CPE 462
VHDL: Simulation and Synthesis

Topic #04 - a) Basic synthesizable data-types

WESTERN NEW ENGLAND UNIVERSITY WNE

BIT (and BIT_VECTOR)

- There are two logic levels: ('0', '1')
- Defined in Package standard of library std (no need to declare it)

Declaration examples

```
SIGNAL x: BIT;
-- x is declared as a one-digit signal
-- of type BIT.

SIGNAL y: BIT_VECTOR (3 DOWNTO 0);
-- y is a 4-bit vector,
-- with the leftmost bit being the MSB.

SIGNAL w: BIT_VECTOR (0 TO 7);
-- w is an 8-bit vector,
-- with the rightmost bit being the MSB.
```

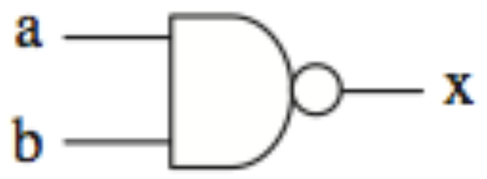
Assignment examples

```
x <= '1';
-- x is a single-bit signal,
-- whose value is '1'. Notice that single
-- quotes (' ') are used for a single bit.

y <= "0111";
-- y is a 4-bit signal (as specified above),
-- whose value is "0111"
-- (MSB='0'). Notice that double quotes (" ")
-- are used for vectors.

w <= "01110001";
-- w is an 8-bit signal, whose value is
-- "01110001" (MSB='1').
```

BIT usage



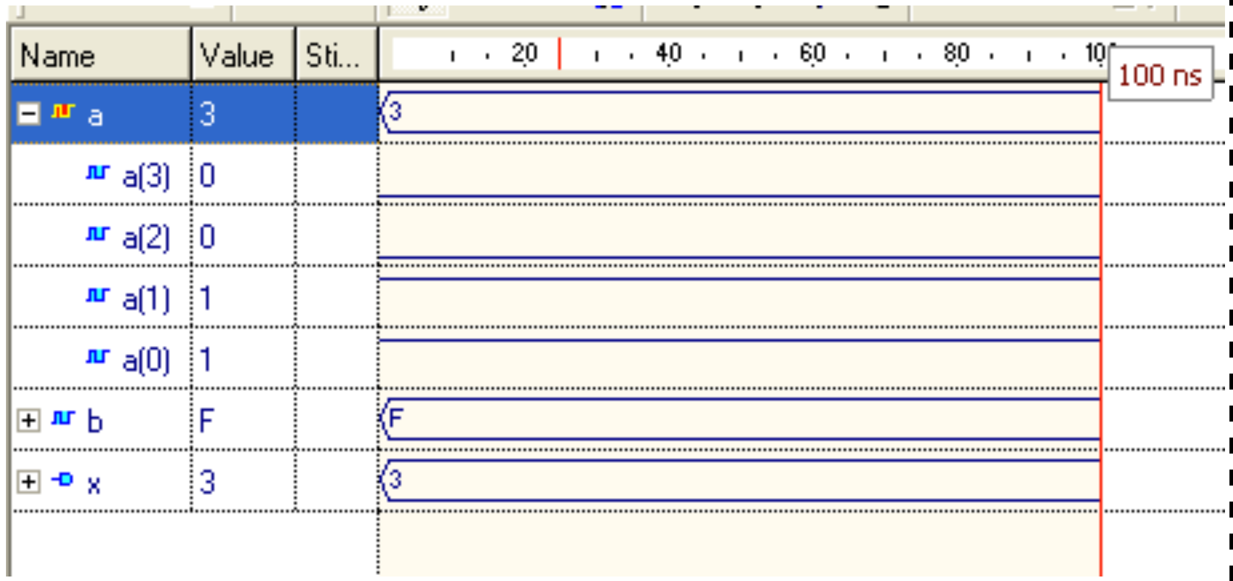
```
ENTITY nand_gate IS
   PORT (a, b : IN BIT;
         x : OUT BIT);
END nand_gate;

ARCHITECTURE myarch OF nand_gate IS
BEGIN
   x <= a NAND b;
END myarch;
```

- We've seen this before
- We take two input bits (0/1) and we AND them together

BIT_VECTOR usage

- This entity+architecture has no inputs and a single output
- I declare two internal signals a&b. Think of signals as wires...
- I can manually specify the values for these signals **ONLY** because they are not inputs to my entity



Name	Value
a	3
a(3)	0
a(2)	0
a(1)	1
a(0)	1
b	F
x	3

```
entity main_block is
   port (x : out bit_vector (3 downto 0));
end main_block;

architecture myarch of main_block is
signal a : bit_vector(3 downto 0);
signal b : bit_vector(3 downto 0);
begin
   a(0) <= '1';
   a(1) <= '1';
   a(2) <= '0';
   a(3) <= '0';
   -- this is the same as
   --a <= "0011";

   b <= "1111";
   x <= a AND b;
end myarch;
```

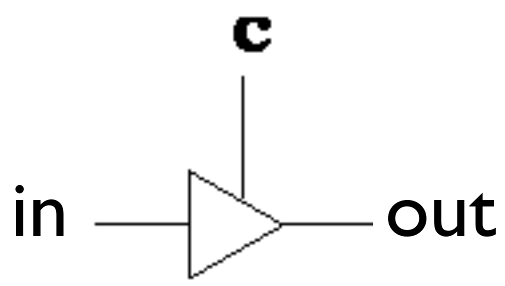
STD_LOGIC (and STD_LOGIC_VECTOR)

- STD_LOGIC (and STD_LOGIC_VECTOR): 8-valued logic system introduced in the IEEE 1164 standard
- There are several logic levels (not just '0' and 'I')
- Only some logic levels are synthesizable (can be deployed in hardware)
- The ones that are not synthesizable are only intended for simulation
- We will focus mainly on the synthesizable logic levels

Logic Level	Synthesizable?
X' Forcing Unknown	(synthesizable unknown)
0' Forcing Low	(synthesizable logic 'I')
I' Forcing High	(synthesizable logic '0')
Z' High impedance	(synthesizable tri-state buffer)
W' Weak unknown	(simulation only)
L' Weak low	(simulation only)
H' Weak high	(simulation only)
–' Don't care	(simulation only)

Start of minor detour

What is a Tri-state Buffer?

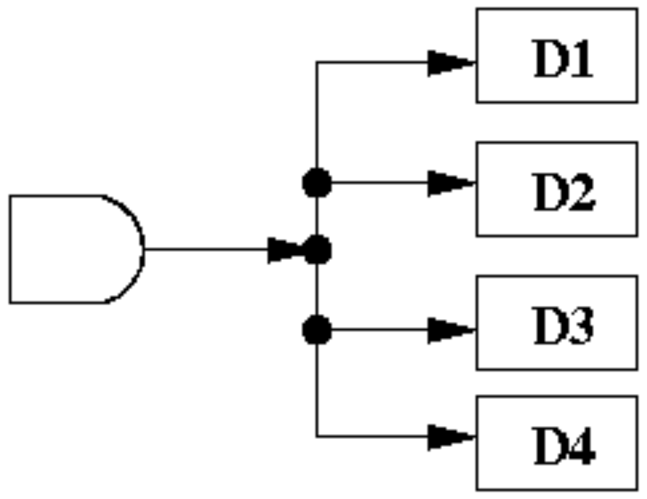


This is how a tri-sate buffer looks like

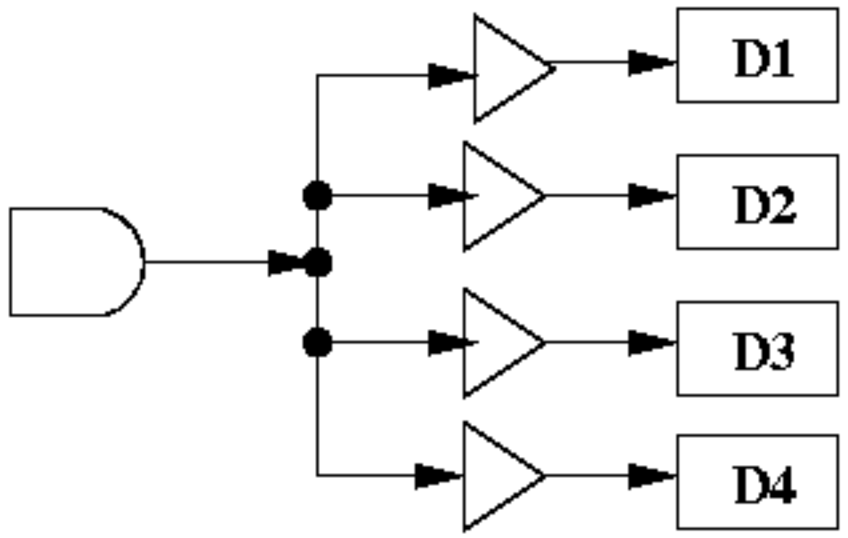
Minor digression... what is the purpose of buffers in real-designs?

- To artificially create some delay
- To increase current that a gate is feeding to other devices

Purpose of adding buffers



- This AND gate has a fanout of 4
- If each of the four devices gets equal current, then each device gets 1/4 of the initial current



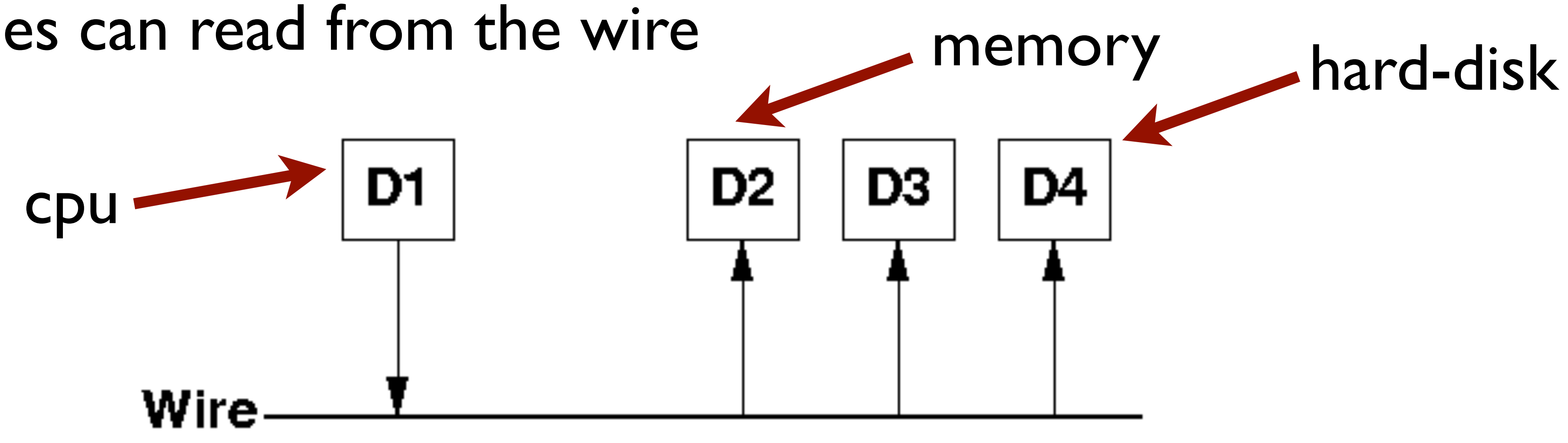
- By adding a buffer, I am boosting current to each device

Important: Why should I care about ensuring each gate gets a decent amount of current? You can't really accurately measure voltage levels if there is no current!

What is High Impedance (Z)?

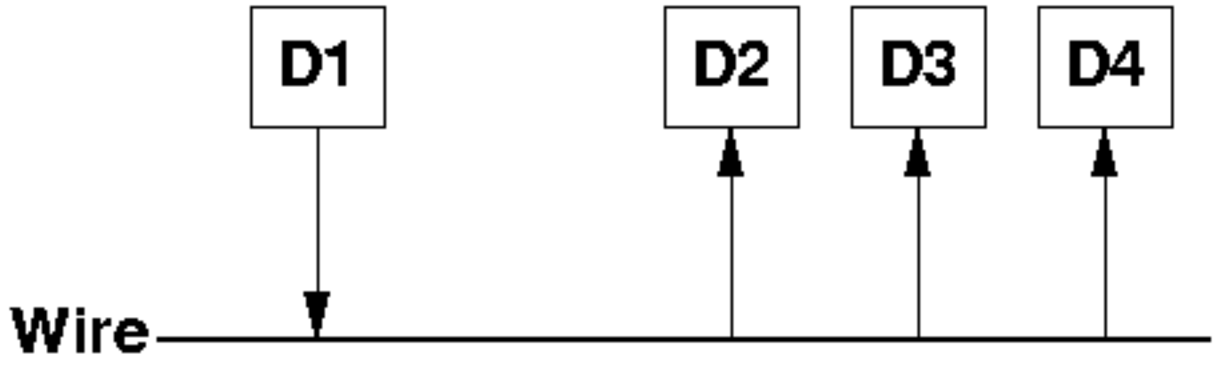
We can define a wire as follows:

- A piece of conductive material that allows electron flow
- A wire allows a 1-bit signal to be sent on it
- At most one device can write to a wire
- A device can write either a ‘0’ or ‘1’ on the wire
- Devices can read from the wire



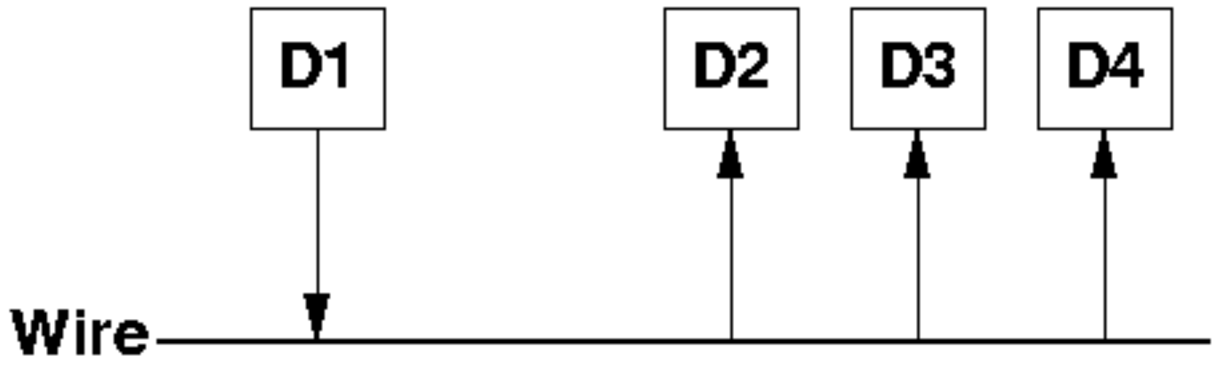
Wire may connect multiple devices

- When a device writes a ‘1’ or ‘0’, in reality, it is asserting a voltage, such as 0 volts for ‘0’ and 5 volts for ‘1’
- If two devices attempt to write a ‘0’ and ‘1’, then the wire is assumed to have a garbage value
- A device attempting to read from the wire, in such a situation, may read ‘0's’ sometimes and read ‘1's’ at other times
- We want to avoid two devices writing at the same time
- More than one device can read a value from a wire



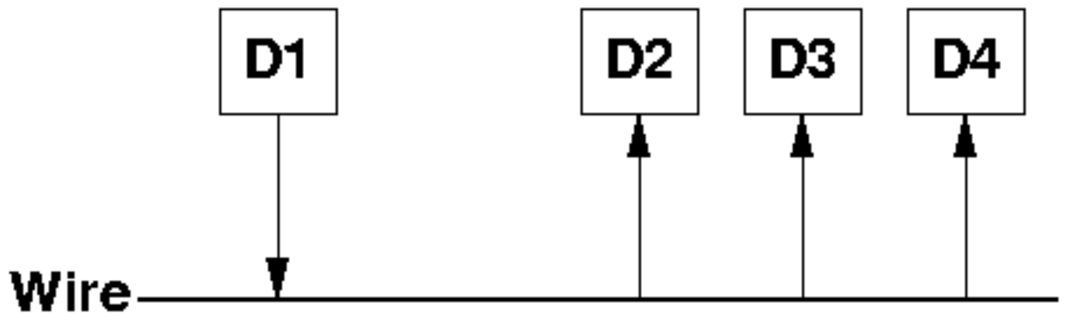
High impedance state

- If no devices write to the wire, then the wire has value Z, which stands for high impedance
- High impedance means that it is neither 0 nor 1
- If no device is writing to a wire, then reading from a wire gets an unknown value (either 0 or 1, but nothing predictable)
- A wire has no memory. That is, if you write a 1 to the wire, the wire does not store the value. The device must continuously assert a 1



Wire truth-table

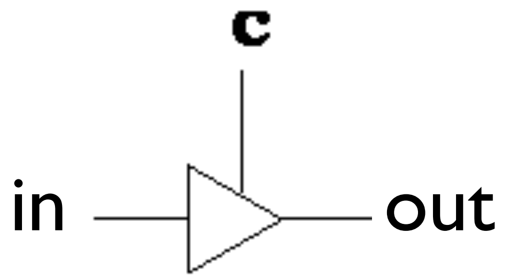
Device #1	Device #2	Output
Write 0	Write 0	0
Write 0	Write 1	Garbage
Write 1	Write 0	Garbage
Write 1	Write 1	1
Write nothing (Z)	Write 0	0
Write nothing (Z)	Write 1	1
Write nothing (Z)	Write nothing (Z)	Nothing (Z)



Garbage as in... sometimes we read a '1' other times we read a '0'.

Tri-state Buffer is a control valve

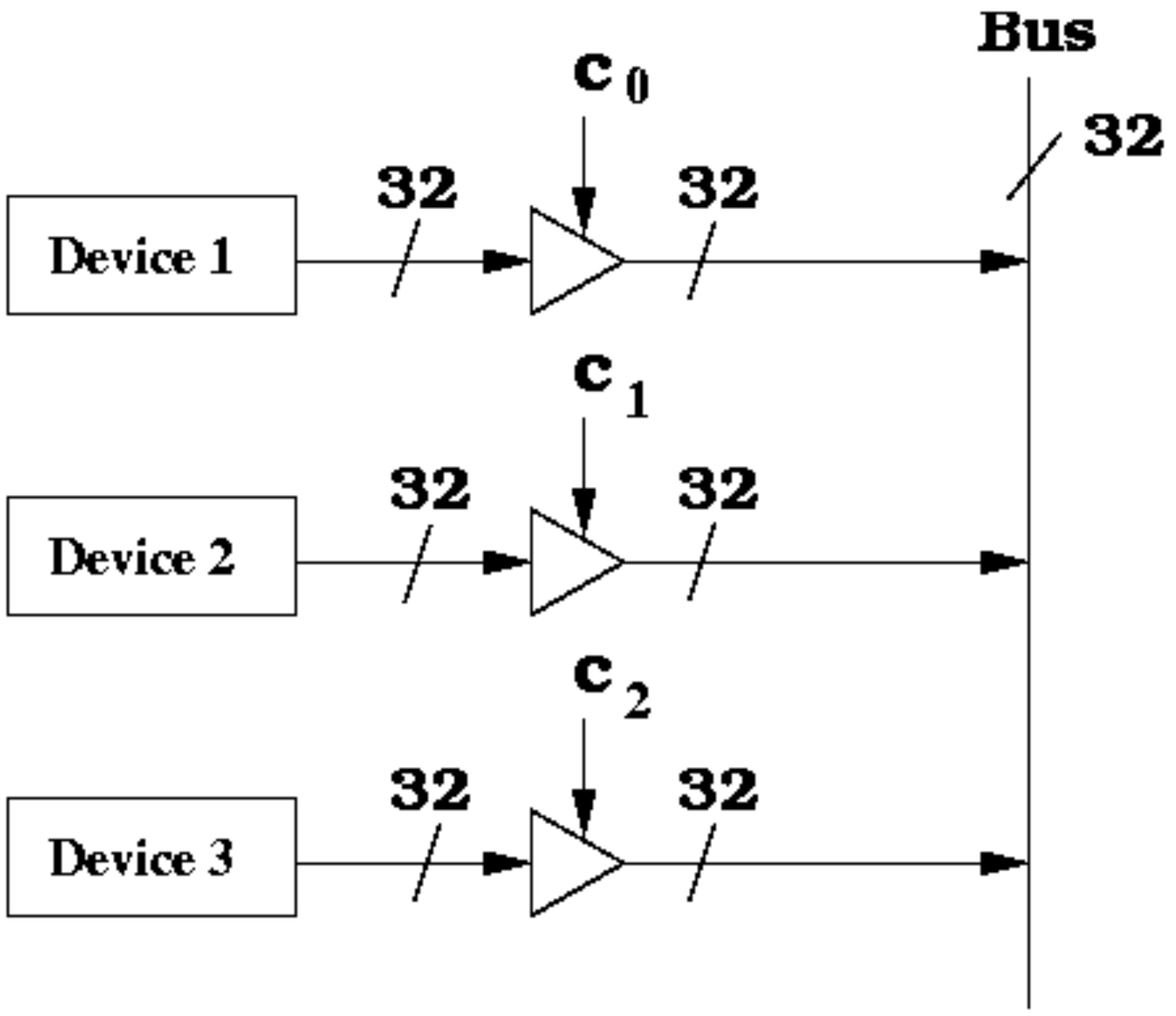
- When the control input is not active, the output is "Z"
- The "valve" is open, and no electrical current flows through
- Thus, even if x is 0 or 1, that value does not flow through



c	in	out
0	0	Z
0	1	Z
1	0	0
1	1	1

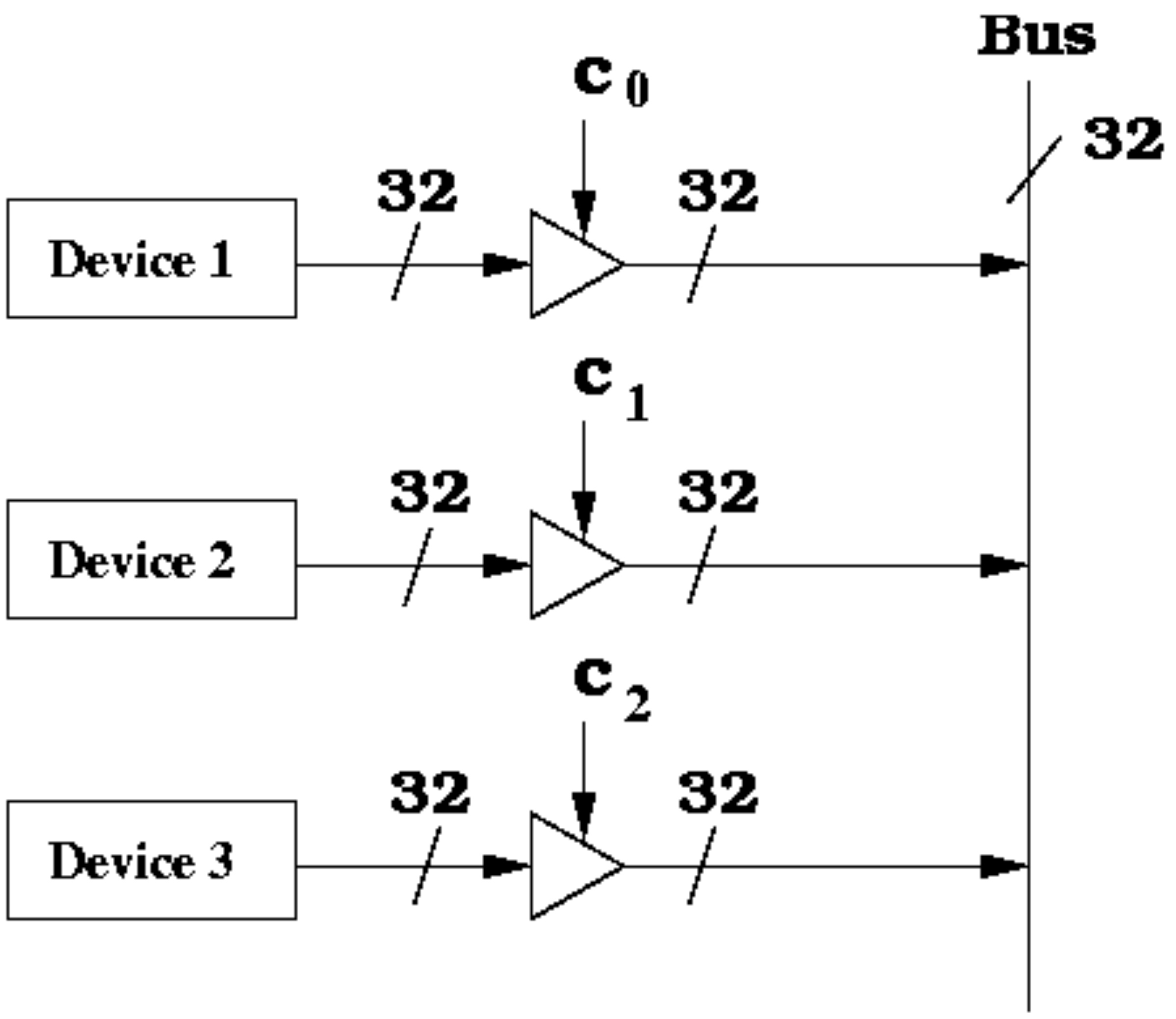
Why tri-state buffers?

- A common way for many devices to communicate with one another is on a bus
- That a bus should only have one device writing to it, although it can have many devices reading from it
- Since many devices always produce output (such as registers) and these devices are hooked to a bus, we need a way to control what gets on the bus, and what doesn't.



Tri-state buffers vs MUX

- Who cares? Why don't I replace these three state buffers with a MUX?
- With a MUX we're guaranteed only one device makes it to the bus.
- What if we don't want any devices to make it to the bus?
- One solution is to add an enable input to a MUX. Only when the enable is active, the output is selected from one of the inputs.



End of minor detour

STD_LOGIC_VECTOR usage

In order to use STD_LOGIC_VECTOR or STD_LOGIC you **MUST** add this library

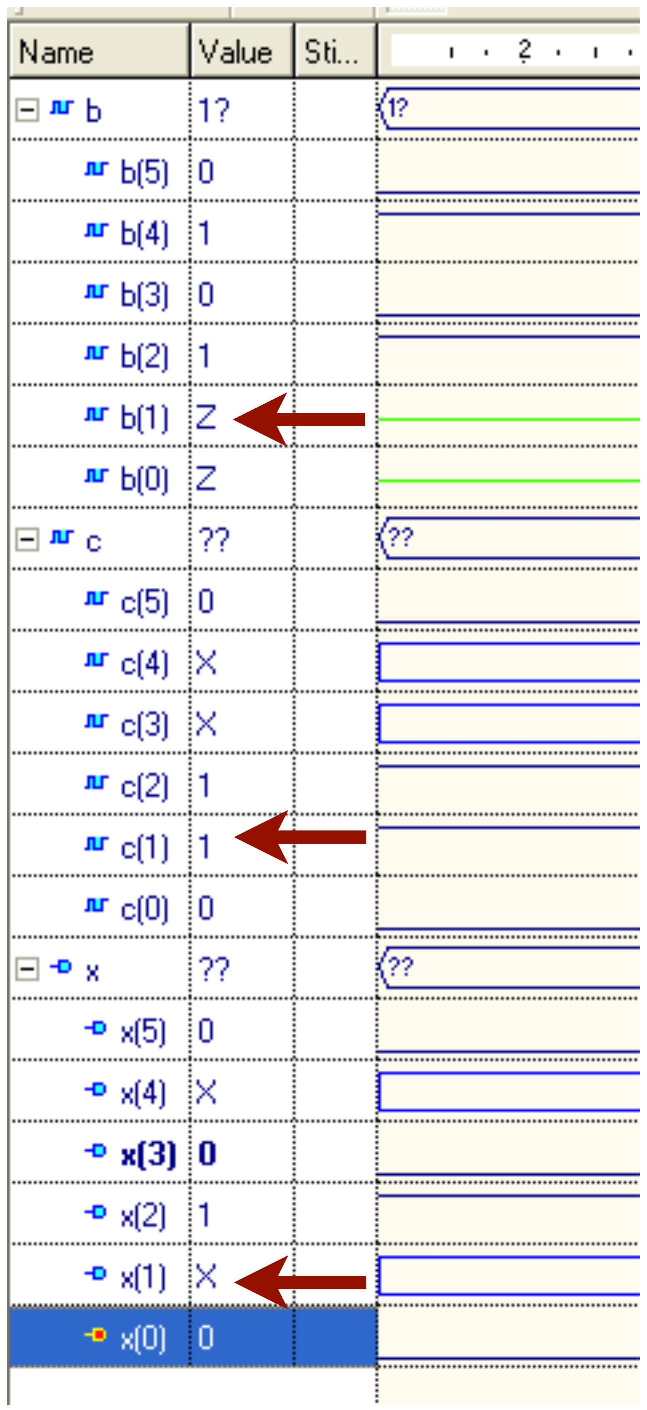
Since b and c are internal wires I force their initial values this way

```
library ieee;
use ieee.std_logic_1164.all;

entity main_block is
   port (x : out std_logic_vector(5 downto 0));
end main_block;

architecture myarch of main_block is
signal b : std_logic_vector(5 downto 0):= "0101ZZ";
signal c : std_logic_vector(5 downto 0):= "0XX110";
begin
   x <= b AND c;
end myarch;
```

STD_LOGIC_VECTOR



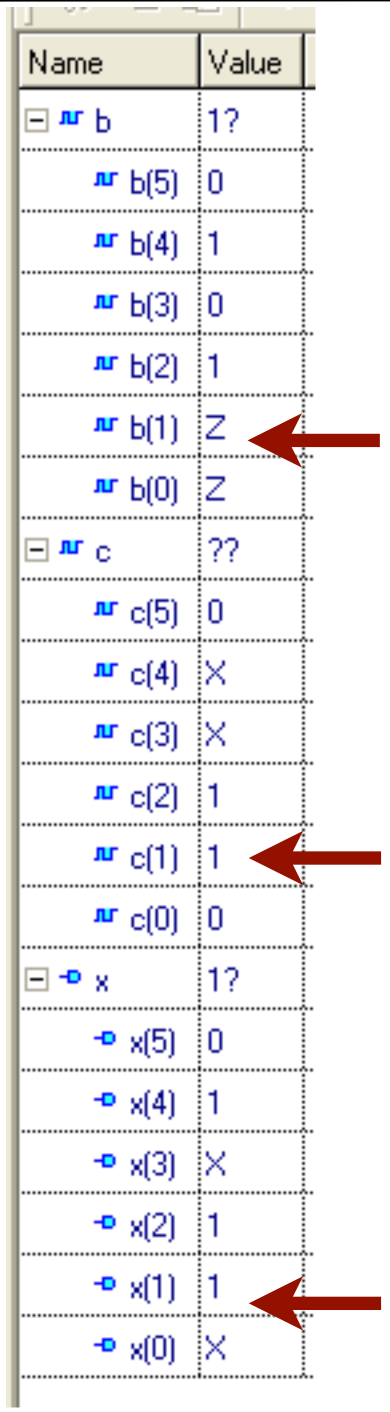
Name	Value	Sti...
b	1?	
b(5)	0	
b(4)	1	
b(3)	0	
b(2)	1	
b(1)	Z	
b(0)	Z	
c	??	
c(5)	0	
c(4)	X	
c(3)	X	
c(2)	1	
c(1)	1	
c(0)	0	
x	??	
x(5)	0	
x(4)	X	
x(3)	**0**	
x(2)	1	
x(1)	X	
x(0)	0	

```
library ieee;
use ieee.std_logic_1164.all;

entity main_block is
   port (x : out std_logic_vector(5 downto 0));
end main_block;

architecture myarch of main_block is
signal b : std_logic_vector(5 downto 0):= "0101ZZ";
signal c : std_logic_vector(5 downto 0):= "0XX110";
begin
   x <= b AND c;
end myarch;
```

STD_LOGIC_VECTOR



Name	Value
b	1?
b(5)	0
b(4)	1
b(3)	0
b(2)	1
b(1)	Z
b(0)	Z
c	??
c(5)	0
c(4)	X
c(3)	X
c(2)	1
c(1)	1
c(0)	0
x	1?
x(5)	0
x(4)	1
x(3)	X
x(2)	1
x(1)	1
x(0)	X

```
library ieee;
use ieee.std_logic_1164.all;

entity main_block is
   port (x : out std_logic_vector(5 downto 0));
end main_block;

architecture myarch of main_block is
signal b : std_logic_vector(5 downto 0):= "0101ZZ";
signal c : std_logic_vector(5 downto 0):= "0XX110";
begin
   x <= b OR c;
end myarch;
```

Boolean logic with unknowns (X)

X can be 0 or 1

If A=0 and B=X
0 AND 0 = 0
0 AND 1 = 0

0 OR 0 = 0
0 OR 1 = 1

If A=1 and B=X
1 AND 0 = 0
1 AND 1 = 1

1 OR 0 = 1
1 OR 1 = 1

A	B	A and B	A or B	A xor B
0	X	0	X	X
1	X	X	1	X

If A=0 and B=X
A AND B is always 0, because the value of B can never alter the result

If A=1 and B=X
A OR B is always 1, because the value of B can never alter the result

Boolean logic with high impedance (Z)

Z can be 0 or 1 or neither

If A=0 and B=Z
0 AND 0 = 0
0 AND 1 = 0
0 AND neither = 0

0 OR 0 = 0
0 OR 1 = 1
0 OR neither = 0

A	B	A and B	A or B	A xor B
0	Z	0	X	X
1	Z			

Boolean logic with high impedance (Z)

Z can be 0 or 1 or neither

If A=1 and B=Z
1 AND 0 = 0
1 AND 1 = 1
1 AND neither = 1

1 OR 0 = 1
1 OR 1 = 1
1 OR neither = 1

A	B	A and B	A or B	A xor B
0	Z	0	X	X
1	Z	X	1	X

Other synthesizable VHDL data types

- BOOLEAN: True, False
- INTEGER: 32-bit integers (from -2,147,483,647 to +2,147,483,647)
- NATURAL: Non-negative integers (from 0 to +2,147,483,647)
- SIGNED and UNSIGNED: data types defined in the std_logic_arith package of the ieee library. They have the appearance of STD_LOGIC_VECTOR, but accept arithmetic operations, which are typical of INTEGER data types

Assignment examples

`x0 <= '0';`	`bit, std_logic, or std_ulogic value '0'`
`x1 <= "00011111";`	`bit_vector, std_logic_vector, std_ulogic_vector, signed, or unsigned`
`x4 <= B"101111"`	`binary representation of decimal 47`
`x5 <= O"57"`	`octal representation of decimal 47`
`x6 <= X"2F"`	`hexadecimal representation of decimal 47`
`n <= 1200;`	`integer`
`m <= 1_200;`	`integer, underscore allowed`
`IF ready THEN...`	`Boolean, executed if ready=TRUE`

Full code examples

- Useless entity, but exemplifies how we can declare internal signals with some initial values.

```
library ieee;
use ieee.std_logic_1164.all;

entity main_block is
end main_block;

architecture myarch of main_block is
   signal x0 : std_logic :='0';
   signal x5 : std_logic_vector(5 downto 0):= O"57";
   signal x6 : std_logic_vector(7 downto 0):= X"2F";
   signal n : integer := 1200;
   signal m : integer := 1_200;
   signal ready : boolean := true;
begin

end myarch;
```

Full code examples

- Same outcome as the previous slide, but signals are initialized inside the architecture begin/end block.

```
library ieee;
use ieee.std_logic_1164.all;

entity main_block is
end main_block;

architecture myarch of main_block is
   signal x0 : std_logic;
   signal x5 : std_logic_vector(5 downto 0);
   signal x6 : std_logic_vector(7 downto 0);
   signal n : integer;
   signal m : integer;
   signal ready : boolean;
begin
   x0 <= '0';
   x5 <= O"57";
   x6 <= X"2F";
   n  <= 1200;
   m  <= 1_200;
   ready <= true;
end myarch;
```

Legal or Illegal?

A good way to confirm these, is to try to compile this code in active HDL.

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5);      -- legal (same scalar type: BIT)
b(0) <= a;      -- legal (same scalar type: BIT)
c <= d(5);      -- legal (same scalar type: STD_LOGIC)
d(0) <= c;      -- legal (same scalar type: STD_LOGIC)
a <= c;         -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d;         -- illegal (type mismatch: BIT_VECTOR x
                -- STD_LOGIC_VECTOR)
e <= b;         -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d;         -- illegal (type mismatch: INTEGER x
                -- STD_LOGIC_VECTOR)
```

Practice Exercises

Exercise #1- Legal or Illegal Assignments?

& means append!

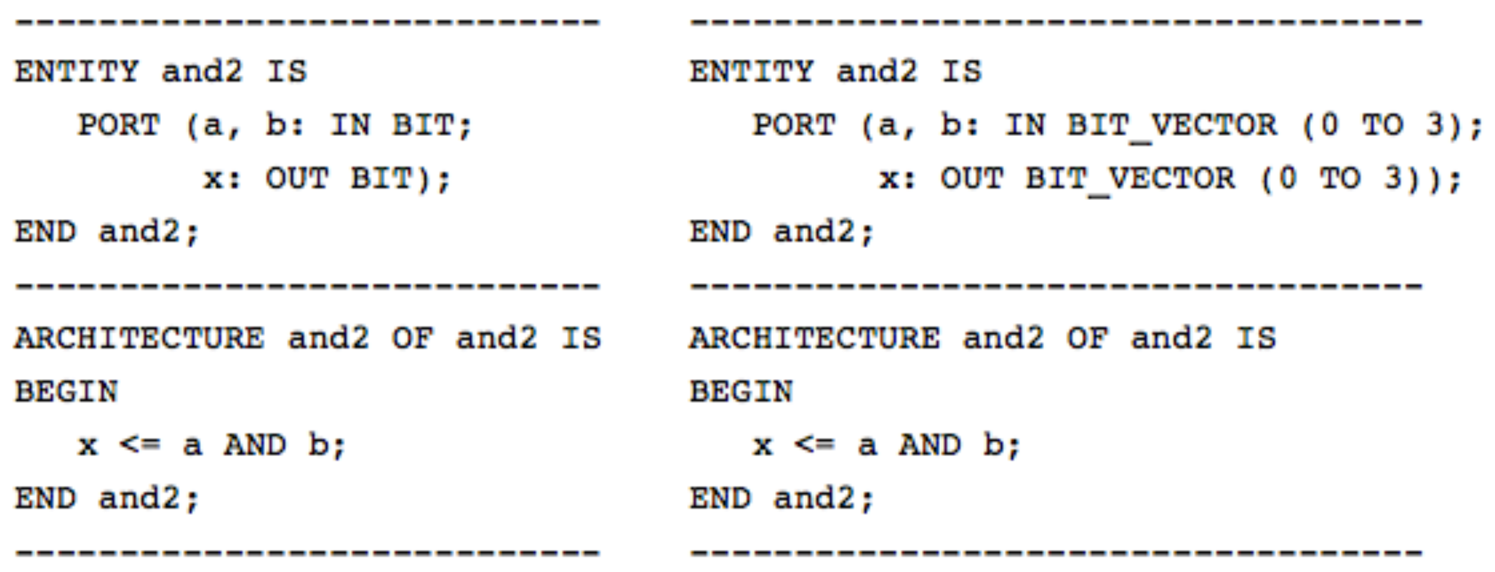
1) Look at these signals... Which are the legal assignments? Why?

```
SIGNAL a: STD_LOGIC;
SIGNAL b: BIT;
SIGNAL x: byte;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL v: BIT_VECTOR (3 DOWNTO 0);
SIGNAL z: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Not sure about some of these? Use Active HDL to check.

`z <= "11111" & "000";`
`x(2) <= a;`
`b <= a;`
`y(5 TO 7) <= z(6 DOWNTO 0);`
`y(0) <= x(0);`
`y <= ('1','1','1','1','1','1','0','Z');`
`x <= "11111110";`
`z <= y;`
`b <= v(3);`
`y(2 DOWNTO 0) <= z(6 DOWNTO 4);`
`x <= y;`
`z(7) <= x(5);`

Exercise #2- What is the difference between these two implementations



```
------------------------------
ENTITY and2 IS
   PORT (a, b: IN BIT;
         x: OUT BIT);
END and2;
------------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
   x <= a AND b;
END and2;
------------------------------
```

```
-----------------------------------
ENTITY and2 IS
   PORT (a, b: IN BIT_VECTOR (0 TO 3);
         x: OUT BIT_VECTOR (0 TO 3));
END and2;
-----------------------------------
ARCHITECTURE and2 OF and2 IS
BEGIN
   x <= a AND b;
END and2;
-----------------------------------
```

Draw the inferred circuit from each code snippet.