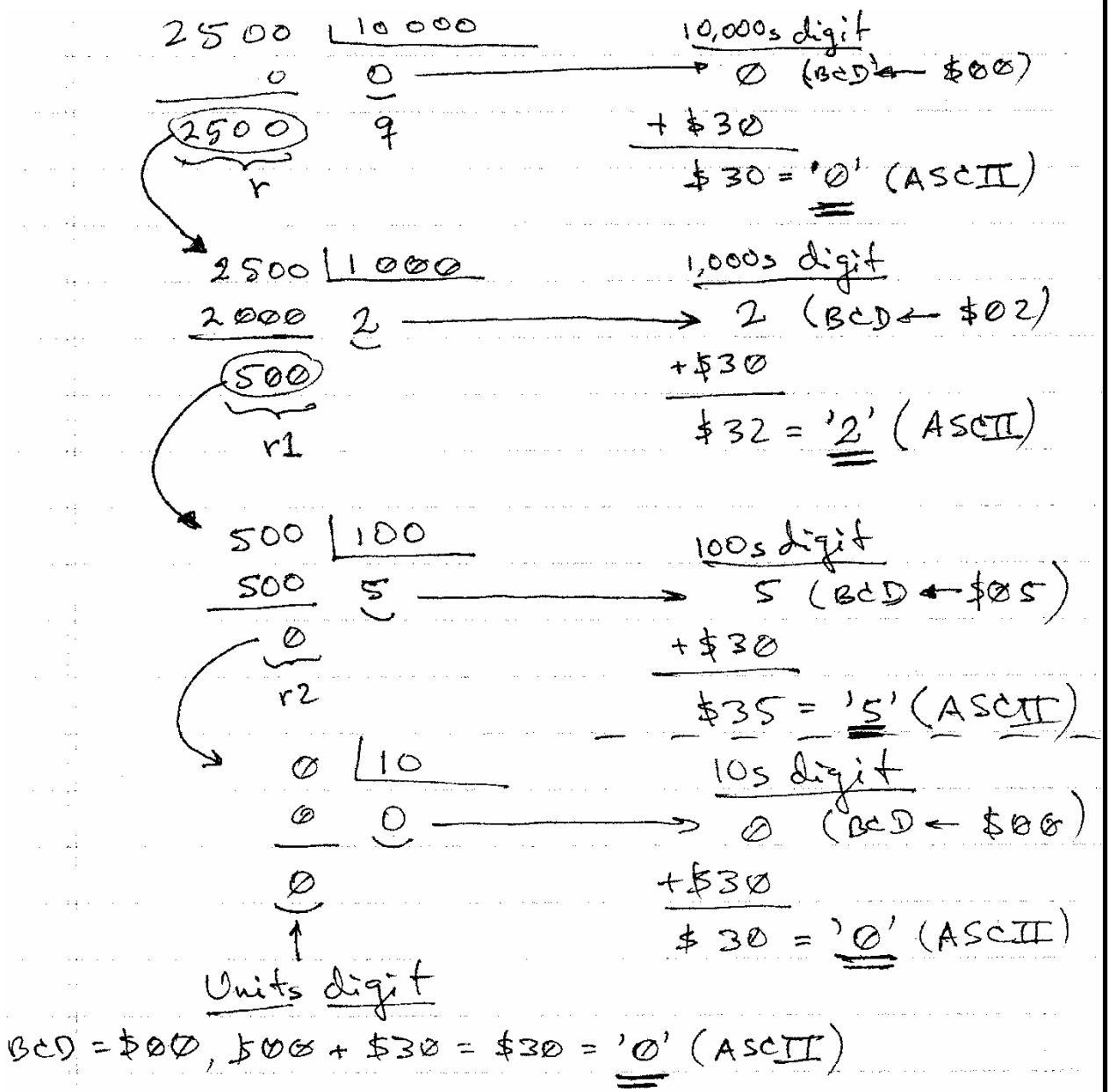
- Assuming 5000 E-clk cycles ($PR_0 = PR_1 = 0$) are sufficient to have OF serviced before IC.
- Avoids wrong correction when TIC_{new} < $FFFF but close to it.

Solving a Missed Overflow (Alternative – No IncFlag)

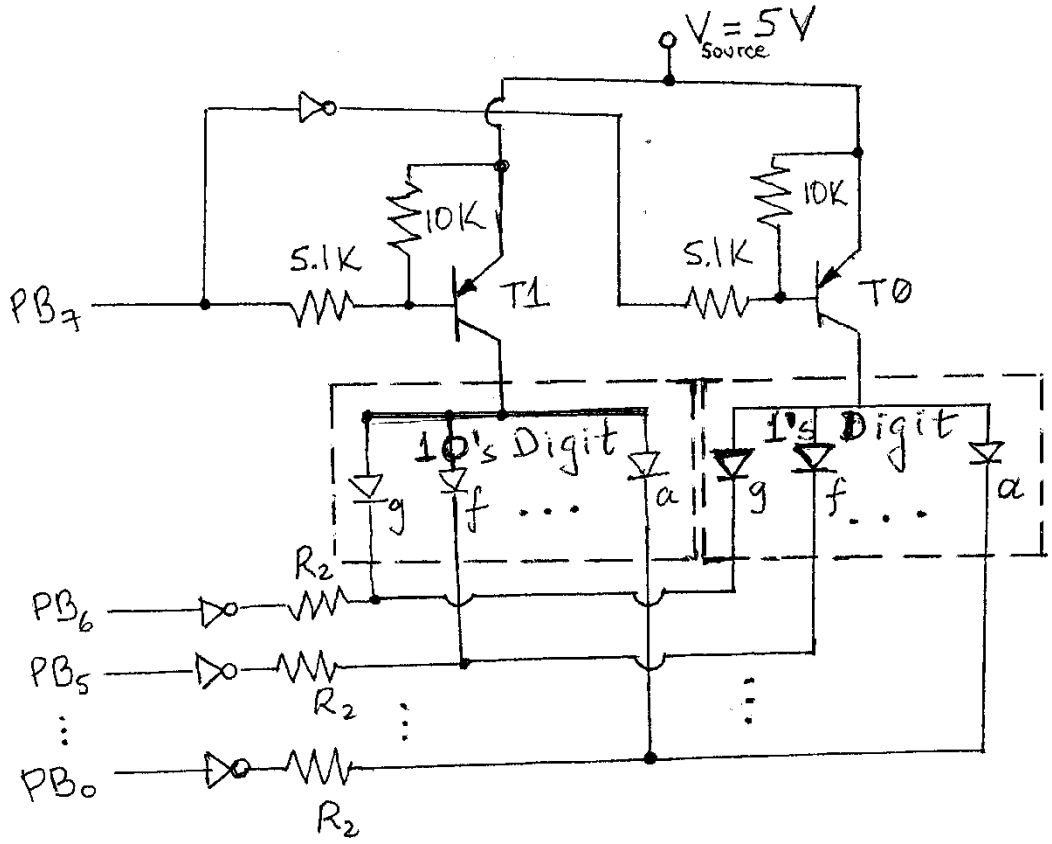
```
TIC_isr   « Find & Save: TIC_Δ = TIC_new – TIC_old »
          « If TIC_Δ < 0, Counter ← Counter – 1 »
          « If TIC_new < 5000_d                          } Detecting & Solving
              « read TOF »                               } a Missed Overflow
              « If TOF is SET,                           }
                    inc   Counter                        }
                    Save Counter       ; Preparing for measuring
                    Counter ← $FF      ; next period if OF missed
                else
                    Save Counter       ; Preparing for measuring
                    Counter ← $00 »    ; next period if OF serviced
            else Save Counter, Counter ← $00 »
          « clear IC_xF flag »         ; Acknowledge interrupt service
          rti

OF_isr    incCounter
          « clear TOF flag »           ; Acknowledge interrupt service
          rti
```

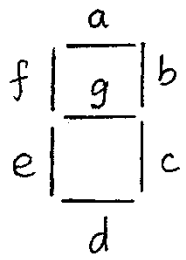
Hex to ACD (ASCII-coded decimal)

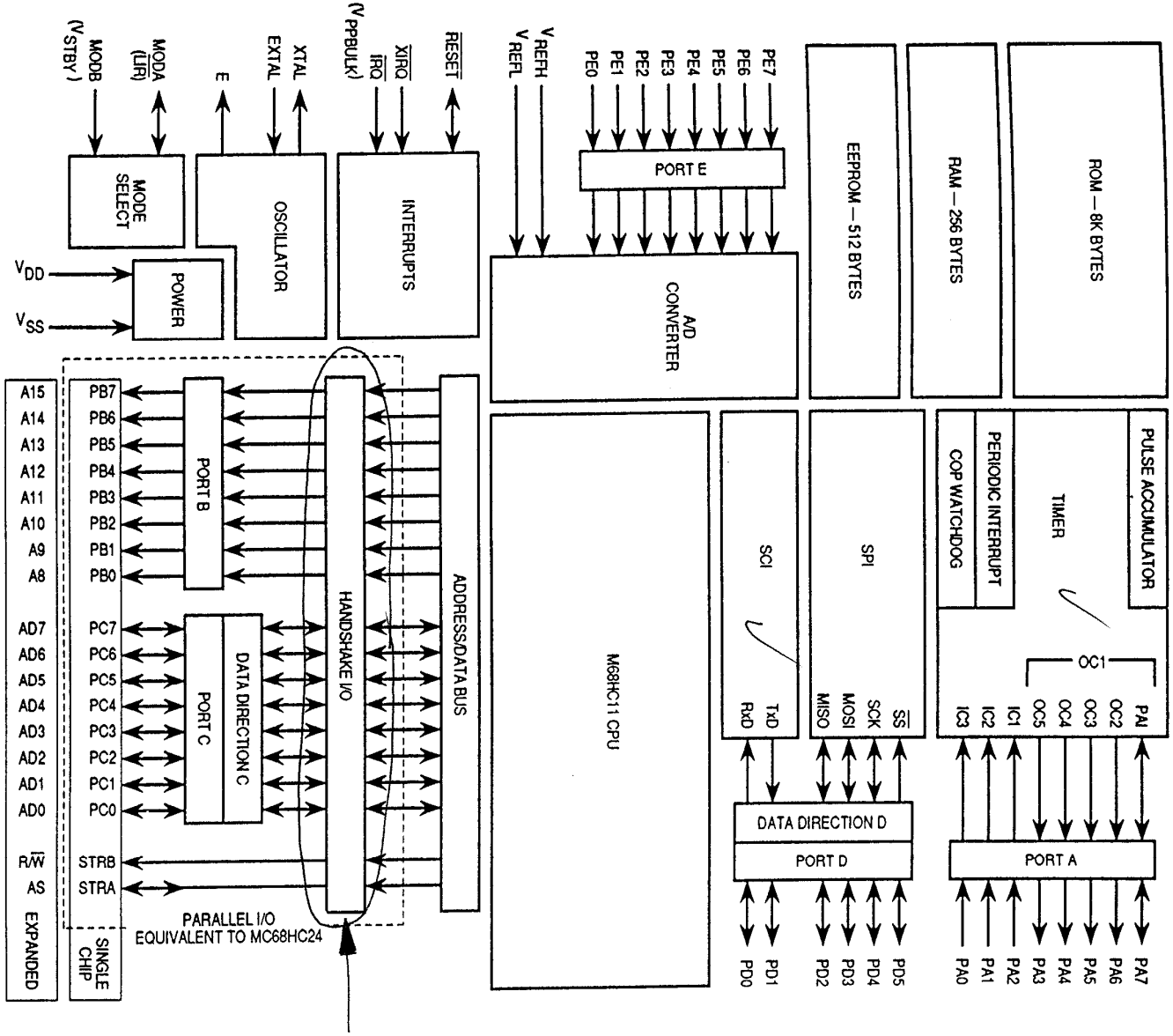
• HtoD Subroutine - Example



Parallel PORTB Control of a Two Digit Common-Anode 7-Segment LED Display





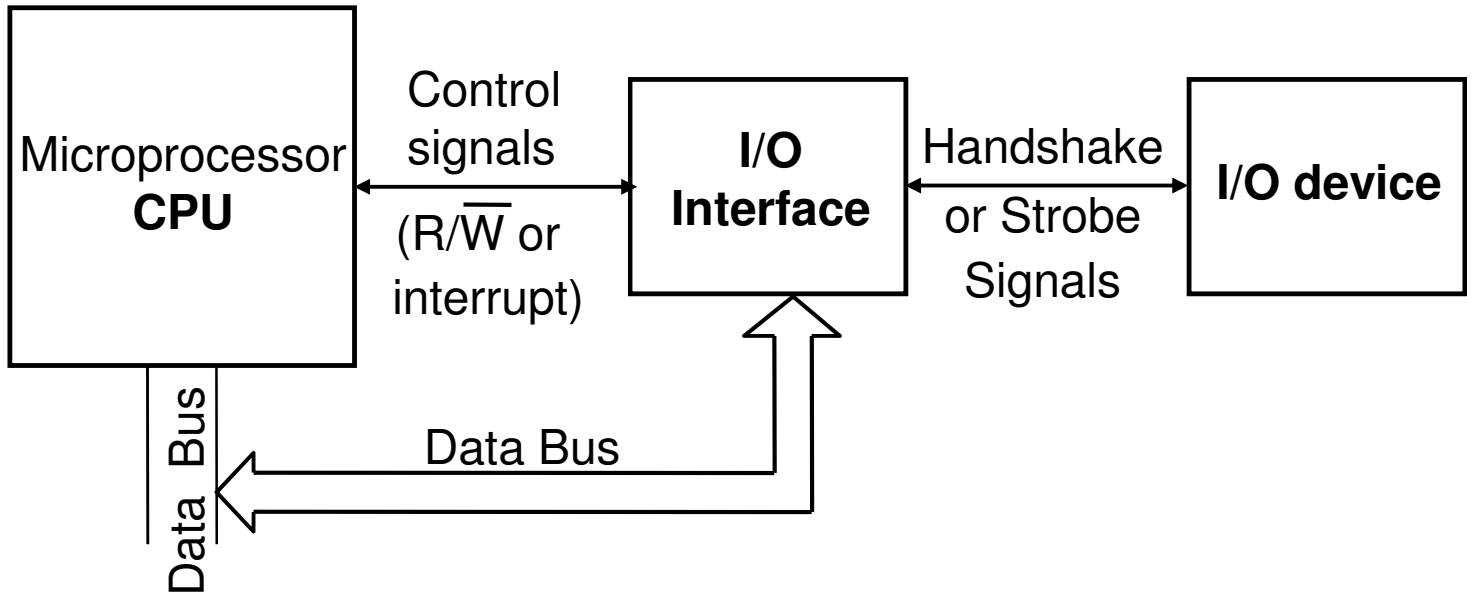


MC68HC11 Block Diagram

I/O Transfer Synchronization

The role of an I/O interface unit

1. Synchronizing data transfer between CPU and I/O interface unit.
2. Synchronizing data transfer between I/O interface and I/O device.



Synchronizing the Microprocessor CPU and the I/O Interface Unit

The polling method

1. for **input** -- the CPU checks a status bit of the interface unit to find out if the interface unit has received new data from the input device.
2. for **output** -- the CPU checks a status bit of the interface unit to find out if it can send new data to the interface unit.

The interrupt-driven method

1. for **input** -- the interface unit interrupts the CPU whenever it has received new data from the input device.
2. for **output** -- the interface unit interrupts the CPU whenever it can accept new data from the CPU.

Synchronizing the I/O Interface Unit and I/O Devices

Brute-force method -- useful when the data timing is unimportant

1. for **input** -- nothing special is done. The CPU reads the interface unit and the interface unit returns the voltage levels on the input port pins to the CPU.
2. for **output** -- nothing special is done. The interface unit places the data that it received from the CPU directly on the output port pins.

The strobe method -- a strobe signal is used to indicate that data are stable on I/O port pins

1. for **input** -- the interface unit latches the data into its data register using the strobe signal.
2. for **output** -- a) the interface unit places the data received from the CPU on the output port pins and asserts the strobe signal.
   b) the output device latches the data using this strobe signal.

Synchronizing the Interface Unit and I/O Devices (cont'd)

The handshake method -- used when timing is crucial

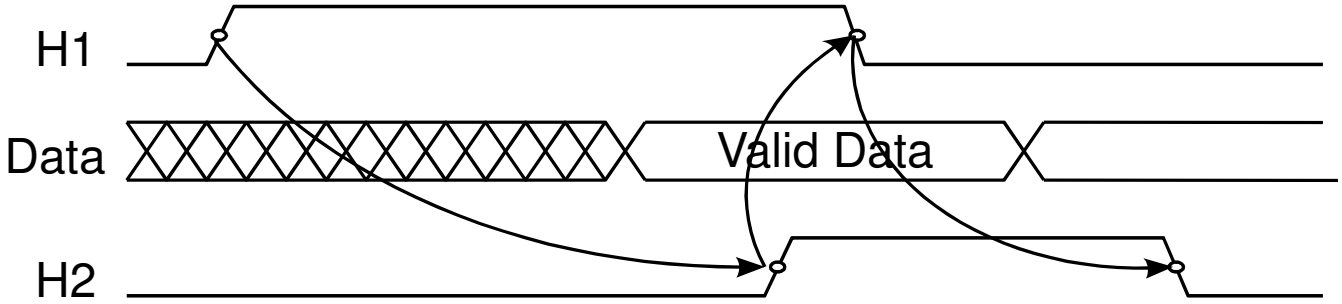
- For **input** and **output,**
  - Two handshake signals are used to synchronize the data transfer:
    1. One signal, call it H1, is asserted by the interface unit.
    2. The other signal, call it H2, is asserted by the I/O device.
- Two handshake modes are available -- ***pulse*** mode and ***interlocked*** mode.

Input Handshake Protocol

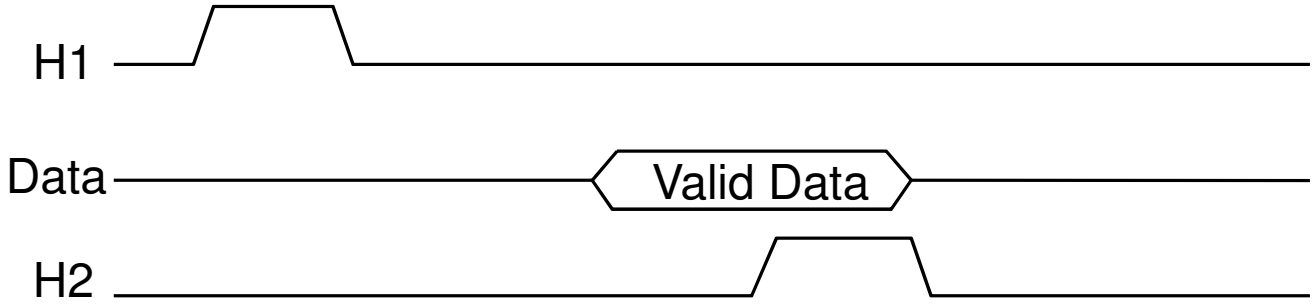
Step 1. The interface unit asserts (or pulses) H1 to indicate its intention to input data.

Step 2. The input device puts data on the data port pins and also asserts (or pulses) the handshake signal H2.

Step 3. The interface unit latches the data and de-asserts H1.
After some delay, the input device also de-asserts H2.



(a) Interlocked



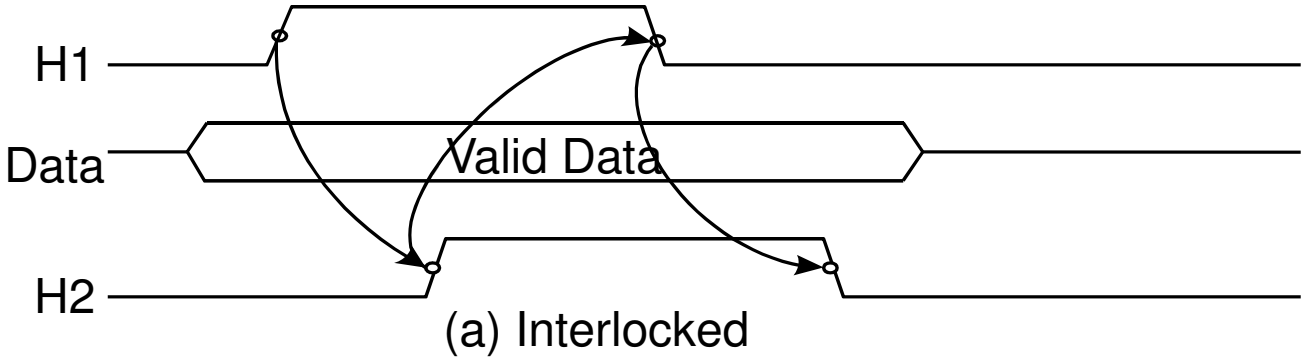
(b) Pulse mode

Output Handshake Protocol

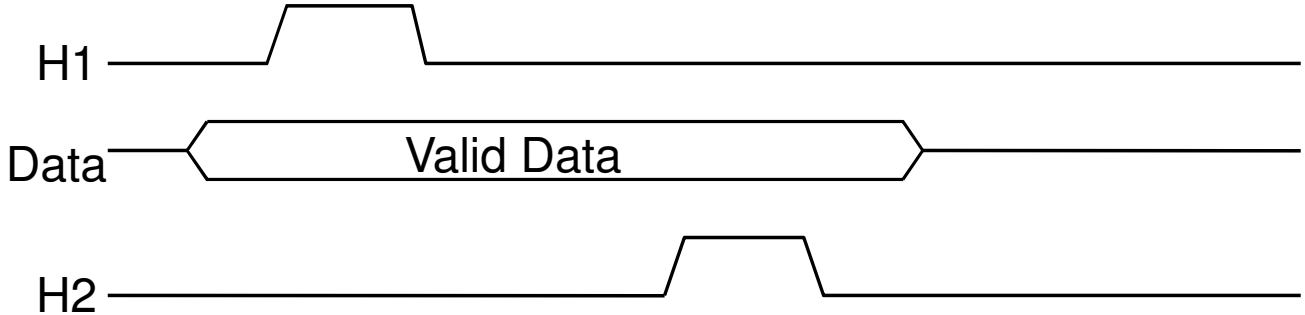
Step 1. The interface unit places data on the port pins and asserts (or pulses) H1 to indicate that it has valid data to be output.

Step 2. The output device latches the data and asserts (or pulses) H2 to acknowledge the receipt of data.

Step 3. The interface unit de-asserts H1 following the assertion of H2. The output device then de-asserts H2.



(a) Interlocked



(b) Pulse Mode

Parallel I/O Control Register (PIOC)

- All strobed mode I/O and handshake I/O are controlled by this register

	7	6	5	4	3	2	1	0	
	STAF	STAI	CWOM	HNDS	OIN	PLS	EGA	INVB	$1002
Upon RESET	0	0	0	0	0	U	1	1	

STAF: *Strobe A flag*
This bit is set when a selected edge occurs on the STRA signal.

STAI: *Strobe A interrupt enable*
When STAF = STAI = '1', an interrupt is requested to the CPU.

CWOM: *Port C wired-or mode*
0: All port C outputs are normal CMOS outputs.
1: All port C outputs act as open-drain outputs.

HNDS: *Handshake/simple strobe mode select*
0: simple strobe mode
1: handshake mode

OIN: *Output/input handshake*
0: input handshake
1: output handshake

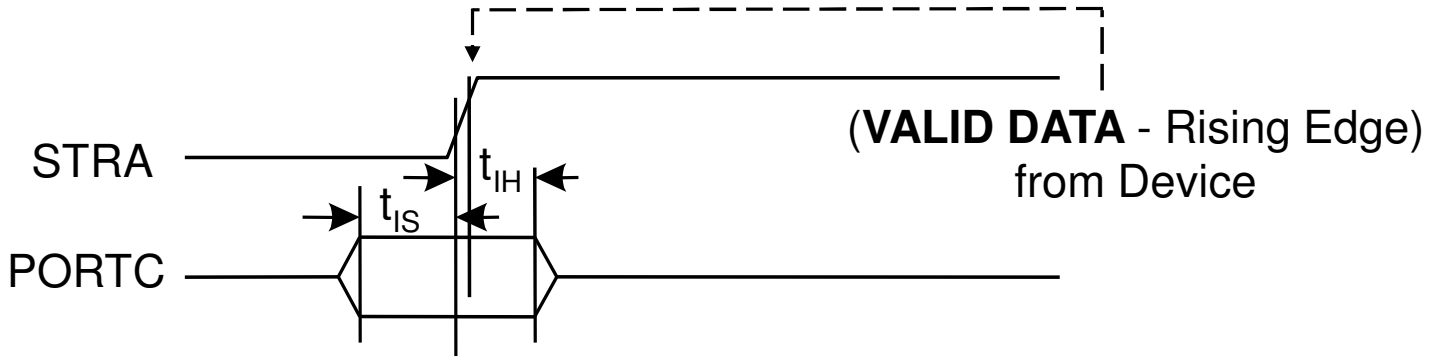
PLS: *Pulse/interlocked handshake operation*
0: interlocked handshake selected
1: pulse handshake selected

EGA: *Active edge for STRA*
0: falling edge
1: rising edge

INVB: *Invert STRB*
0: STRB active low
1: STRB active high

Simple Strobe I/O Mode

- Selected when HNDS = '0' (default upon RESET)
- Port C becomes the strobe input port.
- STRA active edge latches the values of port C pins into PORTCL reg.



t_{IS}: input setup time (60 ns at 2 MHz)

t_{IH}: input hold time (100 ns at 2 MHz)

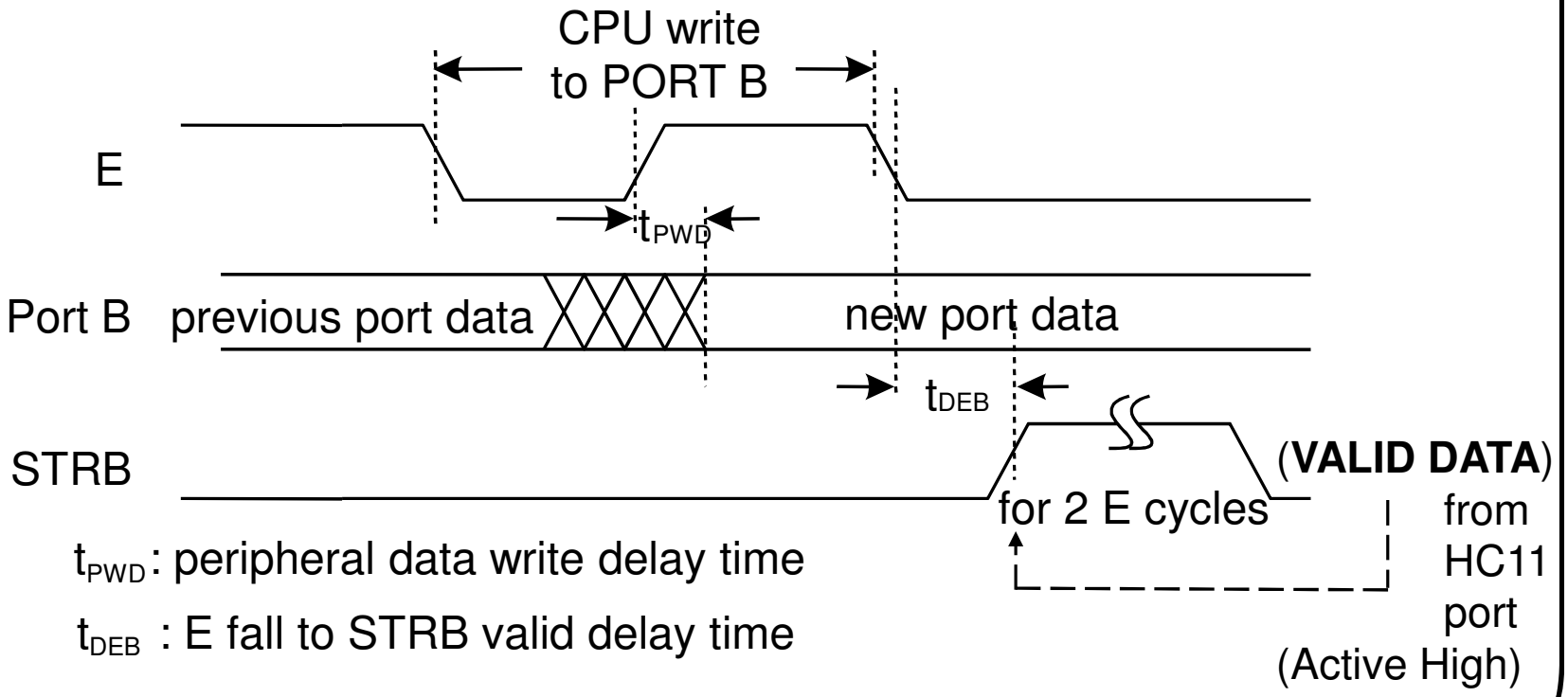
Port C strobed input timing

Simple Strobe Mode - Input (Port C)

- Bit 1 of the PIOC register (EGA) selects the active edge of the STRA pin.
- Reading PORTC register returns the current values on Port C pins.
- Reading PORTCL register returns Port C values latched with last STRA active edge.
- When enabled (STAI = '1'), the active edge of the STRA signal will request an interrupt to the CPU.
- The STRA interrupt vector is at $FFF2:FFF3 (same as for $\overline{\text{IRQ}}$ pin).
- STAF clearing sequence:
    1. Read PIOC register.
    2. Read PORTCL register.

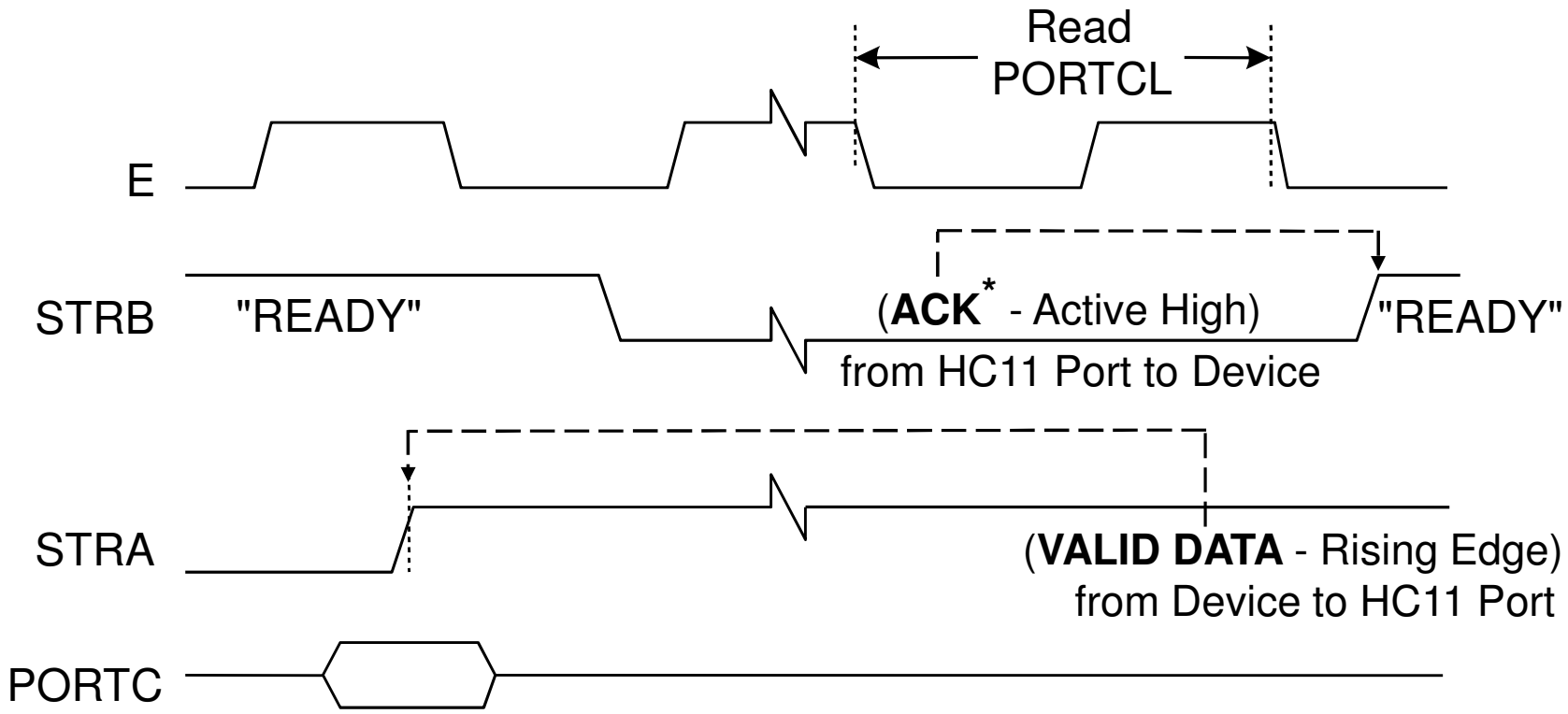
Simple Strobe Mode - Output (Port B)

The strobe signal STRB is pulsed for two E clock cycles each time there is a write to port B.



Port B strobed output timing

Port C Interlocked Input Handshake Protocol

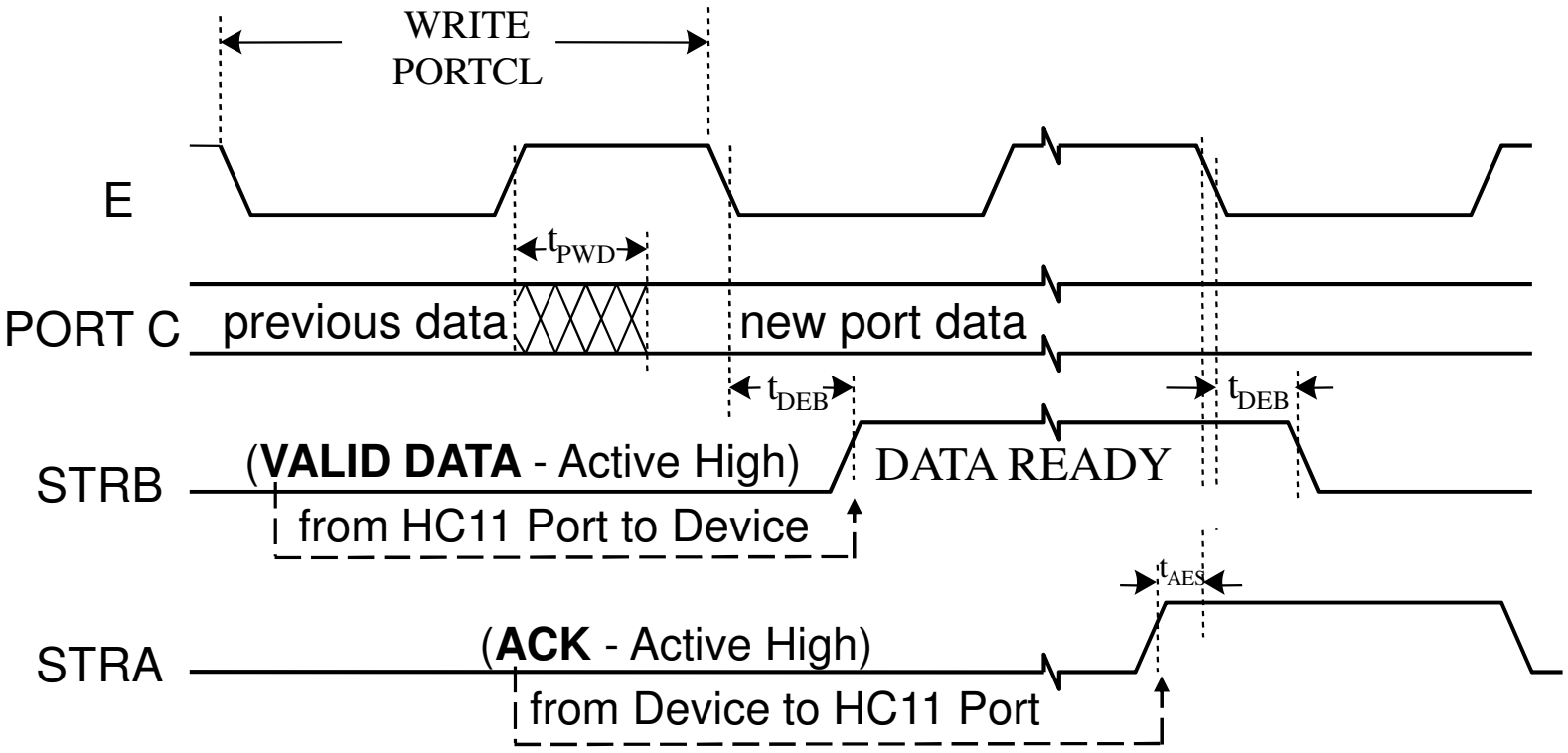


* **ACK** - Acknowledge line / signal

Port C Input Handshake Protocol

- STRA is a **valid** data latch command asserted by the input device (active edge is rising in previous figure).
- STRB is an **acknowledge**/**ready** output driven by the 68HC11 (active high in figure).
- When ready for accepting new data, the HC11 asserts (or pulses) STRB pin.
- The input device places data on port C pins and asserts the STRA signal.
  1) The active edge of STRA latches data into the PORTCL register,
  2) Sets the STAF flag in PIOC register, and
  3) De-asserts the STRB signal.
     - The de-assertion of STRB inhibits the external device from strobing new data into port C.
     - New data can be applied on port C pins once the CPU reads PORTCL.
- STAF clearing sequence:
  1. Read PIOC register.
  2. Read PORTCL register.

Port C Interlocked Output Handshake Protocol



t_{PWD} : Peripheral data write delay time, 150 ns max (at 2 MHz)

t_{DEB} : E fall to STRB delay, 225 ns (at 2 MHz)

t_{AES} : STRA asserted to E fall setup time, 0 ns (at 2 MHz)

Port C Output Handshake Protocol

- **STRA** is an **acknowledge** input (driven by the external device)
- **STRB** is a **valid** data or data ready output (driven by the 68HC11)
- In the figure, STRA activates with rising edge and STRB is active high.

Protocol sequence:

(a) The 68HC11 writes data into PORTCL and then asserts **STRB** to indicate that there are **valid** data on port C pins.

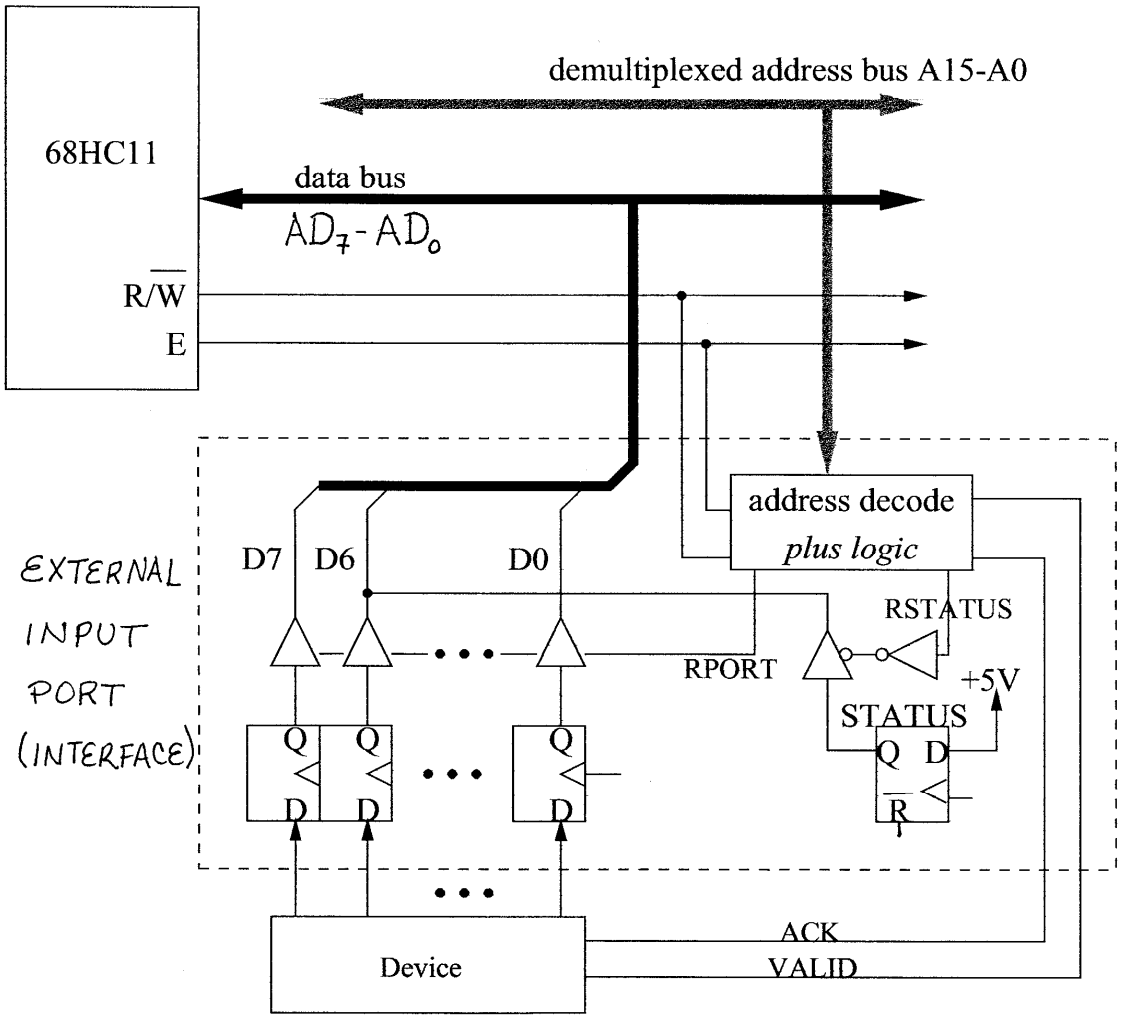
(b) The external device then asserts **STRA** to **acknowledge** the receipt of data which will then cause STRB to be de-asserted and the STAF flag to be set.

(c) After the de-assertion of STRB, STRA is also de-asserted.

- STAF clearing sequence:
    1. Read PIOC register.
    2. Write PORTCL register.

Code Example using Simple Strobe I/O

```
; Addresses
PIOC        equ   $1002   ; Address for the Parallel I/O Control register
PORTB       equ   $1004   ; Address for Port B data register
PORTCL      equ   $1005   ; Address for Port C Latched data register

; Constants
BITMASK     equ   $80     ; Bit mask for STAF flag in PIOC register

; Program Code
            org   $C000         ; To be loaded at $C000

            ldaa  #BITMASK      ; ACCA ← STAF bit mask
loop        bita  PIOC          ; Is STAF Set?
            beq   loop          ; If NOT loop back & wait until it is
            ldab  PORTCL        ; If YES, ACCB ← data latched in PORTCL
            stab  PORTB         ; Sending latched value just read to Port B
            bra   loop          ; Branch back and keep polling STAF for
                                ; next valid input data in Port C.
            end
```



External Parallel Port Example

Code for Servicing the External Input Port

```
PSTATUS     equ   $8000      ; Port Status Address
PDATA       equ   $8001      ; Port Data Address
BUF_START   equ   $D000      ; Start of Buffer
BUF_END     equ   $D0FF      ; Last element in Buffer

            org   $C000      ; Start of code

            ldaa  PDATA      ; Dummy read to clear port status
            ldx   #BUF_START ; Initialize IX pointer
POLL        ldaa  PSTATUS    ; Get current status
            anda  #%01000000 ; Check B6 by masking other bits
            beq   POLL       ; If status is still zero keep polling
            ldaa  PDATA      ; If NOT, get data from port
            staa  0,X        ; Store it in Buffer using IX pointer
            inx              ; Update Buffer pointer
            cpx   #BUF_END   ; Check if Buffer end has been reached
            bls   POLL       ; If NOT, go back wait for new data from port
            swi              ; If YES, exit to BUFFALO

            end
```

Alternative to define Buffer space

```
            org     $D000   ; Start of Buffer

BUF_START   ds      !255    ; One element less than total space
BUF_END     ds      1       ; Last element (to allow labeling it)
```

Alternative check for program end

```
            :
            staa    0,X             ; Store it in Buffer using IX pointer
            cpx     #BUF_END        ; Check if last element written
            beq     DONE            ; If YES, finish program
            inx                     ; If NOT, update Buffer pointer &
            bra     POLL            ; Go back wait for new data from port
DONE        swi                     ; Exit to BUFFALO
```