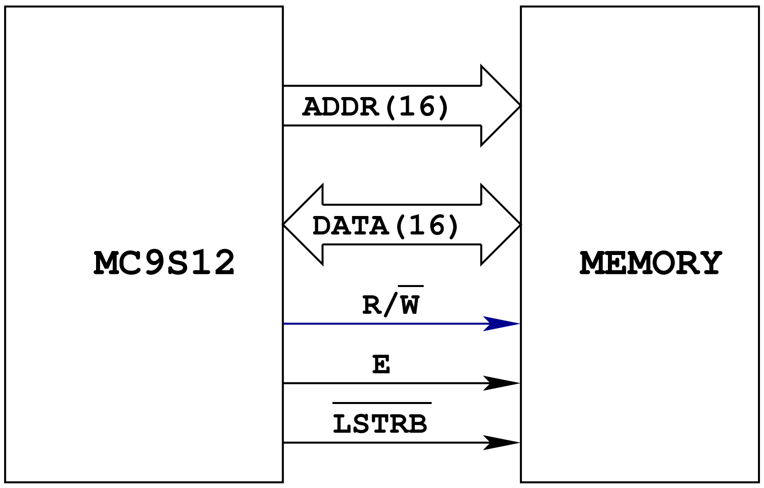
The Multiplexed Address/Data Bus



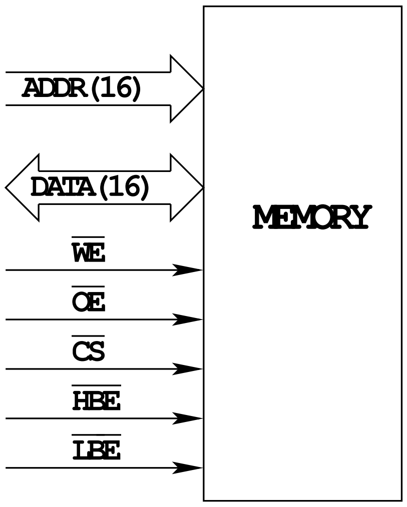
MC9S12 has 16-bit address and 16-bit data buses

Requires 35 bits

Not enough pins on MC9S12 to allocate 35 pins for buses and pins for all other functions

Memory Chip Interface

- Memory chips need separate address and data bus
  - Need way to de-multiplex address and data lines from MC9S12
- Memory chips need different control lines than the MC9S12 supplies
- These control lines are:
  - Chip Select – goes low when the MC9S12 is accessing memory chip
  - Write Enable – goes low when the MC9S12 is writing to memory
  - Output Enable – goes low when the MC9S12 is reading from memory
  - High Byte Enable – goes low when the MC9S12 is accessing the High Byte (Odd Address) of memory
  - Low Byte Enable – goes low when the MC9S12 is accessing the Low Byte (Even Address) of memory



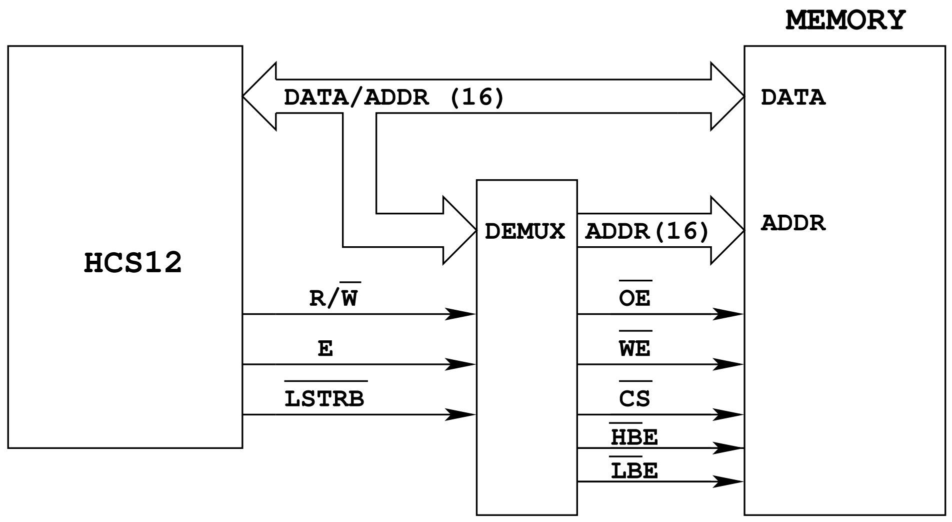
$\overline{WE}$ - Write Enable
$\overline{OE}$ - Output Enable (Read)
$\overline{CS}$ - Chip Select
$\overline{HBE}$ - High Byte Enable
$\overline{LBE}$ - Low Byte Enable

Memory needs separate address and data busses

Need way to separate address and data

The Multiplexed Address/Data Bus

- To talk to memory chip we will need to build a demultiplexer between the MC9S12 and the memory chip



$\overline{OE}$ - Output Enable (Read)
$\overline{WE}$ - Write Enable
$\overline{CE}$ - Chip Enable
$\overline{HBE}$ - High Byte Enable
$\overline{LBE}$ - Low Byte Enable

HCS12 has 16-bit address and 16-bit data buses

Requires 35 bits

Not enough pins on HC12 to allocate 35 pins
for buses and pins for all other functions

Solution: multiplex address and data buses

```
16-bit Bus:  While E low, bus supplies address
             While E high, bus supplies data
```

Accessing External Memory and Ports on the MC9S12 in Expanded Mode

- In expanded mode, the MC9S12 has a multiplexed 16-bit address and data bus.
- With a 16-bit address bus, the MC9S12 can access $2^{16} = 65,536$ bytes of data
- With a 16-bit data bus, the MC9S12 can access 16 bits (two bytes) in a single bus cycle
- In expanded mode, the MC9S12 uses Port A and Port B as the multiplexed address/data bus
- Timing is controlled by the E clock
- When the E clock is low, the MC9S12 places the address on the multiplexed bus
  - Port A is used for address bits 15-8
  - Port B is used for address bits 7-0
- When the E clock is high, the MC9S12 uses the multiplexed bus for data: bus
  - Port A is used for the byte at the even address
  - Port B is used for the byte at the odd address

  For example, if accessing the sixteen-bit word at address 0x4000 (the bytes at addresses 0x4000 and 0x4001), Port A will access the byte at address 0x4000, and Port B will access the byte at address 0x4001.

Byte Order in Microprocessors

- There are two ways to store bytes in a microprocessor memory. For example, if you wanted to store the 16-bit word 0x1234 into memory locations 0x2000 and 0x2001, you could do it in two ways:

	Big Endian		Little Endian	
Address	0x2000	0x2001	0x2000	0x2001
Byte	0x12	0x34	0x34	0x12

- Motorola and Freescale (and some other manufacturers) use Big Endian (big end, or most significant part, appears first in memory, big part is in lower part of memory)
- Intel (and some other manufacturers) use Little Endian (little end appears first, smaller part of the number is in lower part of memory)
- Data types of more than one byte written on a Motorola machine will not be read properly on an Intel machine without first swapping byte order (and vice versa).
- In the discussion which follows, even byte refers to a byte at an even address, odd byte refers to a byte at an odd address. High byte refers to the most significant byte of a 16-bit word, low byte refers to the least significant byte of a 16-bit word. For the MC9S12, the high byte is at the even address, and the low byte is at the odd address for a 16-bit access.

How to determine if a bus cycle accesses one or two bytes

- Sometimes you only want to access one byte at a time. For example,
  - `ldaa $4001`

  will access the single byte at address 0x4001.

- To determine whether it should access one byte or two bytes, the MC9S12 uses the $\overline{\text{LSTRB}}$ and A0 lines.
  - $\overline{\text{LSTRB}}$ low means that the MC9S12 is accessing the lower byte (byte at the odd address) of a sixteen-bit word
  - $\overline{\text{LSTRB}}$ high means that the MC9S12 is accessing the upper byte (byte at the even address) of a sixteen-bit word
  - A0 low means that the MC9S12 is accessing the upper (even) byte of a sixteen-bit word
  - A0 high means that the MC9S12 is accessing the lower (odd) byte of a sixteen-bit word

$\overline{\text{LSTRB}}$	A0	Type of Access
0	0	16-bit access of an even address Accesses bytes at even address and subsequent odd address
0	1	8-bit access of an odd address
1	0	8-bit access of an even address
1	1	Not allowed on external bus

- The instruction
  - `ldaa $4000`

  accesses the byte at address 0x4000, but doesn't access the byte at address 0x4001. For this access, the MC9S12 will put 0x4000 on the bus (A0 = 0, access byte at even address), and will make $\overline{\text{LSTRB}}$ = 1 (don't access byte at the odd address).

- The instruction
  - `ldaa $4001`

  accesses the byte at address 0x4001, but doesn't access the byte at address 0x4000. For this access, the MC9S12 will put 0x4001 on the bus (A0 = 1, do not access byte at even address), and will make $\overline{\text{LSTRB}}$ = 0 (access byte at odd address).

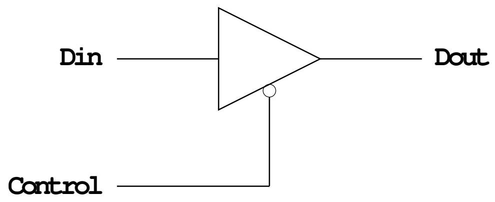
- The instruction
  - `ldd $4000`

  accesses the bytes at addresses 0x4000 and 0x4001. For this access, the MC9S12 will put 0x4000 on the bus (A0 = 0, access byte at even address), and will make $\overline{\text{LSTRB}}$ = 0 (access byte at odd address).

- What to check for on the bus to determine if the MC9S12 is accessing a particular byte
  - To check to see if the byte at address 0x4000 is being accessed, look for 0x4000 on the address bus (do not need to check $\overline{\text{LSTRB}}$).
  - To check to see if the byte at address 0x4001 is being accessed, look for either 0x4000 or 0x4001 on the address bus (i.e., A0 is a don't care), and make sure $\overline{\text{LSTRB}}$ is low.

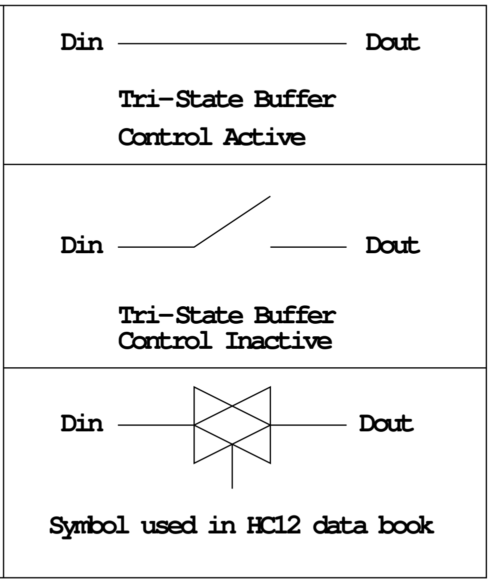
A Simple Parallel Input Port

- We want a port which will read 8 bits of data from the outside
- Such a port is similar to Port A or Port B when all pins are set up as input
- We need some hardware to drive the input data onto the data bus at the time the MC9S12 needs it to be there to read
- The hardware needs to keep the data off the bus at all other times so it doesn't interfere with data from other devices
- A **tri-state buffer** can be used for this purpose
  - A tri-state buffer has three output state: logic high, logic low, and high impedance (high-Z)
  - In high-Z state it is like the buffer is not connected to the output at all, so another device can drive the output
  - a tri-state output acts like a switch — when the switch is closed, the output logic level is the same as the input logic level, and when the switch is open, the buffer does not change the logic level on the output pin
  - A tri-state buffer has a control input which, when active, drives the input logic levels onto the output pins, and when inactive, opens the switch



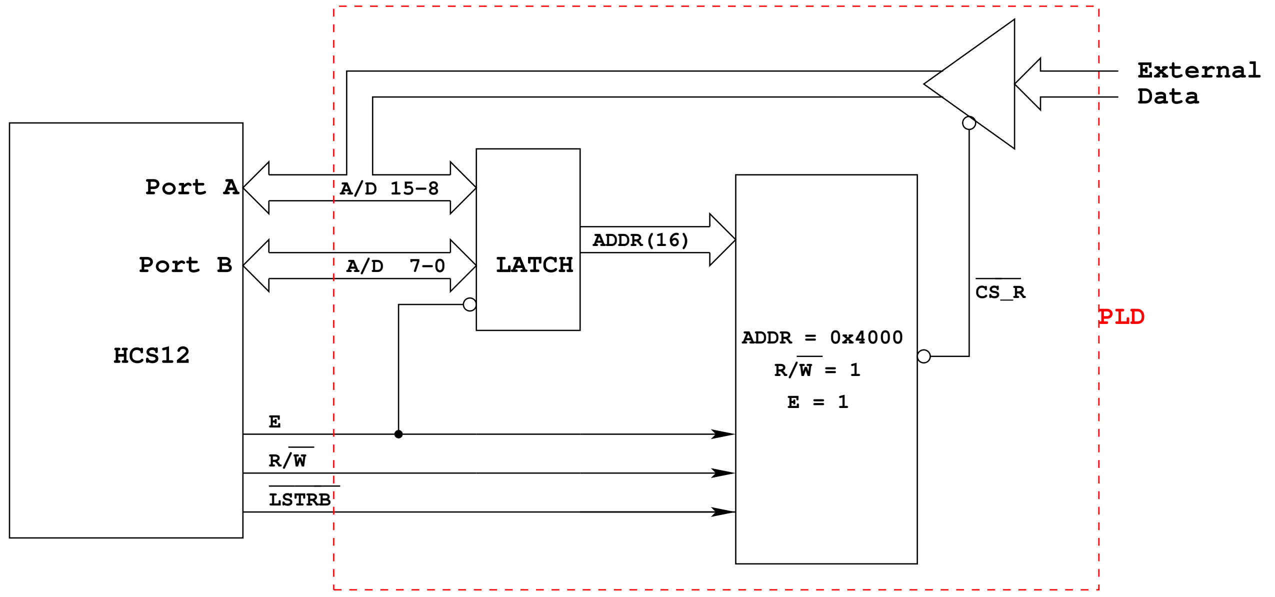
Tri-State Truth Table

Control	Din	Dout
0	0	0
0	1	1
1	X	High Z

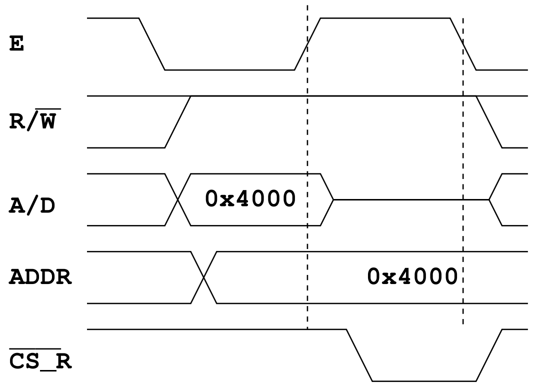


A Simple Parallel Input Port

- When should the tri-state buffer be enabled to drive the data bus?
  - The MC9S12 will access the buffer by reading from an address. We must assign an address for the tri-state buffer
  - We must have hardware to demultiplex the address from the data, and to determine when the MC9S12 is reading from this address
  - The 8-bit input will be connected to 8 bits of the 16-bit address/data bus of the MC9S12
    - If the address of the input is even, we need to connect the output of the buffer to the even (high) byte of the bus, which is connected to AD15-8 (what was Port A)
    - If the address of the input is odd, we need to connect the output of the buffer to the odd (low) byte of the bus, which is connected to AD7-0 (what was Port B)
  - The MC9S12 needs the data on the bus on the high-to-low transition of the E-clock
  - We must enable the tri-state buffer when
    1. The address of the buffer is on the address bus
    2. The MC9S12 is reading from this address
    3. The MC9S12 is reading the high byte if the address is even, or the low byte if the address is odd
    4. E is high
- For example, consider an input port at address 0x4000 (an even address, or high byte):



Example: Read from address 0x4000



Latch address using a transparent latch. When E is low, transfer inputs of latch to outputs. When E is high, outputs don't change even if inputs do.

Verilog code for address latch

```
output reg [15:0] address;
always @(e or port_a or port_b)
    if (e == 1'b0) address = {port_a, port_b};
```

Verilog code for chip select

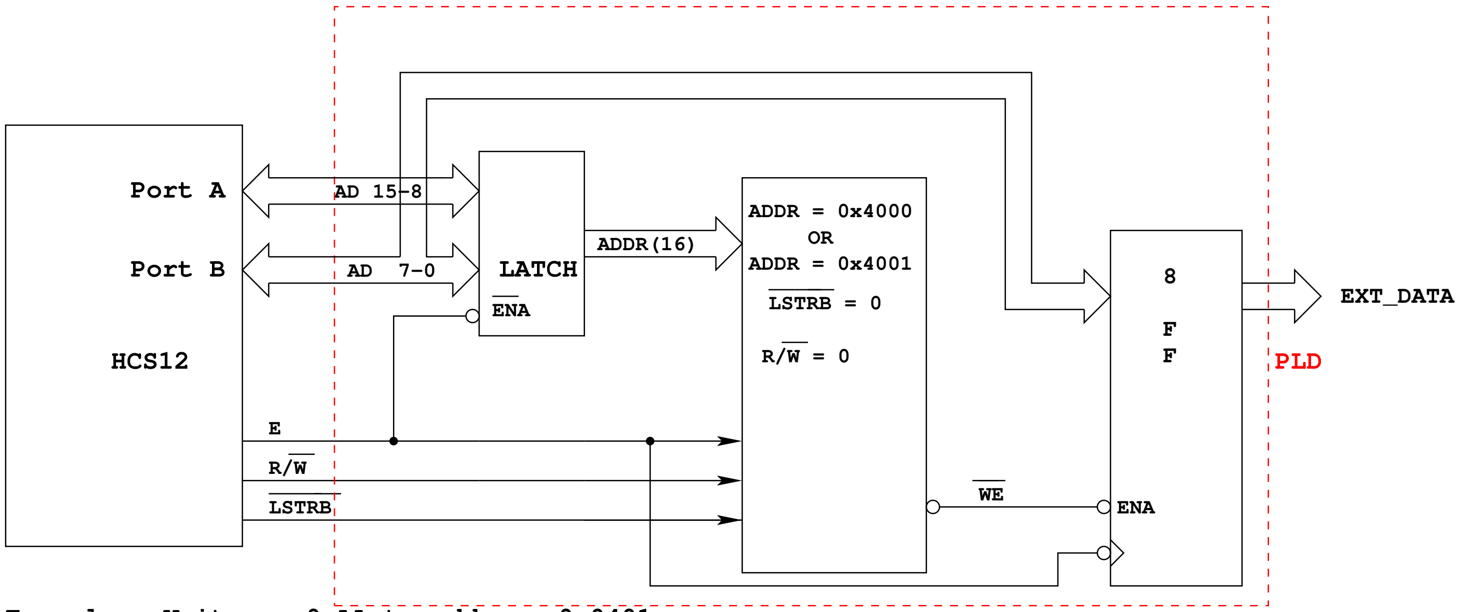
```
assign cs_r = ((address == 16h'4000) &&
             (rw == 1'b1) && (e == 1'b1)) ? 1'b0 : 1'b1;
```

Verilog code for Port A

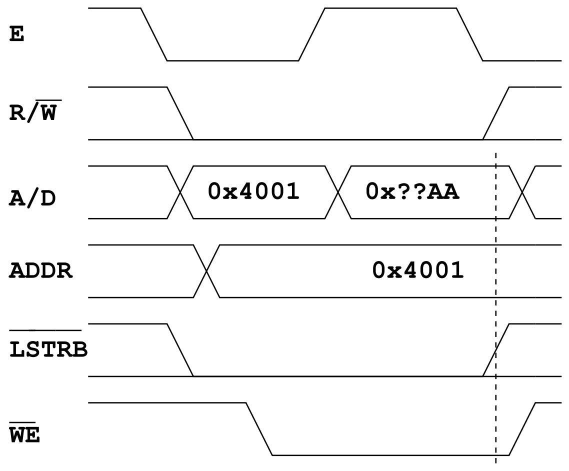
```
inout [7:0] port_a;
input [7:0] ext_data;
assign port_a = (cs_r == 1'b0) ?  ext_data : 8'hz;
```

A Simple Parallel Output Port

- We want a port which will write 8 bits of data to the outside
- Such a port is similar to Port A or Port B when all pins are set up as output
- We need some hardware to latch the output data at the time the MC9S12 puts the data on the data bus
- We can use a set of 8 D flip-flops to latch the data
  - The D inputs will be connected to the data bus
  - The clock to latch the flip-flops should make its low-to-high transition when the MC9S12 has the appropriate data on the bus
  - The MC9S12 will access the flip-flops by writing to an address. We must assign an address for the tri-state buffer
  - We must have hardware to demultiplex the address from the data, and to determine when the MC9S12 is writing to this address
  - The 8-bit inputs of the D flip-flops will be connected to 8 bits of the 16-bit address/data bus of the MC9S12
    - If the address of the input is even, we need to connect the flip flop inputs to the even (high) byte of the bus, which is connected to AD15-8 (what was Port A)
    - If the address of the input is odd, we need to connect the flip flop inputs to the odd (low) byte of the bus, which is connected to AD7-0 (what was Port B)
  - The hardware should latch the data on the high-to-low transition of the E-clock
  - Our hardware should bring the clock of the flip-flops low when
    1. The address of the flip-flops is on the address bus
    2. The MC9S12 is writing to this address
    3. The MC9S12 is writing the high byte if the address is even, or the low byte if the address is odd
    4. E is high
- For example, consider an output port at address 0x4001 (an odd address, or low byte):



Example: Write an 0xAA to address 0x0401



Verilog code for we

```
assign we = ((address[15:1] == 15'b0100_0000_0000_000) &&
                  (lstrb == 1'b0) && (rw == 1'b0)) ? 1'b0 : 1'b1
```

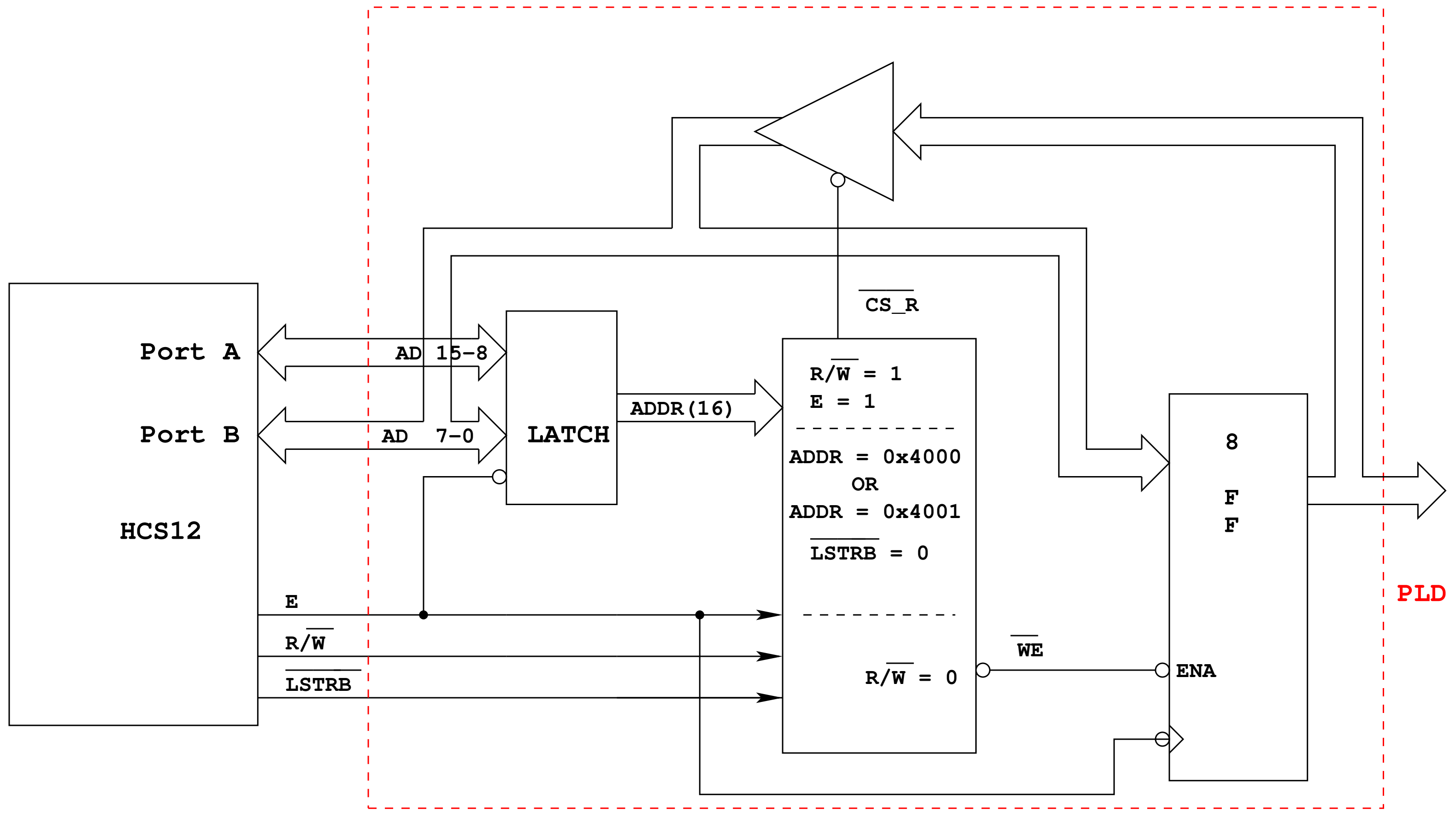
Verilog code for ext_data

```
output reg [7:0] ext_data;
always @(negedge e)
     if (we == 1'b0) ext_data <= port_b;
```

Note: ADDR can be 0x4000 or 0x4001 with LSTRB = 0

An Output Port Which Can Be Read

- Suppose we set ut the MC9S12 Port A for output, and we write a number to Port A
- When we read from Port A, we will read back the number we wrote
- This is a useful diagnostic
- We can make our output port have this same behaviour by connecting the output of the flip-flops back into the data bus through a tri-state buffer
- We should enable this tri-state buffer when the MC9S12 is reading from the address of the output port
- For example, consider the output port at address 0x4001:

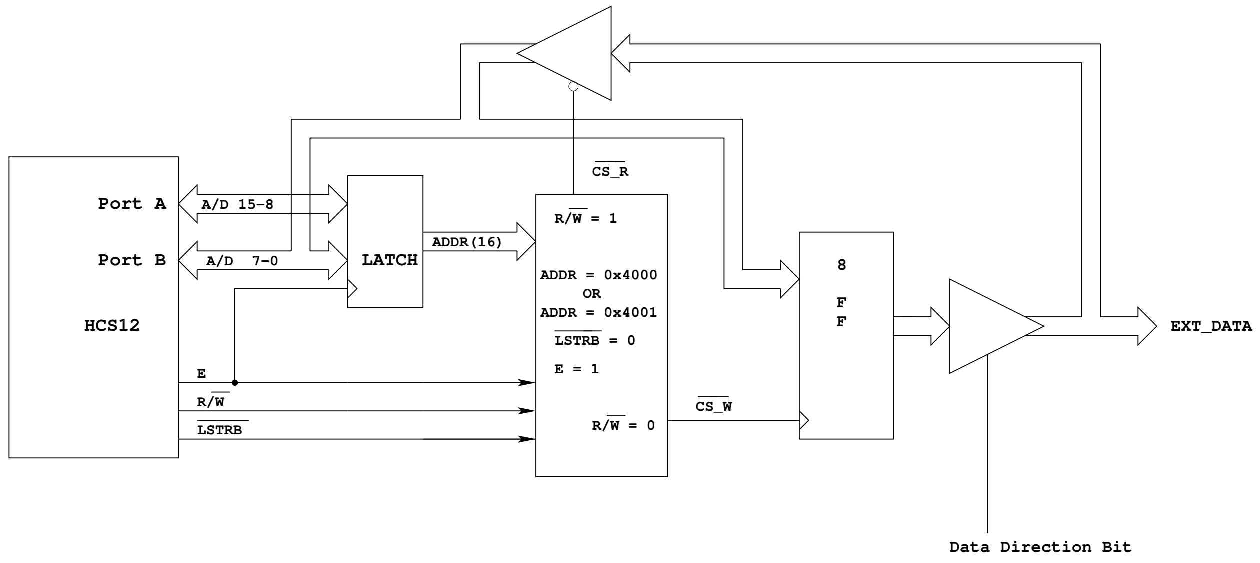


Writing to address 0x4001 (ADDR = 0x4000 or 0x4001, LSTRB low, R/W low) will bring WE low.
On the high-to-low transition of E with WE low, the data into the flip-flops

Reading from address 0x4001 (ADDR = 0x4000 or 0x4001, LSTRB low, R/W high, E high) will bring CS_R low
This will drive the data from the flip-flops onto the data bus
The HC12 will read the data on the flip-flops on the high-to-low transition of the E-clock

An Input-Output Port

- Like Port A, we can make a port be either input or output
- For simplicity, we will make all bits inputs or all bits outputs rather than allowing any individual bit to be either an input or an output
- To do this we need a data direction bit (at another address), and a tri-state buffer on the outputs of the flip-flops
- The data direction bit is simply a flip-flop which is set or cleared by the MC9S12
- When the data direction bit is cleared, the data from the output flip-flops will be removed from the external pins
  - When we read from the port, we will read the logic levels on the pins put there by external logic
- When the data direction bit is set, the data from the output flip-flops will be removed from the external pins
  - When we write to the port, we will drive the data from the flip-flops onto the external pins
  - For example consider an I/O port at address 0x4001. The direction of the port is determined by a data direction bit at address 0x4002:



The data direction bit is the output of a flip-flop which was written to at another address of the HC12
For example, it could be Bit 4 of address 0x4002

Writing a 0 to Bit 4 of address 0x4002 disables the output tri-state buffer
When we write to address 0x4001 we will latch the data into the flip-flops
This data will not be driven onto the external pins
When we read from address 0x4001, we will read what an external device drives onto the pins

Writing a 1 to Bit 4 of address 0x4002 enables the output tri-state buffer
When we write to address 0x4001 we will latch the data into the flip-flops
This data will be driven onto the external pins
When we read from address 0x4001 we will read the data latched into the flip-flops

Verilog code for ext_data

```
inout [7:0] ext_data;
inout [7:0] port_b;
reg [7:0] ext_latch;

always @(posedge cs_w) ext_latch <= port_b;
assign ext_data = (ddr == 1'b1)? ext_latch : 8'hz;
assign port_b = (cs_r == 1'b0)? ext_data : 8'hz;
```