Digital System Design

by
Dr. Lesley Shannon
Email: lshannon@ensc.sfu.ca
Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc350

Simon Fraser University

Slide Set: 1
Date: January 12, 2009

Slide Set Overview

- Review of Basic Combinational logic blocks
- Review of VHDL for combinational components
  - Constructs in VHDL that help you describe combinational circuits
    - Note: Arithmetic circuits are also combinational, but we'll talk about those later

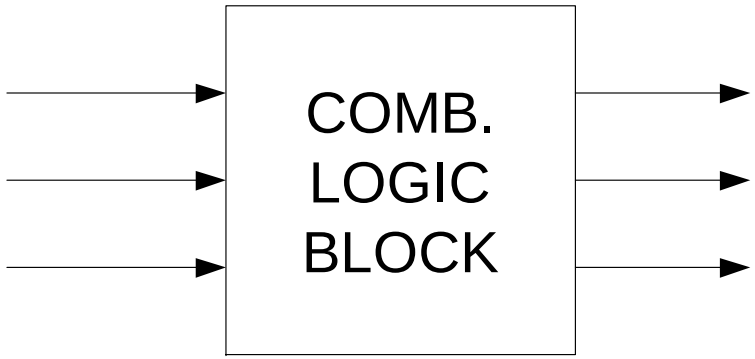
Slide Set Overview

- Initial Review of Sequential Logic
  - We'll go over the logic constructs in the next slide set

Combinational Logic

Combinational Logic

- A combinational logic block is one where the outputs depend only on the current inputs



- A combinational logic block can be implemented using simple gates, look-up tables or other techniques

Ways to describe combinational logic f(x)

1. English:
   - The function has an output of 1 when an even number of inputs are 1 (This can be ambiguous)
2. A Truth Table:

a	b	c	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Ways to describe combinational logic f(x)

3. A Boolean Equation:

f = a'b'c' + a'bc + …

We can use a Karnaugh map to minimize an equation (i.e. produce an equivalent equation with fewer literals)

The simplified equation is:

f =

What type of gate does this equation represent?

Ways to describe combinational logic f(x)

4. VHDL or Verilog
    4. We'll talk about this soon
5. A schematic diagram

5. A schematic diagram of function f

Basic Combinational Building Blocks

- Multiplexers
  - Choose one of two inputs based on a *select* signal

(a) Graphical symbol (b) Truth Table

*Note: You can make a multiplexer out of logic gates – How would you start?

Basic Combinational Building Blocks

- Multiplexers
  - Can make bigger multiplexers, e.g. 4-inputs

(a) Graphical symbol (b) Truth Table

  - How would you build a 4-input mux using 2-input muxes?

Basic Combinational Building Blocks

- Decoders
  - n inputs generate 2^n outputs
  - The decoder asserts ONE of the of the 2^n output signals
    - The asserted signal depends on the input value
    - The other outputs are all ‘0’

Graphical symbol

  - If enable is low, then all of the output lines are ‘0’

Basic Combinational Building Blocks

- Truth table for a 2-bit decoder

Equivalent Logic Circuit:

Give an example of where we might use a decoder.

Basic Combinational Building Blocks

- Encoders
  - Performs the opposite operation of a decoder
  - When one of the 2^n input signals is asserted, the appropriate binary encoded output is generated using n outputs (indicates which of the input signals is high)

Graphical symbol

Basic Combinational Building Blocks

- ***Priority*** Encoders
  - A special case where if more than one input is ‘1’, the index of the “highest priority” input is returned:
    - Normally multiple “1”s are assumed to never happen and are thus “don’t care” patterns

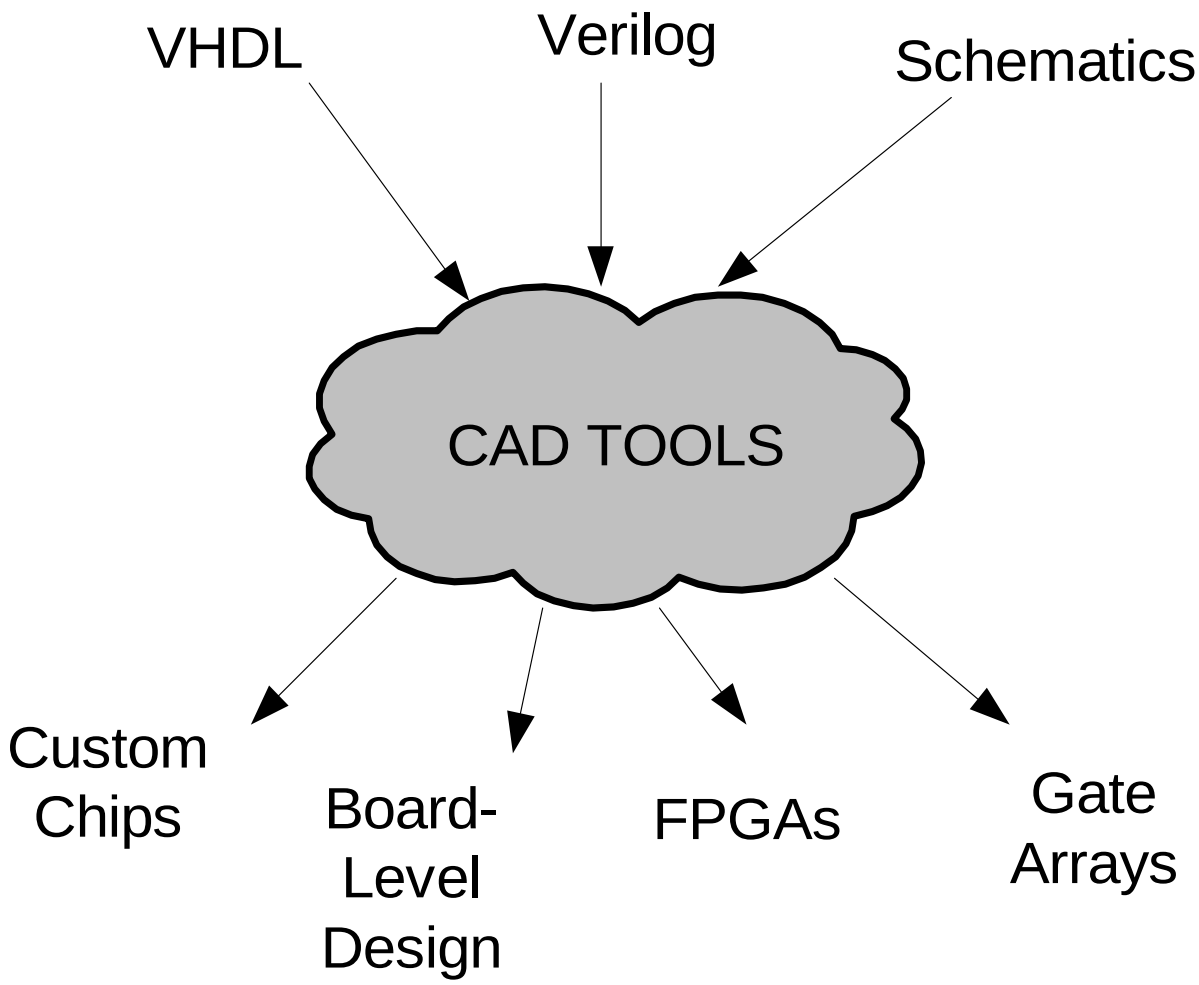
Truth Table

Hardware Design Languages

What is VHDL?

- It is ***NOT*** a programming language!
- It is an IEEE standard method of describing hardware
- VHDL = VHSIC Hardware Design Language
  - VHSIC = Very High Speed Integrated Circuits
  - VLSI = Very Large Scale Integration
    - LSI = Large Scale Integration 100 to 1000 devices on a chip
    - SSI = Small Scale Integration up to 10 devices on a chip

The relationship between HDLs and Chips



Why use a HDL to describe hardware?

- What is wrong with using schematics?
- With HDLs, you can specify hardware in two ways:
  - Structurally: what the hardware looks (like schematics)
  - Behaviourally: what the hardware does
- HDLs:
  - Let you combine structural and behavioural descriptions in the same design (very powerful)
  - Provides a higher level of abstraction (facilitates the representation of complex circuits)

Entirely behavioural description of a processor

- You could write a single piece of VHDL code that describes exactly what the processor will do
  - Why you might do this:
    - Eliminates misunderstandings among designers as to what the processor will do
    - You can simulate the design to make sure there is nothing that you haven't thought of (e.g. did you forget the carry bit)
    - You can send off another team of software engineers to create a compiler (they will have a complete functional model of the design to work with).

  Note: at this stage, you have ***NOT*** done the hardware design yet. You have specified what the hardware will do.

Behavioural/Structural description of a processor

- Behavioural Functional Units:
  - You might divide the design into major functional units, and then write behavioural code to specify what each block does. Then, you can write structural code to specify how the blocks are connected. You might do this to:
    - Allow you to farm out work to individual designers (each designer will know exactly what his/her block is to do)
    - Allow you to experiment with different architectural decisions (how wide should the bus be, should we have a separate floating point unit, etc.)
    - Allow you to swap in/out detailed designs for each module as they are completed.

  Note: at this stage, you have ***NOT*** done the hardware design yet. You have broken the design into major functional units and made major architectural decisions.

Register-Transfer-Level (RTL) Design

- For each functional unit, specify what the hardware looks like, in terms of basic building blocks (i.e. mux, combinational blocks, state machines, etc.)
  - Why you might do this:
    - It allows you to determine exactly what the hardware will look like. You can simulate the design to make sure it matches the behavioural code created earlier, or if you did not create the behavioural code, you can make sure it matches your understanding of the design.
    - You can simulate it with descriptions of the other functional units (either behavioural or RTL versions) to make sure there are no unintended interactions
    - You can ***accurately*** estimate how fast/big your chip will be
    - This description can serve as a source for synthesis (more on this later)

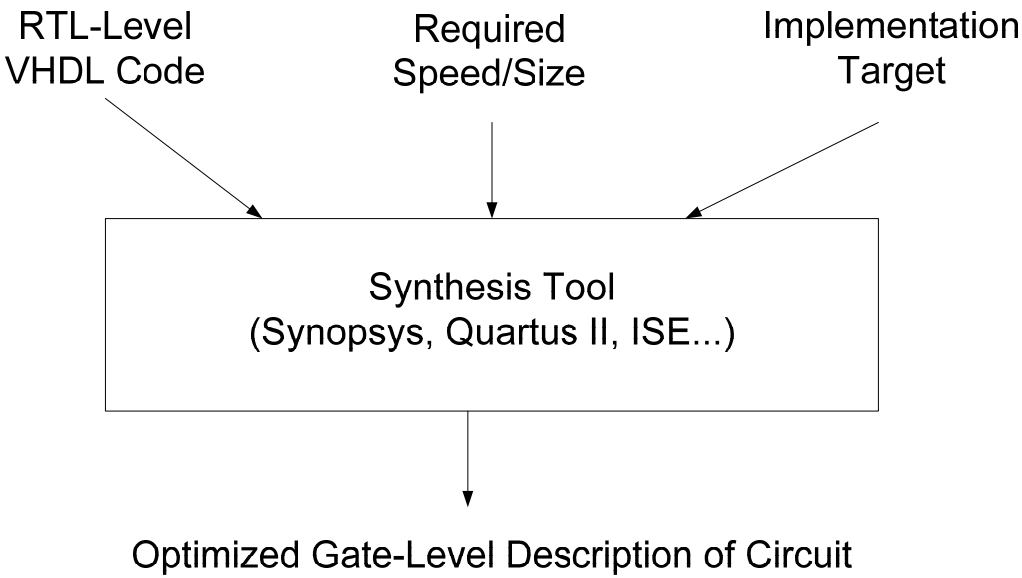
Gate-Level Design

- Can specify your design in terms of individual gates. Often this is created automatically from the RTL description (more on this later) but you might want to specify:
  - Exactly how certain blocks should be constructed (i.e. if you think you can do a better job at state assignment that the synthesis tool, if you think you can implement the circuit in some clever, tricky way that the synthesis tool might no figure out)
  - In most cases, humans are smarter than synthesis tools!
    - At least, today that is often the case
    - A few years from now, if researchers have their way, that might not be the case

So, the fact that you can specify hardware at all these different levels is why VHDL is becoming so popular

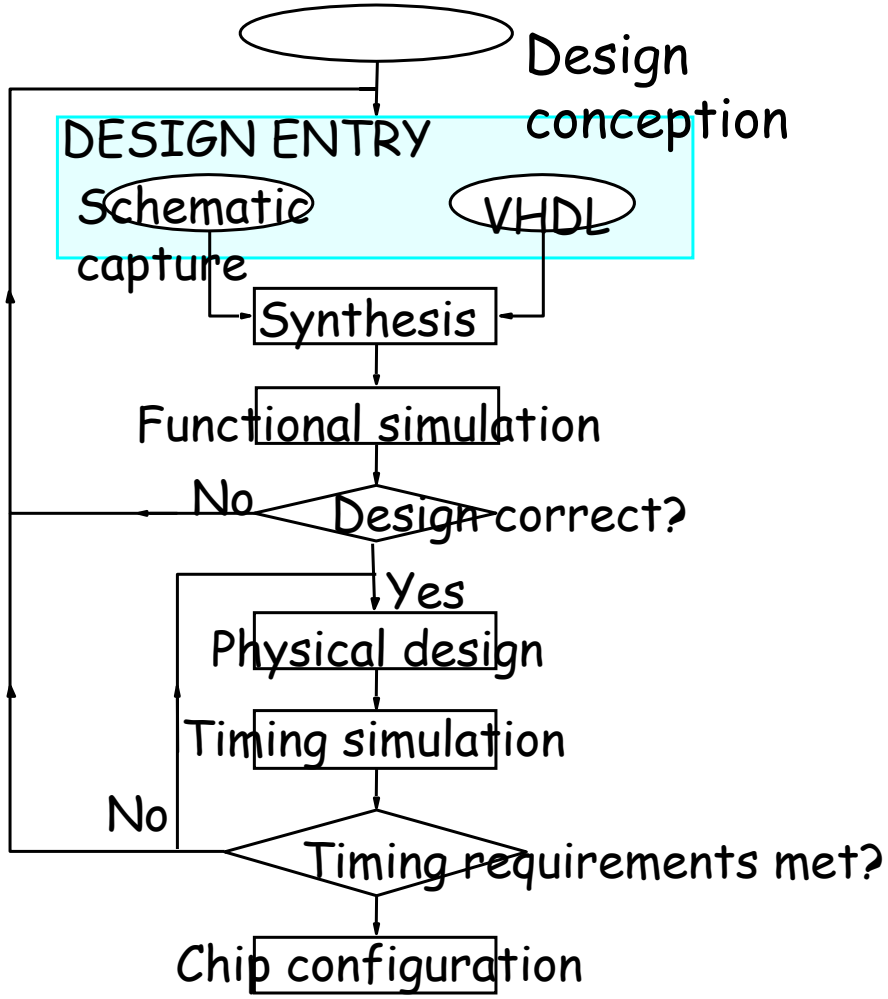
VHDL: A source for synthesis

- Today's definition of Synthesis:
  - Automatically creating an optimized gate-level description from an RTL-level description:



- Tomorrow's definition of Synthesis:
  - Automatically creating an optimized gate-level description from a behavioural description

Design Flow



B&V: Figure 2.29. A typical CAD system.

1. Describe the system
   - Flow chart, requirements
2. Think about the hardware you want to build
   - What are the building blocks?
3. ***Only then write VHDL!!***

Challenge: If you do step 3 without/before the other two, you are (very likely) going to end up with a poor quality circuit

What is Verilog?

- Verilog is another Hardware Description Language
  - Can be used to generate hardware circuits
- Verilog and VHDL are about equally common
  - Who will win? Remember VHS vs Beta?
  - Verilog in Europe and VHDL in North America
- Once you learn VHDL you will easily be able to pick up Verilog quickly.

Things to remember:

-VHDL was not meant as a hardware ***DESIGN*** language, but a hardware ***DESCRIPTION*** Language

-There are many things that we can describe, but cannot build

-Do not expect all VHDL code to be "synthesizable"

Combinational Logic in VHDL

- The basic construct for modelling a digital system in VHDL is called a ***design entity***. Each hardware block is described in one design entity
- A VHDL Design Entity consists of two parts:
  - An ***Interface Description***:
    - Describes the inputs and outputs of the block
    - In VHDL, denoted by the keyword entity
  - A ***Body***:
    - Describes either what the block does and/or what it is composed of
    - In VHDL, denoted by the keyword architecture

A VHDL Description of an XOR Gate

```
entity XOR_GATE is
      port ( A, B : in BIT;
                  Z : out BIT);
end XOR_GATE;


architecture MY_DEFN of XOR_GATE is
begin
      Z <= A xor B;
end MY_DEFN;
```

A VHDL Description of a more complex gate

```
entity BIG_GATE is
      port ( A, B, C : in BIT;
                  Z : out BIT);
end BIG_GATE;


architecture MY_DEFN of BIG_GATE is
begin
      Z <= (not A and B) or (B and not C);
end MY_DEFN;
```

Things to note about the entity part

- Ports are signals that flow into/out of the design
- The direction of flow is called the *mode* of the port
  - E.g. in, out, inout, linkage, buffer
- Type information (e.g. BIT) declares a set of legal values for the port. A signal of type BIT can be either a ‘1’ or ‘0’. Other types are possible
  - More on this later

Things to note about the architecture part

- Each entity might have several architectures
  - One behavioural, one structural for example
- Therefore, must name each architecture
  - E.g. MY_DEFN
- Common names are:
  - Behavioural
  - Structural
  - Dataflow
  - Etc.

A VHDL Description of a block with several outputs

```
entity COMB_BLOCK is
    port (A, B, C, D  : in BIT;
          X, Y, Z     : out BIT);
end COMB_BLOCK;

architecture MY_DEFN of COMB_BLOCK is
begin
    X <= A and B and C;
    Y <= C and not D;
    Z <= A xor B xor D;
end MY_DEFN;
```

What does the circuit look like?
Note: The assignments are performed concurrently

A VHDL Description of a mystery block

```
entity MYSTERY is
        port ( IN1, IN2, IN3  : in BIT;
                    OUT1, OUT2 : out BIT);
end MYSTERY;

architecture MY_DEFN of MYSTERY is
begin
        OUT1 <= IN1 xor IN2 xor IN3;
        OUT2 <= (IN1 and IN2) or (IN1 and IN3) or (IN2
                    and IN3);
end MY_DEFN;
```

What is this Block?

An XOR gate using internal signals

```
entity XOR_GATE is
      port (A, B  : in BIT;
            Z     : out BIT);
end XOR_GATE;

architecture ALT_DEF of XOR_GATE is
       signal INT1, INT2 : BIT;
begin
        INT1 <= A and not B;
        INT2 <= not A and B;
        Z <= INT1 or INT2;
end ALT_DEF;
```

Two Internal Signals

-This is the same as the “xor” function

-We have defined two INTERNAL signals. Sometimes this results in simpler definitions (not in this case, though)

Bad assignment

What's the mistake in this example?

```
entity BAD_MODEL is
    port (A, B, C, D : in BIT;
          X, Y_BAR  : out BIT);
end BAD_MODEL;
architecture EXAMPLE of BAD_MODEL is
begin
    X <= (A and B) or C;
    Y_BAR <= X nand D;
end EXAMPLE;
```

- Ports can be ***in, out***, or ***inout***.
- Rules:
  - Signals that are ***in*** can not be written to (can not appear on the left side
  - Signals that are ***out*** cannot appear on the right side (cannot be read from)

Good assignment- internal signal solution

```
entity GOOD_MODEL is
    port (A, B, C, D : in BIT;
          X, Y_BAR  : out BIT);
end GOOD_MODEL;

architecture EXAMPLE of GOOD_MODEL is
signal INT_SIG : BIT;
begin
    INT_SIG <= (A and B) or C;
    X <= INT_SIG;
    Y_BAR <= INT_SIG nand D;
end EXAMPLE;
```

- Note: ports that are declared as inout do not have these restrictions

A recipe for specifying combinational logic in VHDL

1. Write a boolean expression for each output

2. Include each boolean expression as a signal assignment within an architecture description (as shown in an earlier example)

- This will work and you can follow it. However, there are more efficient and easier ways to do this.

Basic Things to Remember about VHDL

- VHDL is case – *insensitive*, and spaces and new lines can appear *anywhere* (except in the middle of a keyword)

- Comments can appear anywhere, using two dashes

  -- The rest of this line is a comment

- In your assignments/projects (and any VHDL code you might write in the “real” world), please use ***LOTS*** of comments!

Onto more Complex HDL

Structural Specifications

- One of the important concepts in VHDL is hierarchy
  - That is specifying a circuit as a composition of smaller circuits
    - Such a specification is called a structural specification
- Structural Specification can take on many forms:
  - A circuit composed of individual gates -> gate level rep.
  - A circuit composed of RTL components -> RTL rep.
  - A circuit composed of higher level blocks (decoders, etc.)
  - A circuit composed of only one top-level block

In general, there will be many levels of hierarchy

A Really Simple example

1. Start by describing each component

```
entity NAND_GATE is
      port ( A, B : in BIT;

                Z : out BIT);
end NAND_GATE;

architecture MY_DEFN of
    NAND_GATE is
      begin
          Z <= A nand B;
end MY_DEFN;
```

```
entity INV_GATE is
            port ( A : in BIT;

                Z : out BIT);
end INV_GATE;

 architecture MY_DEFN of
      INV_GATE is
      begin
            Z <= not A;
 end MY_DEFN;
```

A Really Simple example

```
entity AND_GATE is
    port ( IN1, IN2 : in BIT;
           OUT1 : out BIT);
end AND_GATE;
```

```
architecture structural of AND_GATE is
    component NAND_GATE
        port (A,B: in BIT;
              Z: out BIT);
    end component;
    component INV_GATE
        port(A: in BIT;
             Z: out BIT);
    end component;
    signal X: BIT;
    begin
      u0: NAND_GATE
          port map(IN1, IN2, X);
      u1: INV_GATE port map(X, OUT1);
end STRUCTURAL;
```

Positional vs Named Notation

- In the previous example, we used Positional notation in the port map statements:

  port map (IN1, IN2, X);

  – This means that IN1 is connected to the first port in the entity definition and IN2 is connected to the second port in the entity definition, etc.

- Can also use named notation:

  port map (A=> IN1, B=> IN2, Z=> X);

  – This means that IN1 is connected to A in the entity def^{n}, etc.

Quick reminder: Packages and Libraries

- A package is a collection of:
  - Component declarations
  - Types, subtypes, and constants
  - Procedures and function delcarations

- A library is a collection of packages

Quick reminder: Packages and Libraries

- Example of how to use stuff from packages in your code:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

  - This indicates that you want to use the library called IEEE
  - Within the IEEE library, you want to use packages:
    - STD_LOGIC_1164 and STD_LOGIC_ARITH

Describing Buses

- Type “BIT” maps nicely to one wire, but also need buses (sets of parallel wires)

  signal x: BIT_VECTOR (2 downto 0);

  – This example defines x as a bus with 3 parallel wires. This is like an array in a programming language. Each element of the array is of type BIT.

  – BIT_VECTOR signals can also be used as input/output signals of design entities

Describing Buses

- The size of a bit-vector and the element numbers are assigned when the signal is defined:

```
signal x: bit_vector(0 to 7); --8 elements accessed using indices 0-7
signal y: bit_vector(8 to 15); --8 elements accessed using indices 8-15
signal z: bit_vector(7 downto 0); --8 elements indexed 7 -0
```

  - Remember you can assign values to all elements in an array at once

```
x <= “00011100”;
```

  - You can copy one bus to another

```
out_bus <= in_bus; -- both must be declared identically
```

Describing Buses

– You can access individual elements of bit_vectors

```
signal x: BIT;
signal bus: BIT_VECTOR(15 downto 0);
x <= bus(6); --copy element 6 to x
```

– You can access “slices” of vectors:

```
signal main_bus: bit_vector (31 downto 0);
signal opcode: bit_vector (3 downto 0);
opcode <= main_bus(31 downto 28); -- 4 bits
```

Describing Buses

– You can concatenate short vectors to produce longer vectors

```
signal main_bus: bit_vector(7 downto 0);
signal short_bus: bit_vector(2 downto 0);
```

– The following assignments are legal:

```
main_bus <= “0000” & “1111”;
main_bus <= “01011” & short_bus;
main_bus <= ‘0’ & “0001111”;
main_bus <= ‘0’ & ‘0’ & ’0’ & ‘0’ & ‘1’ & ’1’ & ‘1’ & ’1’;
```

Describing Buses

– You can operate on entire bit vectors

```
signal sigA, sigB, sigC: bit_vector(7 downto 0);

sigC <= sigA and sigB;  --This is a bitwise and if
                        --sigA = 0x85 & sigB = 0x97,
                        --what is sigC
```

– You can make 2-D arrays:

```
Type REGARRAY is array (3 downto 0) of
                        bit_vector ( 7 downto 0);
signal R: REGARRAY;
```

Look into predefined attributes, generics and parameters to see how you can avoid hard-coding numbers

Review STD_LOGIC and STD_LOGIC_VECTORS

- STD_LOGIC
  - Like BIT, but signals of STD_LOGIC can take values other than ‘0’ or ‘1’ (like BIT), also:
    - ‘Z’ high impedance (a signal no one is driving)
    - ‘X’ unknown (a signal who’s value is unknown)
    - ‘U’ uninitialized (a signal that has not been initialized)
- STD_LOGIC_VECTOR
  - Array of STD_LOGIC elements (analogous to bit_vector)
- ***Use STD_LOGIC(_VECTOR) not BIT type for all of your designs (my advice)***

Readable logic equations: WITH and WHEN

```
SIGNAL x1, x2, sel, f: STD_LOGIC;

WITH sel SELECT
    f <= x1 WHEN ‘0’,
         x2 WHEN OTHERS
```

- This is the same as:

```
f <= (x1 and …
```

Readable logic equations: WITH and WHEN

```
SIGNAL x1, x2, f: STD_LOGIC;

    f <= '1' WHEN x1= x2,
         ELSE '0';
```

- The above statement is equivalent to:

```
f <=
```

Readable logic equations: WITH and WHEN

```
ENTITY priority IS
      PORT(req1, req2, req3: IN STD_LOGIC:
                  f: OUT STD_LOGIC_VECTOR(1 downto 0));
END priority;
ARCHITECTURE behavioural OF priority iS
BEGIN
      f <= “01” WHEN req1 = ‘1’ ELSE
            “10” WHEN req2 = ‘1’ ELSE
            “11” WHEN req3 = ‘1’ ELSE
            “00”;
END behavioural;
```

This is equivalent to:

```
f(0) <=

f(1) <=
```

Sequential Logic

Sequential Circuits

- Two types of circuit building blocks that are Sequential:

1. State Machines: You saw these in ensc250?
   - From a black box view, like a combinational block, except that outputs depend on previous inputs and current outputs
   - “memory” or “state” is stored in flip-flops

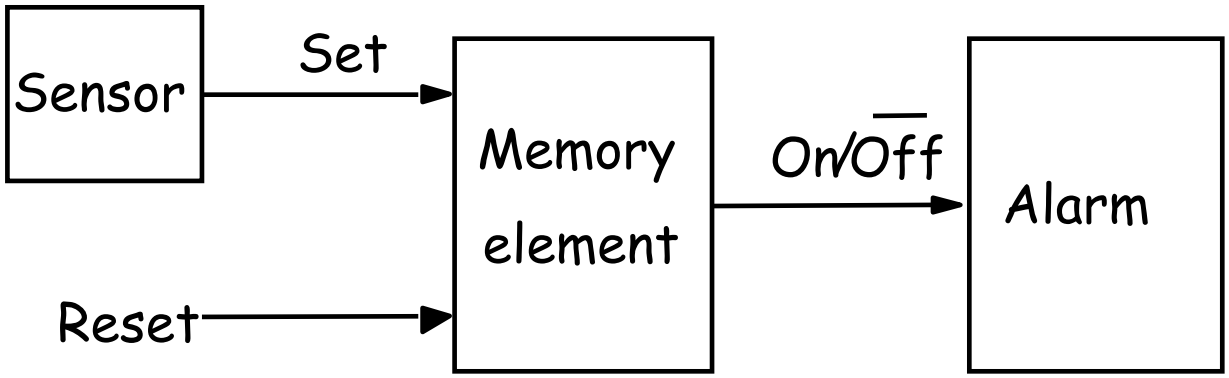
2. Datapaths: You’ll see more of these in this course
   - The part of the circuit that actually does the work
   - Usually contains flip-flops

– During this course, we will talk about both state machines and datapaths, but first we will review the basic flip-flop types.

A simple example of a sequential circuit

Spec: Alarm stays on after sensor "set" pulse until "reset" is asserted.

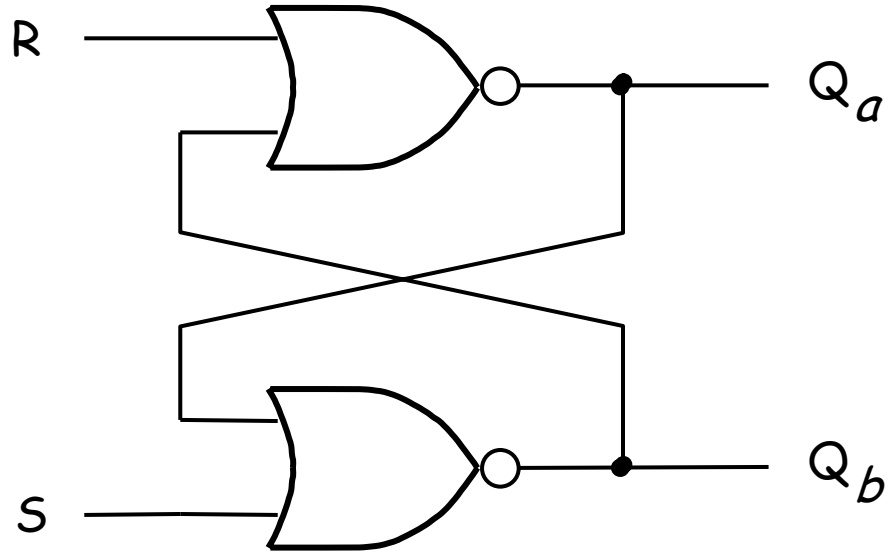
NOTE: Can't satisfy this specification with combinational logic!



B&V: Figure 7.1. Control of an alarm system.

Step by step flipflop creation

- First we start with a set/reset latch



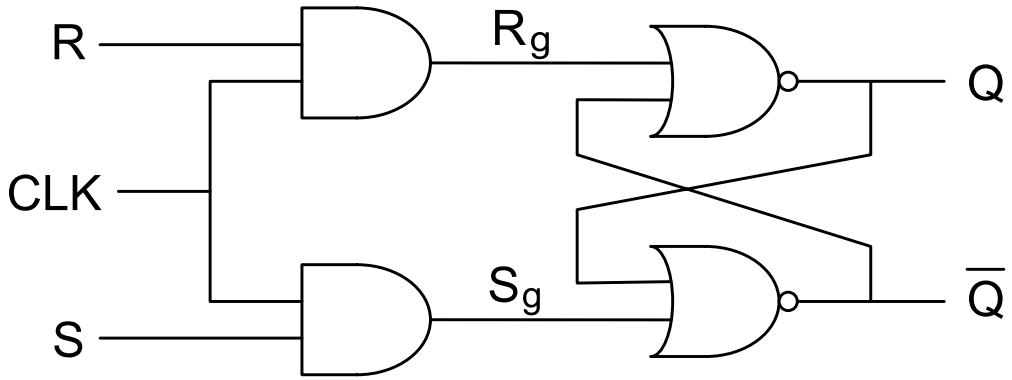
  - Try doing a practice table with S,R, Qa (old), Qb(old) to create Qa (new) and Qb(new)

Step by step flipflop creation

- Remember for a set/reset latch
  - If set goes high, Qa goes to 1
  - If reset goes high, Qa goes to 0
  - If neither set nor reset goes high, the output maintains its value
- What about if S and R are high at the same time?
  - Would not normally use an S-R latch this way

Step by step flipflop creation

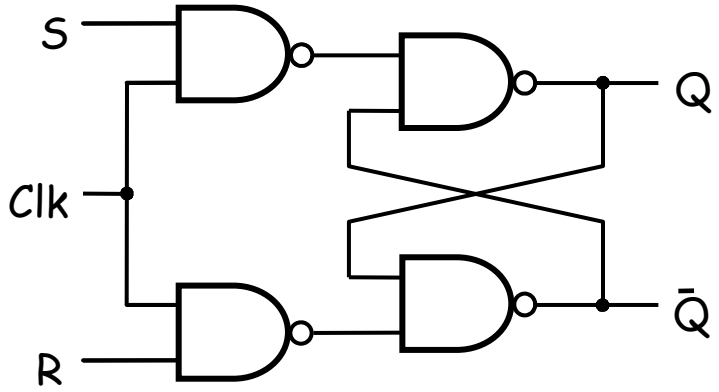
- Second, we transform it into a gated set-reset latch



  - When clk is high, it operates like a normal S-R Latch
  - When clk is low, the latch maintains its value regardless of R and S
  - This called level-sensitive as operation depends on the level of the clk signal

Step by step flipflop creation

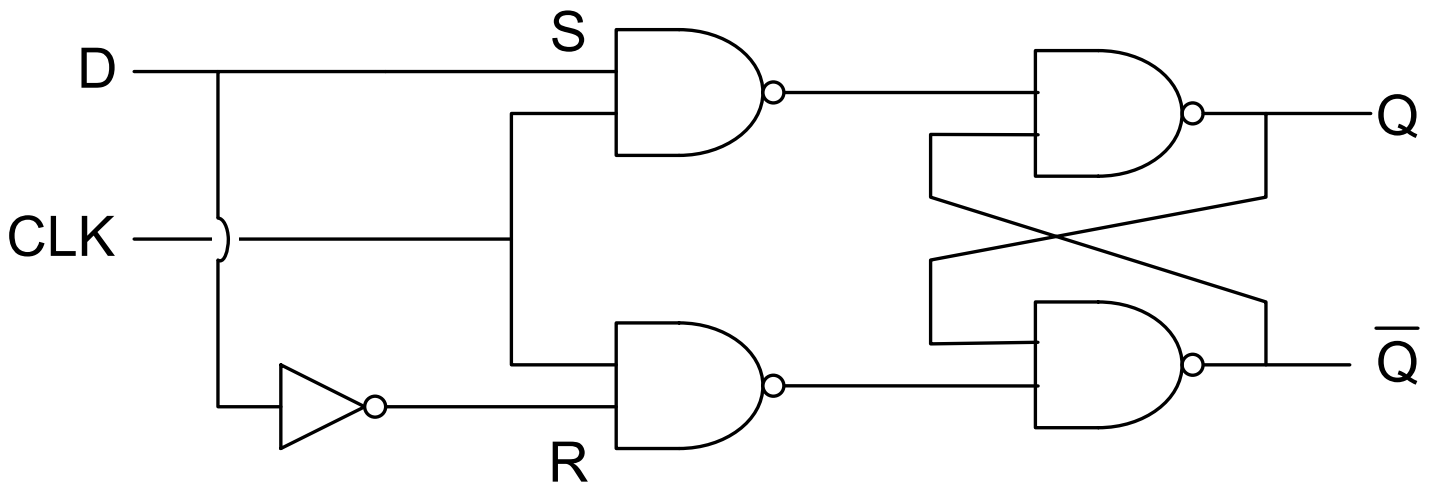
- Another gated set-reset latch implementation:



  - Why should you care?
    - It turns out it uses fewer transistors (you should learn about this if you take an advanced digital electronics course)

Step by step flipflop creation

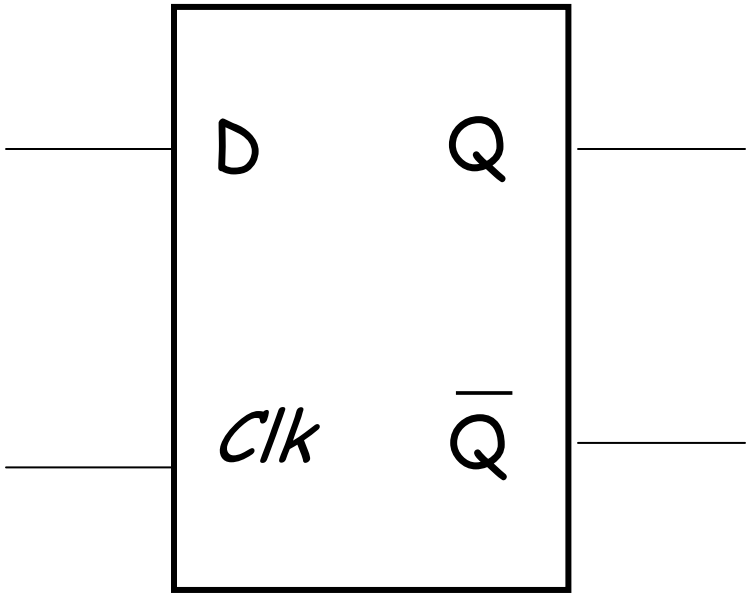
- Third, we create a gated D-Latch:



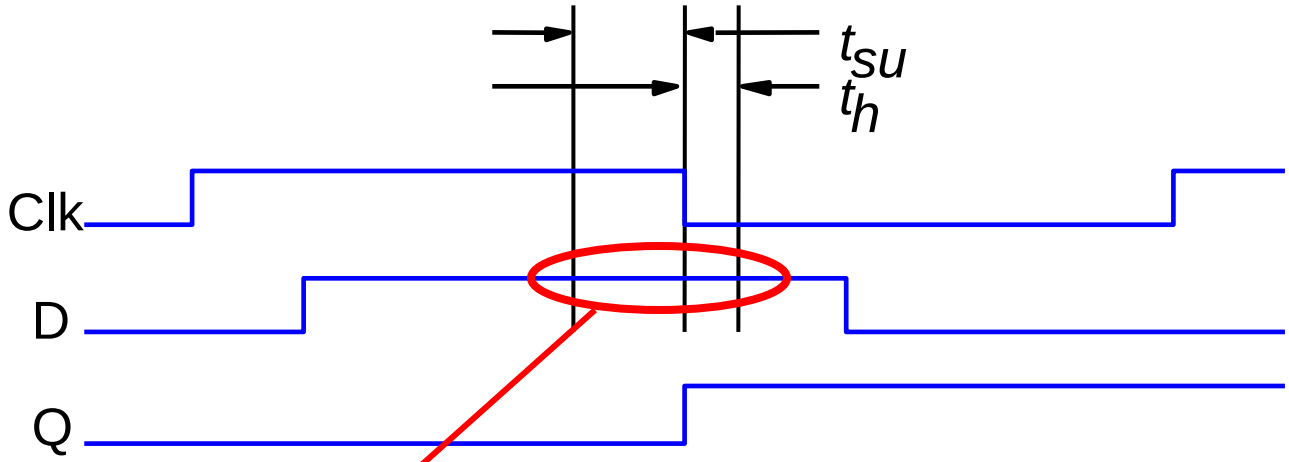
  - If clk is high, Q becomes equal to D
  - If clk is low, Q maintains its value (regardless of D)
  - This is also level-sensitive

Level-Sensitive D-Latch

- The Symbol for the level-sensitive D-latch:



D-Latch Setup and Hold Times

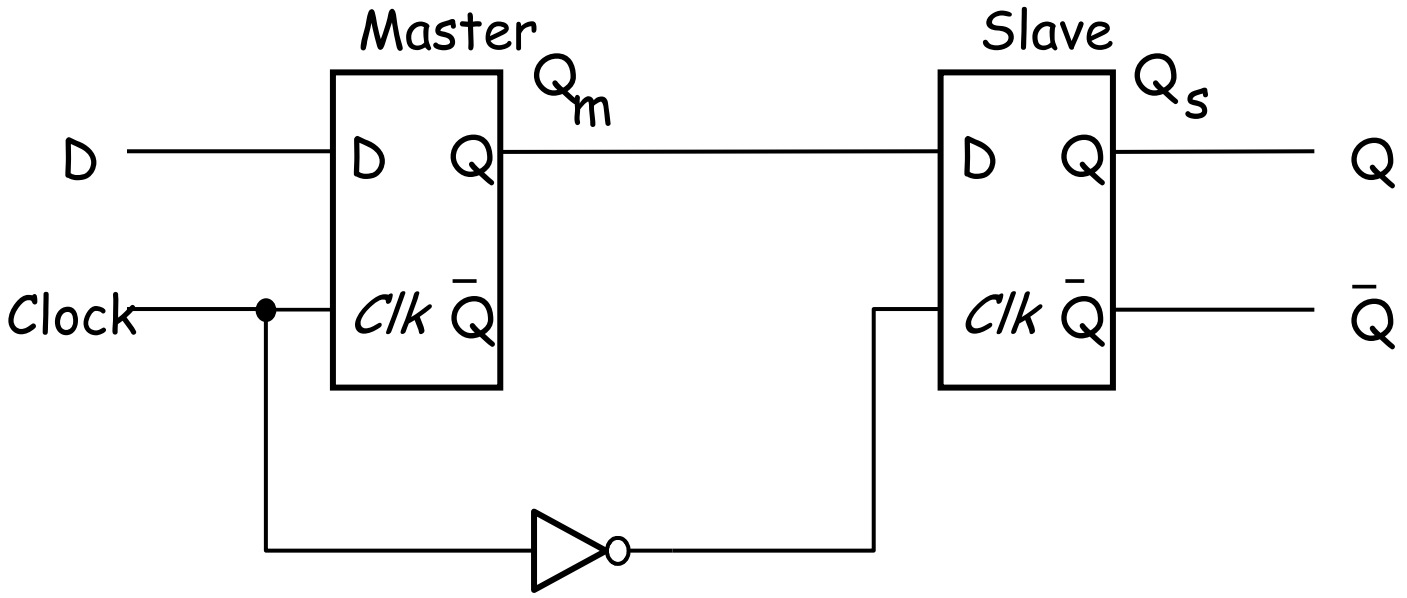


B&V: Figure 7.9. Setup and hold times.

D value must be "stable" (not changing) when Clk goes from 1 to 0.
If you don't do this…. METASTABILITY!!! (I'll tell you what this is later, but for now, trust me, it is bad).

Final Step: Edge Triggered D-Flipflop

- This is also called the Master-Slave flip-flop



Timing Diagram for Edge triggered D-flipflop

Clock

D

Qm

Qs

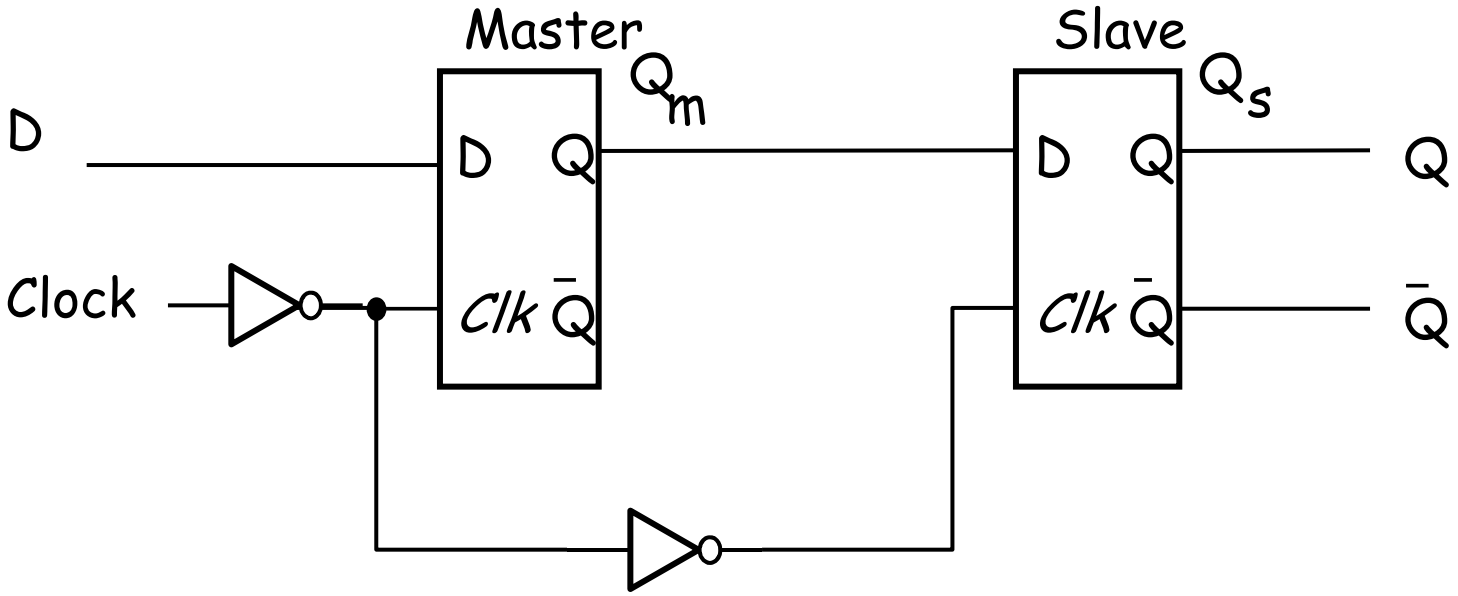
- Is this a rising or falling edge flipflop?

Edge-Triggered D-flip flop Notes

- When Clk changes from 1 to 0, the value on D is copied to Q (“snapshot”)
- At all other times, Q maintains its value
- Note that Q only changes when clk falls from 1 to 0
  - This is called negative edge triggered flip-flop
  - We can also have a positive edge triggered flip-flop

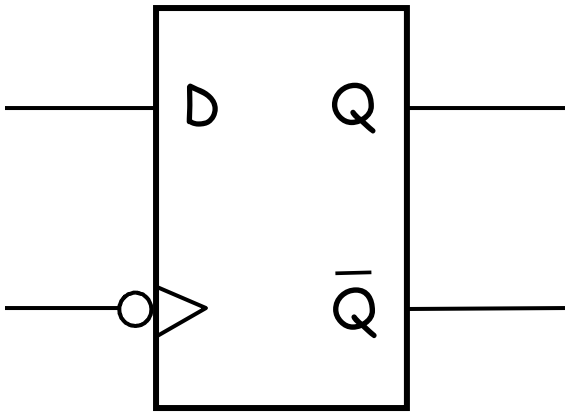
Positive edge triggered D-flipflop

- AKA the Master-Slave flip-flop

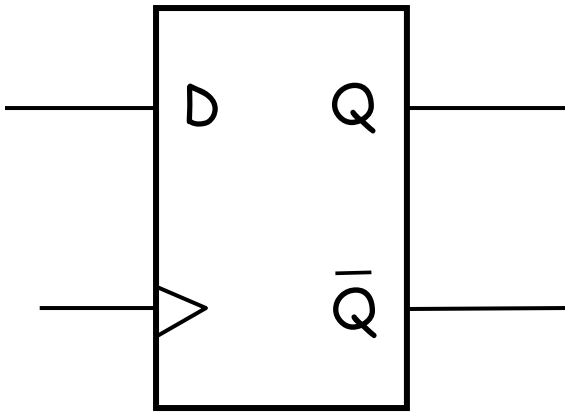


When Clk changes from 0 to 1, the value on D is copied to Q ("snapshot"), otherwise Q maintains its value

Edge triggered flipflop symbols

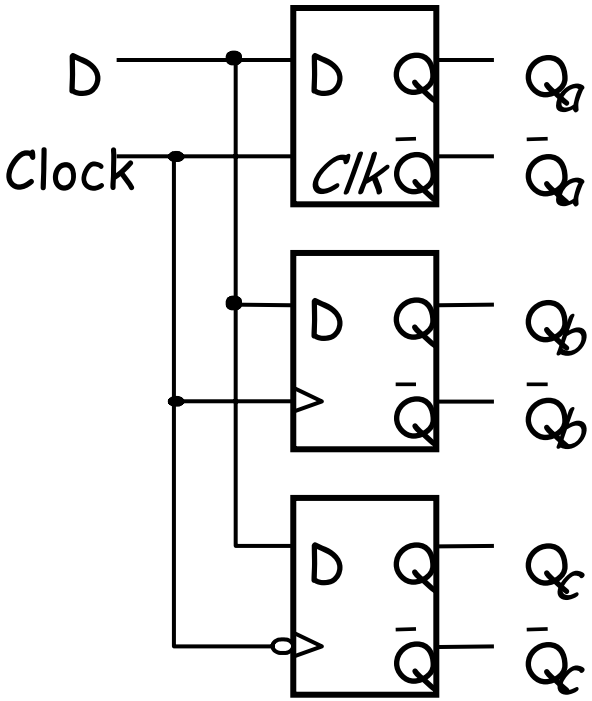


Negative Edge Triggered

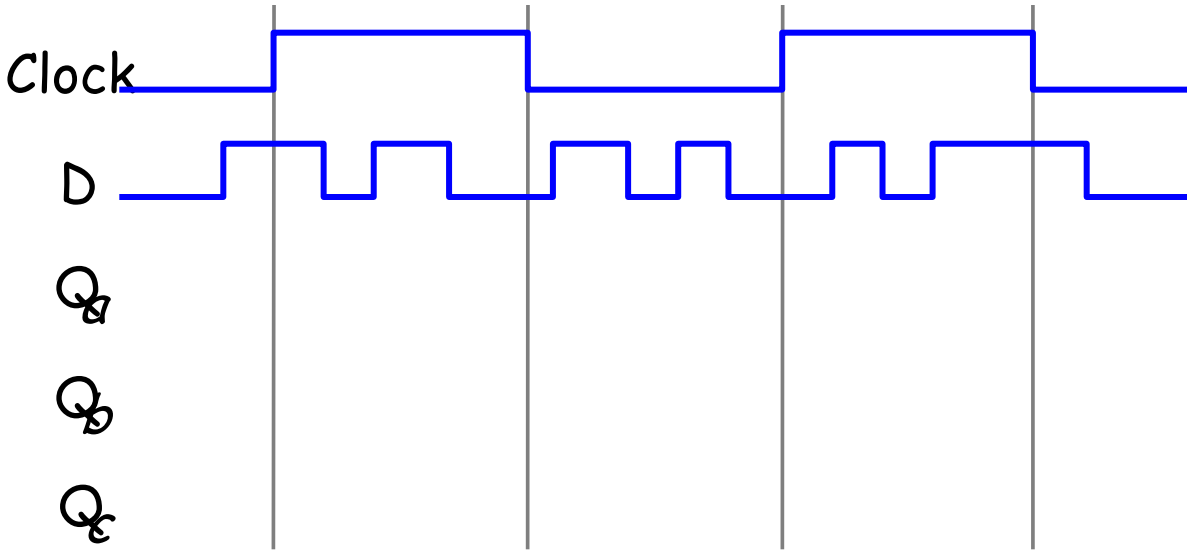


Positive Edge Triggered

D-Latch, rising and falling edge D-flipflops



(a) Circuit



(b) Timing diagram

Flipflop Notes

- You can also find references to T Flipflops and J-K Flipflops in books, etc
- In modern design, these aren't really important
- In most digital designs people use only edge-triggered D flipflops
- An important part of any discussion on FlipFlops is synchronous versus asynchronous resets
  - It's coming soon

Flipflop Notes

- Now that we have created a single bit memory device with reset capabilities (to be discussed later)
  - How do we create registers and larger memory units
- An n-bit piece of data can be stored in a register made of n parallel flipflops:

Flipflop Notes

- Multiple D flipflops can also be cascaded to form a serial shift register
- The output of each stage is delayed by one clock cycle from its input
  - This can be used to pipeline data
  - Typically part of encryption algorithms where the shifted data is accessed in parallel

Summary

- We went over combinational logic
- We talked about what VHDL is and why we use it
  - Originally a method of specifying the behaviour of a circuit
  - Now, also a source for synthesis
- We went over the implementation of some simple combinational logic circuits using VHDL
- We also went over latches and flipflops, i.e. sequential circuit fundamentals

- Next we'll go over implementing sequential circuits in VHDL

Questions

- What does VHDL stand for?

- Is VHDL a programming language?

Questions

- Convert the following numbers to decimal:
  - 1001
  - 0xA
  - 1010
  - 1100
  - 0xC
  - 1111
  - 0x9

Questions

- Convert the following numbers to hexadecimal:
  - 1001 1100
  - 1010 0011
  - 1010 1111
  - 1101 0101
  - 0100 1000
  - 1111 1011
  - 0001 0111

Questions

- What's the difference between structural and behavioural VHDL?

- What is synthesis?

Questions

- Is all HDL code synthesizable?

- What is the difference between an encoder and a priority encoder?

Questions

- How do you use “components”?

- What is the relationship between packages and libraries?

Questions

- SigA (8 downto 0) vs SigB (1 to 9) – what are the MSBs for SigA and SigB?

- What's the difference between "and" and '&'?

Questions

- What's the difference between a (normal) register and a shift register?

- What's the difference between a latch and a flipflop?

Questions

- Why do setup and hold time matter?

- What's a Master-Slave flipflop?

Questions

- Why do we care about how a circuit is implemented (e.g. how should you choose your implementation of a set-reset latch)?