

Signed Integers - 2's Complement

Whereas in normal signed integers, we use a sign-magnitude system, binary uses a continuous (or offset) system such that $(-1) + 1 = 0$

Subtraction and addition are two different processes in decimal. We look at the signs of the operands to determine which operation to perform.

For example

$$(-A) + B \Rightarrow B - A$$

$$(-A) - (-B) \Rightarrow B - A$$

$$A - (-B) \Rightarrow A + B$$

$$(-A) + (-B) \Rightarrow -(A + B)$$

We can do this in binary, we build a separate subtractor using different truth table, from an adder, but it's two large bits of logic. We prefer a system based on the idea $(x + (-x)) = 0$, i.e. all subtractors are based on adding the negative equivalent, or $A - B \Rightarrow A + (-B)$, thus $(-x) = 2^n - x$ hence the 2's Complement.

1st Complement is the inverse of x

2nd Complement = 1st Complement + 1

$$\begin{array}{r}
 x = 1001 \\
 1^{\text{st}} \text{ Complement} = 0110 \\
 \hline
 + = 1111 \\
 +1 \qquad \qquad 1 \\
 \hline
 \end{array}$$

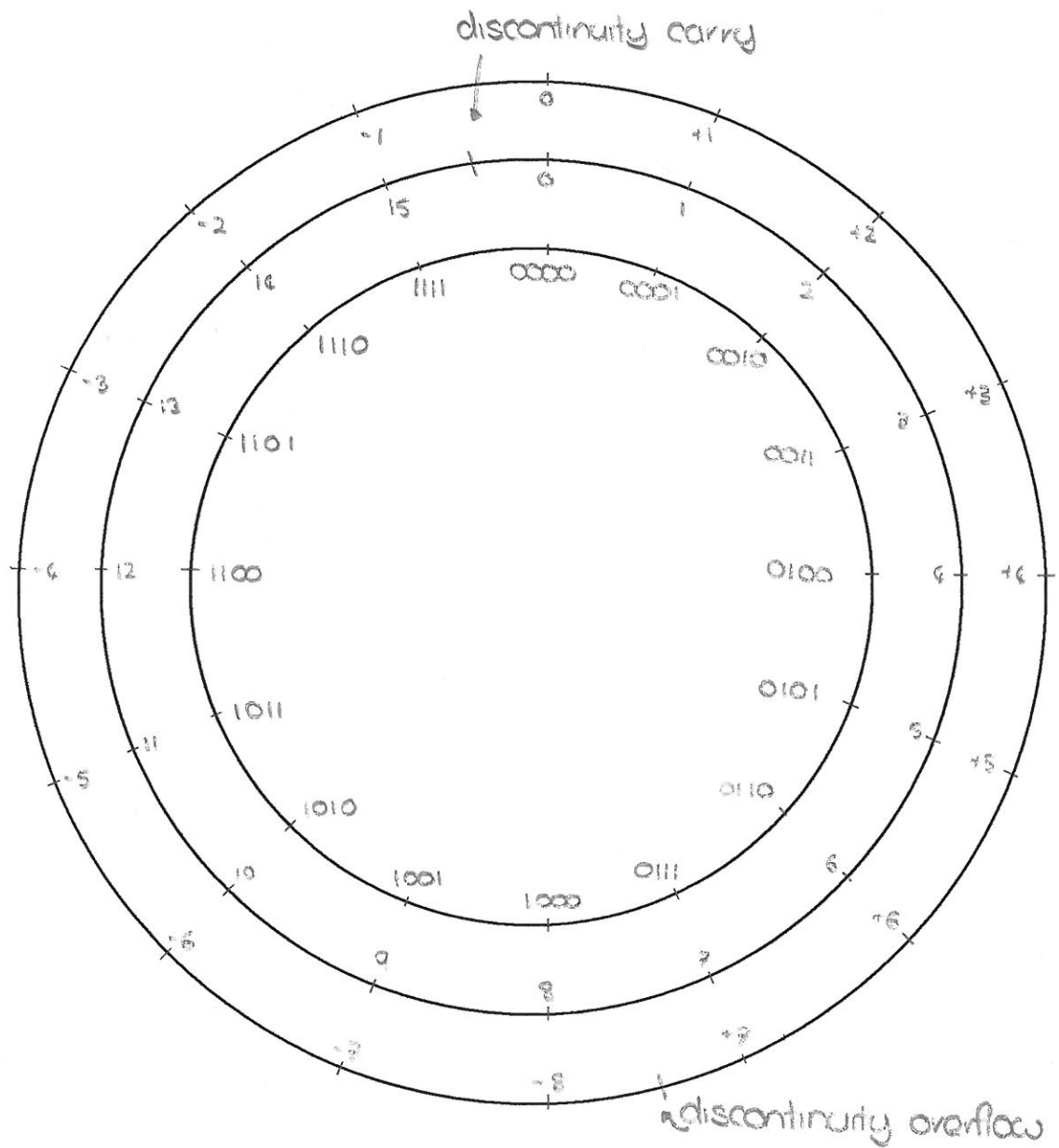
$(1) 0000$
 always ignore the carry in 2's complement

hence we can form a negative integer by inverting its positive value and adding 1 to it. We will see that we can subtract 2's complement numbers.

$$A - (-B) \Rightarrow A + (- - B)$$

But Arithmetic Logic units (ALU) will subtract unsigned numbers (using 2's complement method). From the cyclic 4-bit sequence we see that....

1. Carry flag is only applicable when working in unsigned integer code.
2. In 2's complement $(-1) + 1 = 0$ using normal arithmetic rules. Hence, the carry flag is irrelevant if we are using 2's complement code, it's still irrelevant if subtracting positive integers.
3. 2's complement number system is continuous through zero.
4. The 2's complement discontinuity is between $0111 \leftrightarrow 1000$ and is indicated by the overflow (v) flag.
5. MSB (most significant bit) indicates sign, but not sign magnitude code



2's Complement examples

1. Assume a positive integer mod

$$\begin{array}{r} 3 \\ + 7 \\ \hline 10 \end{array}$$

$$\begin{array}{r} 0011 \\ 0111 \\ \hline 1010 \end{array}$$

$$\begin{array}{r} 11 \\ + 6 \\ \hline 17 \end{array}$$

$$\begin{array}{r} 1011 \\ 0110 \\ \hline (1) 0001 \end{array}$$

↑ carry weight (16)

$$\begin{array}{r} 14 \\ - 5 \\ \hline 9 \end{array}$$

$$\begin{array}{r} 1110 \\ 0101 \\ \hline 1001 \end{array}$$

Initially we use subtract rules

$$\begin{array}{r} 4 \\ - 6 \\ - 2 \end{array}$$

$$(1) 1110$$

14 not -2 due to positive mod

Borrow (carry) weight (-16) $\frac{-16}{+16} = -2$

Prob: the subtractions using 2's complement method

$$14 - 5 \quad 5 = 0101$$

$$1^{\text{st}} = 1010$$

$$2^{\text{nd}} = 1011 \quad (-5)$$

$$+16 = 1110 \quad (16)$$

$$(1) 1001 \quad (9)$$

inverted due to 2nd of 2's complement, so borrow logic, invert the setting in c flag.

6-6

$$6 = 0110$$

$$1^{\text{st}} = 1001$$

$$2^{\text{nd}} = 1010$$

$$+6 = 0100$$

$$(0) 1110$$

inverted to c-flag = 1 i.e., borrow of $-16+16$ in result

2. Assume the 2's complement code

6

$$0100$$

+(-2)

$$1110$$

2

$$(1) 0010$$

Carry flag is ignored in 2's complement code.

3

$$0011$$

+(-6)

$$1010$$

-3

$$(0) 1101$$

(-2)

$$1110$$

+(-5)

$$1011$$

-7

$$(1) 1001$$

Overflow flag

$3+6$

$$\begin{array}{r} 0011 \\ 0110 \\ \hline 1001 \end{array}$$

the v-flag is set since
(+ve) + (+ve) gives a positive answer

$(-5)+(-6)$

$$\begin{array}{r} 1011 \\ 1010 \\ \hline (1) 0101 \end{array}$$

the v-flag is set since
(-ve) + (-ve) gives a positive answer

$5-(-7)$

$$\begin{array}{r} 0101 \\ 0111 \\ \hline 1100 \end{array}$$

the v-flag is set since
(+ve) - (-ve) gives a positive answer

$(-6)-(-6)$

$$\begin{array}{r} 1010 \\ 1100 \\ \hline 0110 \end{array}$$

the v-flag is set again

In 2's complement, sign bit is replicated in most significant bits, as word size is increased.

example $-C = C \Rightarrow FC$
 $\Rightarrow FFFC$

$-5 = 5 \Rightarrow 05$
 $\Rightarrow 0005$

The count starts from the next op-code, the rr byte is calculated with the program counter already pointing to the next op-code

example wait, BRCLR, wait, $-rr = -C(FC)$

PC = 010C

rr = FFFC

(I) = 0108

Compare Instruction

This is a subtract operation without storing the result, but flags are set, thus

$$Acc = M$$

$$Acc < M$$

$$Acc > M$$

$$Acc \leq M$$

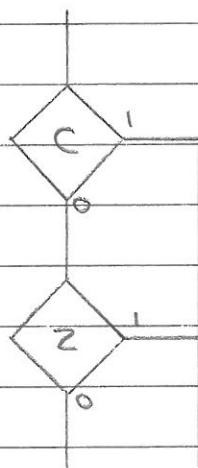
$$Z = 1$$

$$C = 1$$

$$C = Z = 0$$

$$C = 1 \text{ or } Z = 1$$

With a simple microcontroller with only single flag tests, $C + Z = 1$ must be constructed as a "tree" of decisions.



The HC11 gives complex test instructions (branch if lower or same).

Stacks

we have already met the concept of a stack, as a mechanism to store subroutine return addresses. we saw that its life structure allowed "nesting" of these addresses. However, organised "stacking" and "unstacking" operations must be complementary, both in themselves (micro-code) and when used by the programmer.

PSHA

PSHB

...

PULA

PULB

this will reverse the contents of AccA and AccB, so not complementary.

PSHA

PSHB

...

PULB

PULA

this is correct

In practice, most microcontroller stacks grow downwards and because their size can vary (the number of subroutines nested, amount of data stacked) they are usually implemented in data memory, hence the visibility of the stack pointer (sp) to control the stack address. On the HC11, the stack pointer, points to the next available location on the stack. Thus a "stacking" operation is write $sp = sp - 1$, and hence a "unstacking" operation is read $sp = sp + 1$.

BSR write (lower byte)
 $SP = SP - 1$
 write (higher byte)
 $SP = SP - 1$

RTS $SP = SP + 1$
 read (higher byte)
 $SP = SP + 1$
 read (lower byte)

These operations are in microcode and are transparent to the programmer, the program should only ever initialise SP.

It is unfortunate that program return addresses and data are mixed on the same stack. The stack is a convenient way of storing variables temporarily, for example, if the accumulator AccA is to be used in a subroutine but already contains a variable to be reused, we would

PSHA (at start of subroutine)
PULA
RTS (at the end)

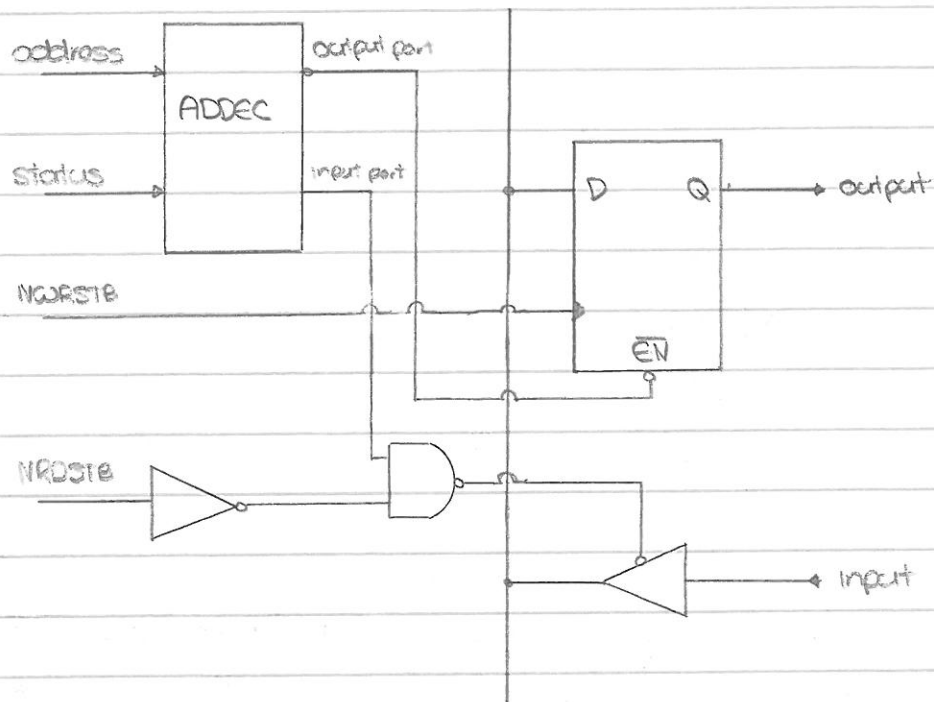
lets say you wanted to pass a parameter to subroutines, using a stack.

⋮	
PSHA	
BRA Fred	Fred PULA
PULA	⋮
⋮	PSHA
	RTS

○ this method does not work! A solution is by having an internal stack for subroutine return addresses, overcomes this, avoids corrupting addresses, makes BSR / RTS totally transparent, and can be a reasonable size.

○

Basic Input/output (I/O) circuit



The I/O is a simple input and output port. It contains three main components, they are the Address decoder (ADDEC), the output port, and the input port.

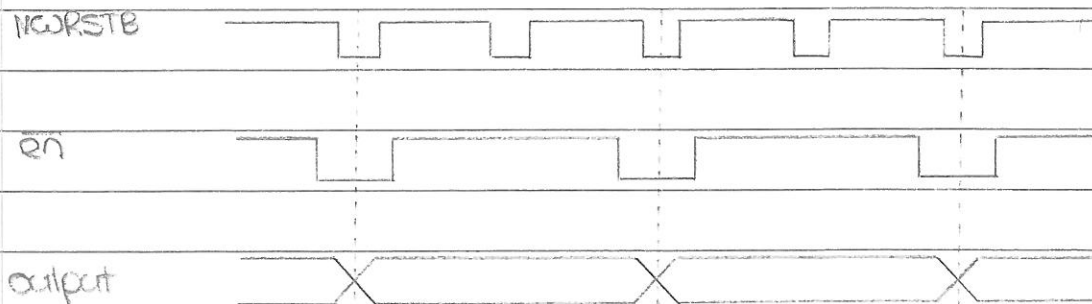
① Address Decoder

The address decoder or (ADDEC) for short is responsible for decoding the address and status bus information and selecting individual locations (enable). Essentially it is an AND function. To decode the 86 address is....

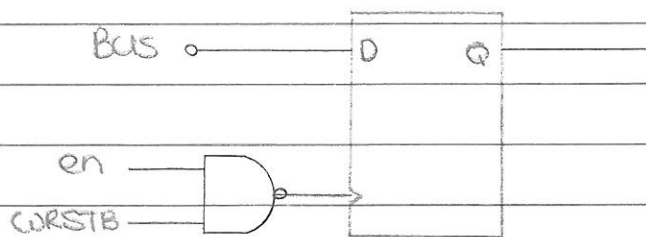
$$10110110 = A_7 \bar{A}_6 A_5 A_4 \bar{A}_3 A_2 A_1 \bar{A}_0$$

2. Output port

The output port is 8-bits deep, and is basically a clock enabled D-type ie, it acts as a D-type only if enable is active. The port retains the information set to it until it is written to again.



we try to avoid 'gated' clock type circuits



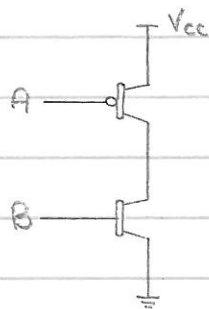
because they introduce delays to things and can cause false edges, will be seen as clock edges by the edge-triggered D-type. Remember 'en' is generated through a decoder, which can generate switching noise (glitches) due to different path lengths through the logic.



if signals change at the same time then a glitch occurs due to a delay as signal goes through the not gate.

3. Input port

The input port is a tri-state buffer, this port connects the input to the Bus, if it's enabled. The port buffers the signal (regenerates logic levels). If it is disabled, the output of the buffer is isolated, allowing other devices to drive Bus. This is a tri-state buffer whose output is connected together, must be enabled exclusively, otherwise you will short together two or more ordinary totem pole outputs. This arrangement forms a multiplexer.



totem pole $A=B$, either pull up or pull down

tristate off $A \neq B$ both disable their drive transistors

The apparent "gated clock" arrangement here is not a clock but an enable in the data path. For an input operation the data Bus is latched into the accumulator in the CPU i.e., the accumulator is a clocked device. Data signal path signals can tolerate initial hazard noise, to a limited extent.

V_{DD} = 5.0 Vdc ± 10%, V_{SS} = 0 Vdc, T_A = T_L to T_H

Num	Characteristic	Symbol	1.0 MHz		2.0 MHz		3.0 MHz		Unit
			Min	Max	Min	Max	Min	Max	
	Frequency of Operation (E-Clock Frequency)	f _o	dc	1.0	dc	2.0	dc	3.0	MHz
1	Cycle Time	t _{cy}	1000	—	500	—	333	—	ns
2	Pulse Width, E Low PW _{EL} = 1/2 t _{cy} - 23 ns (Note 1)	PW _{EL}	477	—	227	—	146	—	ns
3	Pulse Width, E High PW _{EH} = 1/2 t _{cy} - 28 ns (Note 1)	PW _{EH}	472	—	222	—	141	—	ns
4a	E and AS Rise Time	t _r	—	20	—	20	—	20	ns
4b	E and AS Fall Time	t _f	—	20	—	20	—	15	ns
9	Address Hold Time t _{AH} = 1/8 t _{cy} - 29.5 ns (Note 1, 2a)	t _{AH}	95.5	—	33	—	26	—	ns
12	Nonmuxed Address Valid Time to E Rise t _{AV} = PW _{EL} - (t _{ASD} + 80 ns) (Note 1, 2a)	t _{AV}	281.5	—	94	—	54	—	ns
17	Read Data Setup Time	t _{DSR}	30	—	30	—	30	—	ns
18	Read Data Hold Time (Max = t _{WAD})	t _{DHR}	0	145.5	0	83	0	51	ns
19	Write Data Delay Time t _{DOW} = 1/8 t _{cy} + 65.5 ns (Note 1, 2a)	t _{DOW}	—	190.5	—	128	—	71	ns
21	Write Data Hold Time t _{DHW} = 1/8 t _{cy} - 29.5 ns (Note 1, 2a)	t _{DHW}	95.5	—	33	—	26	—	ns
22	Muxed Address Valid Time to E Rise t _{AVM} = PW _{EL} - (t _{ASD} + 90 ns) (Note 1, 2a)	t _{AVM}	271.5	—	84	—	54	—	ns
24	Muxed Address Valid Time to AS Fall t _{ASL} = PW _{ASH} - 70 ns (Note 1)	t _{ASL}	151	—	26	—	13	—	ns
25	Muxed Address Hold Time t _{AHL} = 1/8 t _{cy} - 29.5 ns (Note 1, 2b)	t _{AHL}	95.5	—	33	—	31	—	ns
26	Delay Time, E to AS Rise t _{ASD} = 1/8 t _{cy} - 9.5 ns (Note 1, 2a)	t _{ASD}	115.5	—	53	—	31	—	ns
27	Pulse Width, AS High PW _{ASH} = 1/4 t _{cy} - 29 ns (Note 1)	PW _{ASH}	221	—	96	—	63	—	ns
28	Delay Time, AS to E Rise t _{ASED} = 1/8 t _{cy} - 9.5 ns (Note 1, 2b)	t _{ASED}	115.5	—	53	—	31	—	ns
29	MPU Address Access Time (Note 2a) t _{ACCA} = t _{cy} - (PW _{EL} - t _{AVM}) - t _{DSR} - t _r	t _{ACCA}	744.5	—	307	—	196	—	ns
35	MPU Access Time t _{ACCE} = PW _{EH} - t _{DSR}	t _{ACCE}	—	442	—	192	—	111	ns
36	Muxed Address Delay (Previous Cycle MPU Read) t _{WAD} = t _{ASD} + 30 ns (Note 1, 2a)	t _{WAD}	145.5	—	83	—	51	—	ns

NOTES:

1. Formula only for dc to 2 MHz.

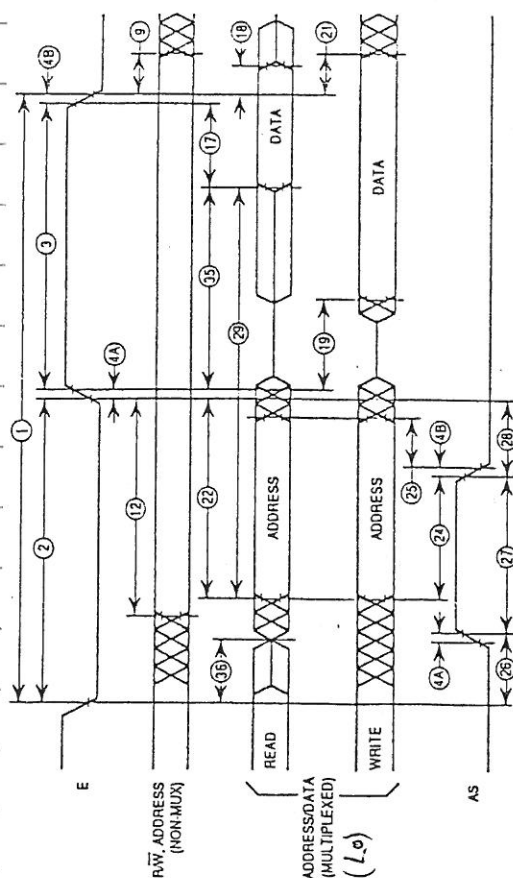
2. Input clocks with duty cycles other than 50% affect bus performance. Timing parameters affected by input clock duty cycle are identified by (a) and (b). To recalculate the approximate bus timing values, substitute the following expressions in place of 1/8 t_{cy} in the above formulas, where applicable:

- (a) (1-DC) × 1/4 t_{cy}
(b) DC × 1/4 t_{cy}

Where:

DC is the decimal value of duty cycle percentage (high time).

3. All timing is shown with respect to 20% V_{DD} and 70% V_{DD}, unless otherwise noted.



NOTE: Measurement points shown are 20% and 70% of V_{DD}.

The HC11 Bus shown is a synchronous Bus, the CPU dictates all timings and peripheral devices must work at its speed.

These are semi-synchronous Buses which by differing mechanisms allow a slow peripheral to "stretch" or "extend" a Bus cycle till the peripheral is ready to continue.

$t_{\text{set-up}}$ and t_{hold}

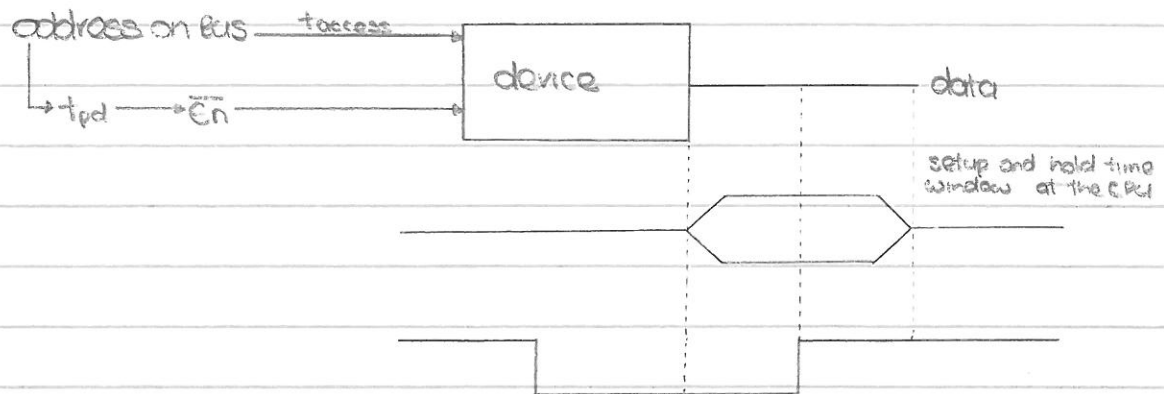
The set-up and hold time define a window around the active clock edge. Here the trailing edge of $\overline{\text{NRDSTB}}$ to the CPU, and the $\overline{\text{NWRSTB}}$ to external output ports. When the data is stable for the latch to capture the data accurately.

t_{access}

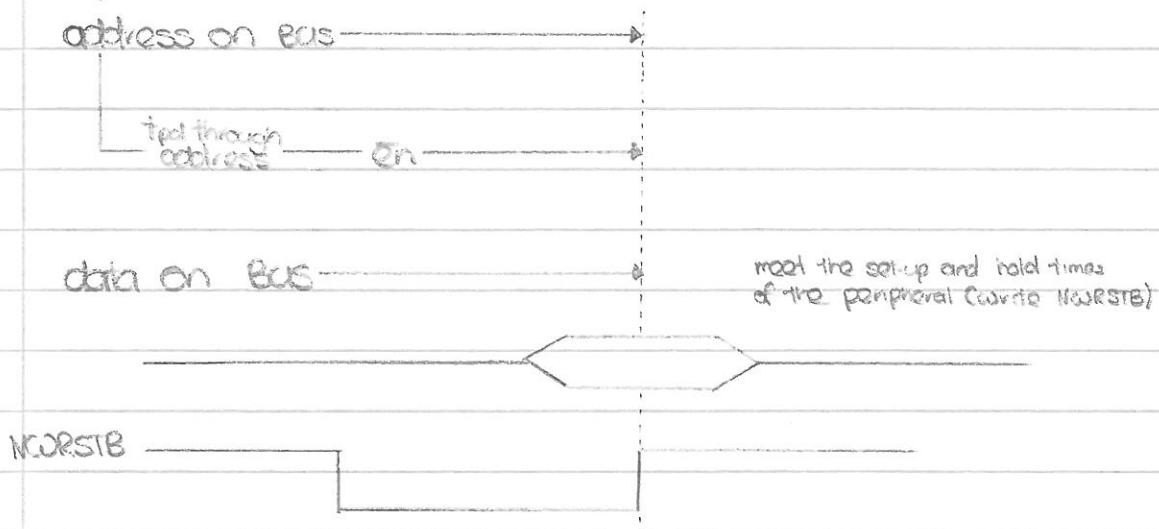
The access time is the time it takes the peripheral to produce data, i.e. propagation delay through the peripheral. The access time may be from address or from chip select ($\overline{\text{CS}} \equiv \overline{\text{En}}$)

Bus interface Timing Equations

Input



Output



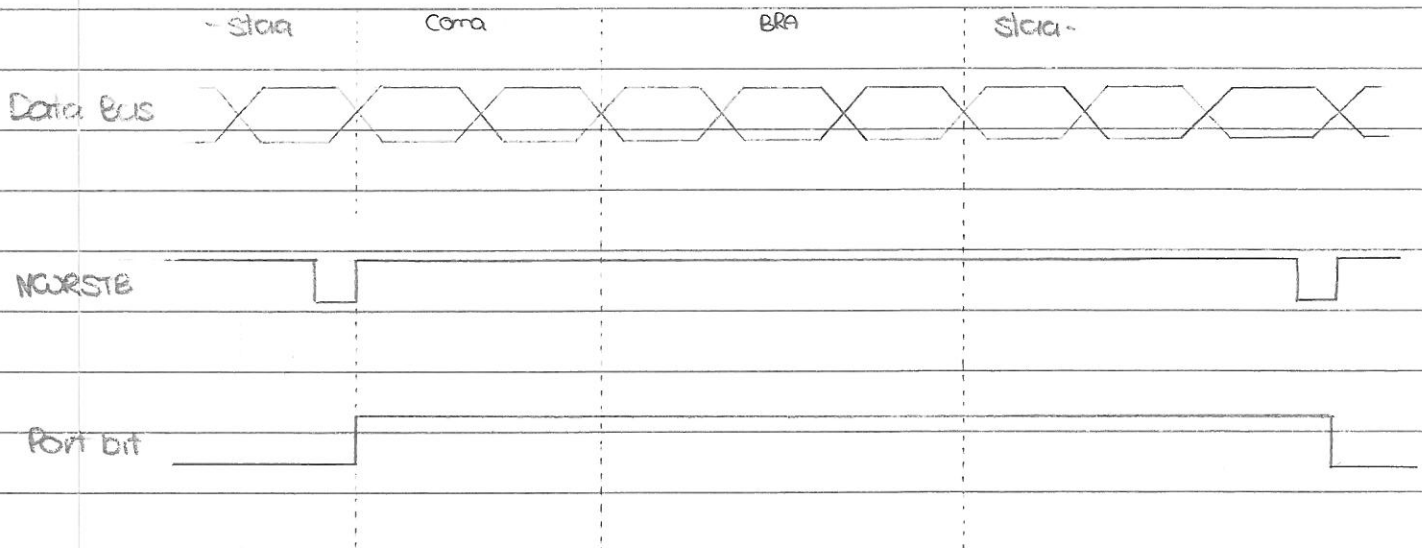
Input/output (i/o) port characteristics

Output

We saw that a basic output port was a latching device thus information sent to an output port is held until the next output is sent to that output port. This is suitable for combinational (continuous) output devices, for example, LED's, indicators, and DAC (digital to analogue converter)

	cycles
• STAA	1
└ Coma	2
└ BRA	3

total = 9 cycles (at 2MHz a 4.5µs loop time)

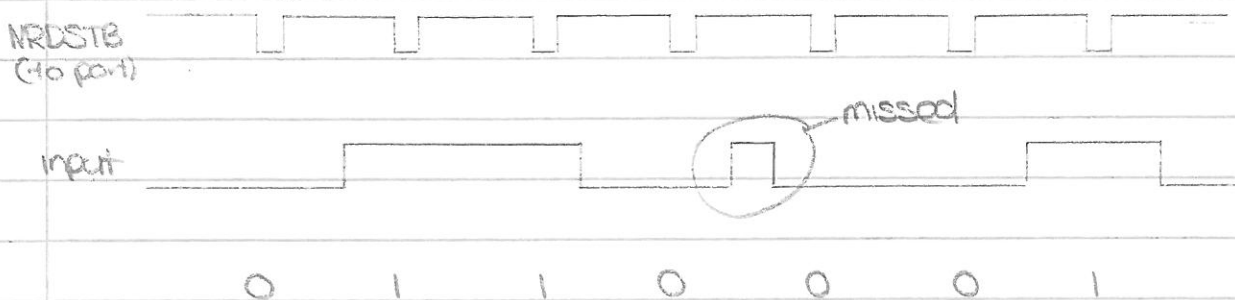


Input

The basic input port is a combinational device and hence repeated input operations sample the data on the input port at that time. This is known as "polling". Input data which comes and goes between samples (or polls) will not be seen, unless a more complex latching input port is devised.

port	cycles
LDAA	4
STAA table x	4
INCC	3
BRN	3

total = 14 cycles (at 2MHz, 7 μ s loop time)



The Nyquist Criteria says that we must sample at twice the maximum rate of the signal to be captured.

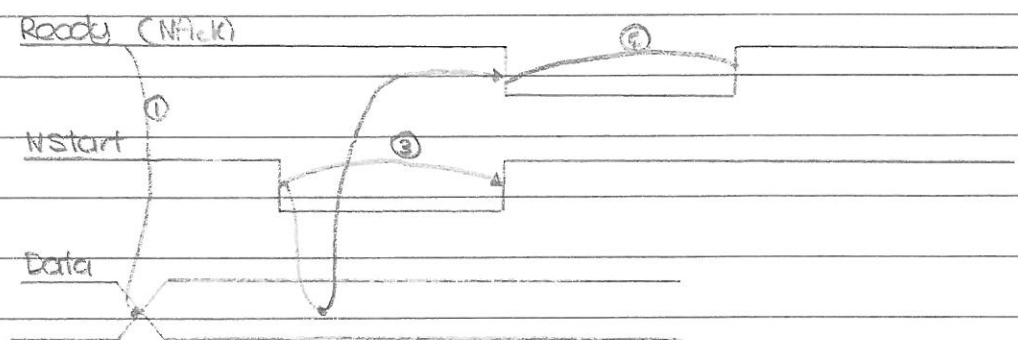
Handshakes

Frequently we need to transfer discrete data samples between devices and protocols are required to signal when valid data is available or to be used compared with continuous, combinatorial use. Such data must be transferred by a guaranteed and unique protocol. The commonest of such protocols is the ready-start handshake. This is a synchronous protocol, that allows the receiver to delay the sending of the next piece of data, i.e., to say when it is ready. Such a protocol can be implemented on a semi-synchronous Bus, but on a synchronous bus, with no feedback signal available to the CPU, we can only implement this at the programmed (I/O) level, i.e., we write a program to generate the required signals on the I/O ports.

Ready - Start Handshake

The ready-start handshake is a simple mechanised protocol for the transfer of data between two devices. This handshake essentially allows a slow device to synchronise the data transfer. We deal at the programmed I/O level over a synchronous bus, a semi-synchronous bus allows this protocol to be implemented at the bus transfer level.

The ready-start handshake is a "reduced" handshake, and as a result, it can fail. we will develop the full handshake later to detect possible transfer failure.



① The source checks the destination if it's ready to receive data, if so, it outputs data and takes "start" active.

② The destination waits for "start" to go active, when seen, it takes data and acknowledges, or goes "not ready" whilst it processes the data.

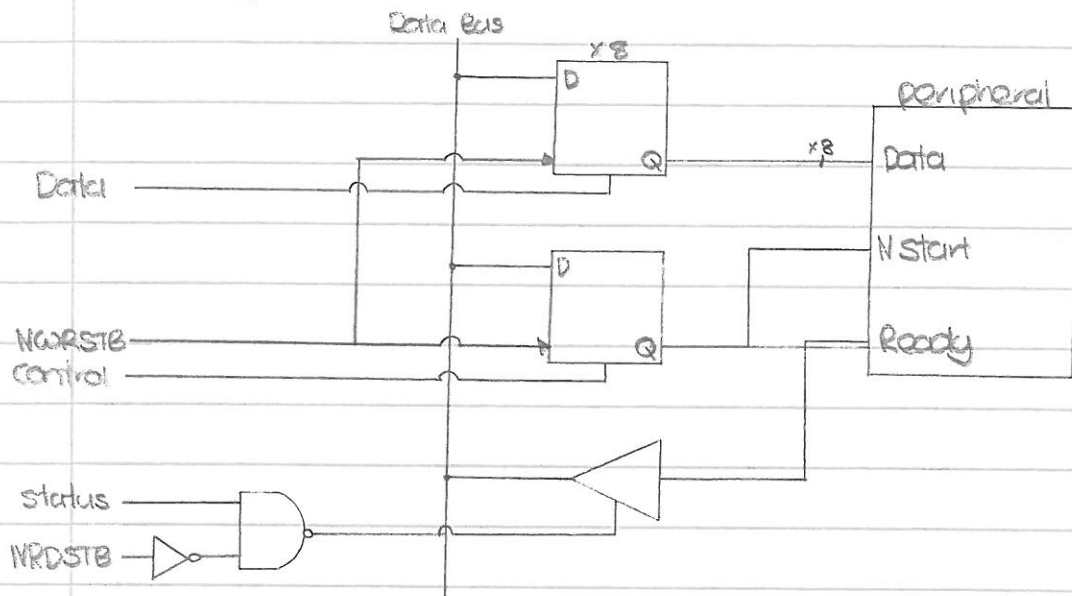
The processes of ① and ② are the same for the full handshake, the difference comes from ③ and ④, and are self timed.

③ The source takes the "start" as inactive. In its own time, maybe based on knowledge of destination, ie, requires a start pulse of $> 5\text{ms}$.

④ The destination will become ready again, in its own time, when it has used the data.

The data maybe removed after it is read, but in this handshake we don't detect when this is. Thus, it is likely to be left until changed at the start of the next transaction.

Input/output (I/O) circuit for Programmed I/O Ready-start handshake output from microcontroller

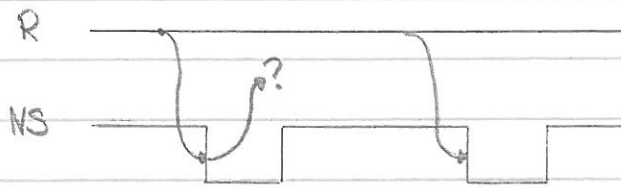


"data", "status", and "control" are outputs from the address decoder (ADDEC) corresponding to the address set by EQU directives, eg data EQU \$1027

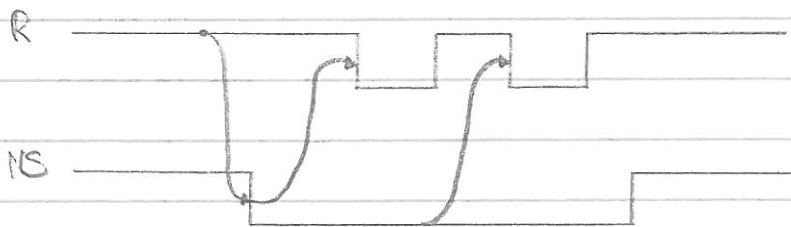
Driver to transfer byte from a microcontroller
Peripheral using a ready-start handshake

```
loop:  BRCLR (status-x), x           (ready bit loop)
        STAB data                   (data put into AccB)
        BCLR (control-x), x         (start bit)
        < delay if needed >
        BSET (control-x), x         (start bit)
        RST
```

Data Transfer failures

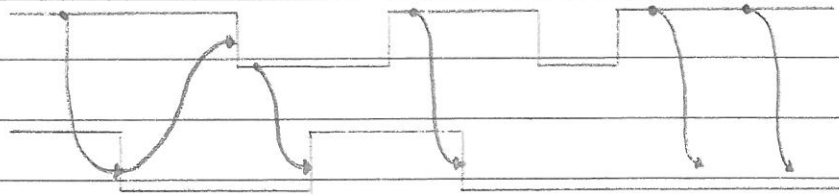


The destination does not respond as intended. The source does not detect this and continues to send data, but is not received. The solution is to check for acknowledge.



The start pulse is too long, or the destination response (Ack) is too quick. The destination responds more than once to same start pulse, ie, multiple copies are sent. The solution is to check the starts off before responding again.

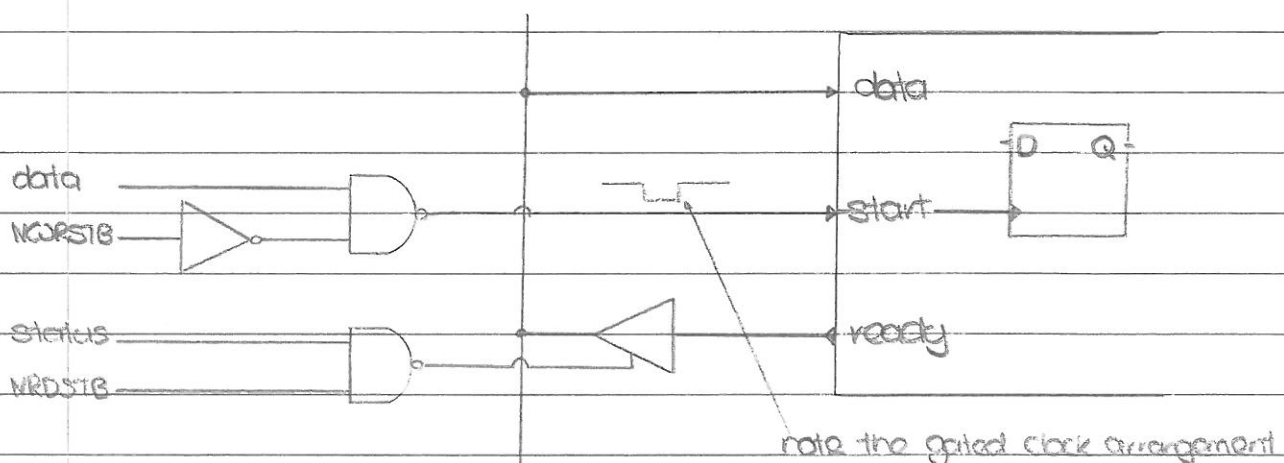
Assuming we look for "Ack" to counteract the first problem on the previous page.



If the acknowledge is too quick, the source assumes the data is not sent, and reports an error erroneously or resends data. The solution is to make acknowledge wait till NS is inactive.

Realistic Ready-start Handshakes

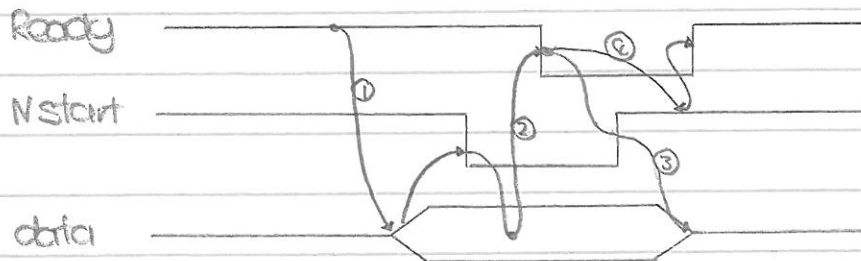
In many electronic peripherals the Nstart signal derived from the NCORSTB Bus signal, giving a hybrid implementation of the handshake. The data register in the peripheral, which is written by "Nstart" is mapped directly onto the microcontroller Bus. This requires the Nstart signal to be compatible with NCORSTB and other Bus timings.



It also requires access to the Bus, extra logic and address decoder. There is a loss of isolation/buffering between the Bus and external devices. A Programmed input/output (PIO) usually requires only a wiring harness between parts and the device, and can be extremely flexible, but is less efficient in run-time code.

full handshake

The full handshake gives a guaranteed, unique data transfer.



- ① and ② are identical to the ready-start handshake
- ③ The source checks the destination if it has responded before going inactive, data can now be removed
- ④ The destination after using the data checks start inactive before becoming ready again.

The ready start signal cannot now change phase without first checking the correct status of other party. This is what generates the unique data transfer even between asynchronous devices, all the signal edges are now synchronised to the other parties response.

If a peripheral does not implement a full handshake, the protocol cannot be imposed on it by the driver.

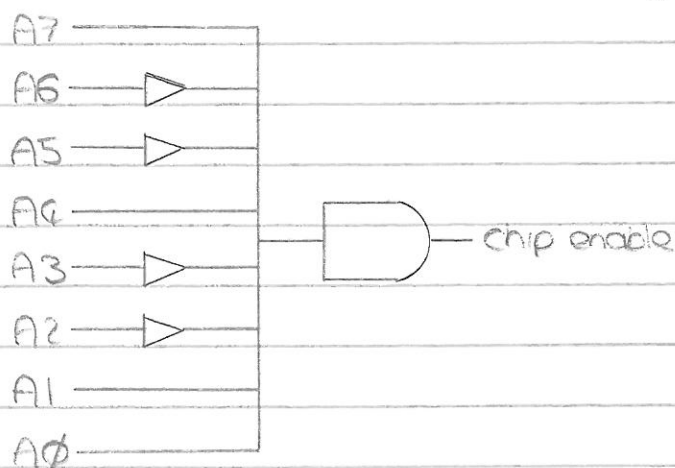
Address Decoder

The function of the address decoder (ADDEC) is to generate a unique chip enabled signal for the device being accessed from the address and status information on the Bus. It is essentially an AND operation, there are many practical ways of implementing an AND function. for example from an 8-bit bus to detect address 93₁₆ we are basically looking for the pattern.....

A7	A6	A5	A4	A3	A2	A1	A0
1	0	0	1	0	0	1	1

$$\text{chip enable (ce)} = A7 \cdot \bar{A6} \cdot \bar{A5} \cdot A4 \cdot \bar{A3} \cdot \bar{A2} \cdot A1 \cdot A0$$

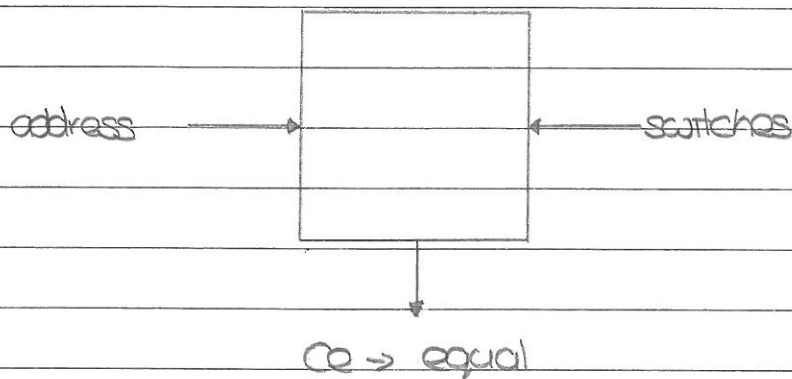
This can be implemented using an AND gate.....



but this is unlikely due to bulk, so the use of PLD's become useful because.....

- * they are reprogrammable
- * have large complexity
- * they have a SOP potential for complex decoder

another way of implementing the chip enable signal is to use an equality comparator for the address decoder (ADDEC).



$$\text{equal} = (A_3 S_3 + \overline{A_3 S_3})(A_2 S_2 + \overline{A_2 S_2}) \\ (A_1 S_1 + \overline{A_1 S_1})(A_0 S_0 + \overline{A_0 S_0})$$

$$\text{equal} = A_3 \overline{A_2} \overline{A_1} \overline{A_0}$$

Status Information

So far we have considered only the address bits, but we know we must also consider status (address qualifiers) as well.

M1 op-code

INTACK interrupt cycle

STACK stack access

RDNWR read cycle

MEMMIO memory access (not I/O port)

I/O mapping

Some processors (Intel based) have a separate I/O map (set of addresses) designed to accommodate I/O devices.

I/O mapping is usually half the size of the main memory address Bus size. This gives a simpler decoder for individual locations. This map was accessed only by I/O access instructions. for example when the IN-port, OUT-port instructions are executed (for the bus cycle accessing the port)

$$\text{MEMNIO} = 0$$

A peripheral is said to be I/O mapped, if the address decoder (ADDEC) function accessing it includes $\text{IOMEM} = 1$ or $\text{MEMVIO} = 0$. This device can now only be accessed by the special I/O access instructions (in-out). Conversely a peripheral can be memory mapped. We can also I/O map memories, but it doesn't make a lot of sense to do so.

We use I/O mapping to make a smaller address decoder (ADDEC), it uses direct addressing which is less likely to generate a wrong address and has to address the ports indirectly.

Example

Device to be read at I/O address $6C_{16}$ and written at I/O address 75_{16} . Note the I/O address is usually duplicated on the full address Bus, thus an I/O address A6 is distinguished from memory address A6A6 only by the IONMEM status bit. Use the tabular method of specifying the ADDEC functions.

		A7	A6	A5	A4	A3	A2	A1	A0	RDNWR	IONMEM	CE
read	$6C_{16}$	0	1	1	0	0	1	0	0	1	1	0
written	75_{16}	0	1	1	1	0	1	0	1	0	1	0

Example

Device to be read at I/O address C9 and written at memory address C9xx. Note C9xx mean any address between C900 and C9ff.

This is called "partial decoding" and is another technique to simplify the address decoder (Addrec) but it gives a block address, all of which accesses the one device. A block of 256 addresses is known as a page. There is a danger of partially decoded devices overlapping, for example C9xx and Cxx3 overlap. This can be used to your advantage for common access, but take care

		A7	A6	A5	A4	A3	A2	A1	A0	RDWR	stbm	ce
read	C9	1	1	0	0	1	0	0	1	1	1	0
write	C9	0	1	0	0	1	0	0	1	0	1	0

Example

Device which is read & written at memory 19xx
and at I/O 6x

		A7	A6	A5	A4	A3	A2	A1	A0	RDNOR	IONM	CE
read	19 ₁₆	0	0	0	1	1	0	0	1	x	0	0
write	6x ₁₆	0	1	1	0	x	x	x	x	x	1	0

Example

Device which is read and written at memory at
A6xx and I/O at A6.

		A7	A6	A5	A4	A3	A2	A1	A0	RDNR	IO/M	CE
read	A6	1	0	1	0	0	1	1	0	x	x	0
write	A6	1	0	1	0	0	1	1	0	x	x	0

↑ same because of A6xx and A6