Digital Design: exercise session 2

Weijie Jiang, Ce Ma, Wim Dehaene

January 16, 2025

1 Introduction

After finishing the hspice tutorial and the exercises with the inverter, we are now familiar with the hspice simulation environment. As shown in the first session, hspice is the engine behind any simulator to design circuits. Working at this level removes the abstraction of any gui. The goal of this session is the **design** of more complex gates with hspice. To **validate** their behaviour automatically, we will use .vec files. We will **characterize** in terms of speed (and symmetry), dynamic power and static power in each cell. We recommend using measurement statements (.meas) to automatize the process. A minimum size inverter is used as the output load to consider a more realistic scenario.

2 Environment setup

1. Boot/login in linux
2. Open a terminal and type the following lines:

```
cd ~/Documents/DigitalDesign/
cp -rP ~wdehaene/Public/DDIC/Project/DD_Es2 DD_Es2
cd DD_Es2
```

3 Introducing vector files in Hspice

3.1 What is .vec file?

The .vec file is used and called during the hspice simulation (E.g. line 39 in "NAND.sp"). This file mainly contains a list of input values and expected outputs. Therefore, the simulation can check the circuit's output for errors, meaning that the expected output does not match the output of the simulation. Otherwise, **no messages are displayed in the comparison waveform ".err0" file if the behaviour is correct.**

```
The comparison waveforem result values include,
0: matched
1: mismatched
X: ignored (output vector = X or bi-directional vector at input stage
are possible causes)
```

The simulator converts the digital vector waveform into a piecewise linear (PWL) waveform as shown in figure 1. The rising/falling edge occurs at the switching state point of the digital waveform.

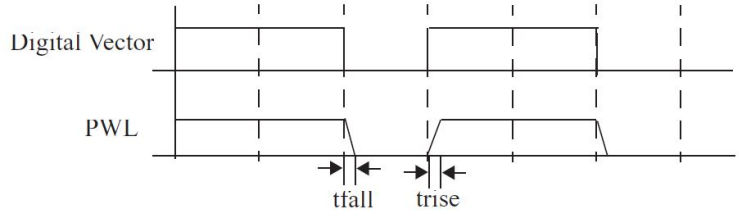


Figure 1: Conversion of Digital Waveform to PWL Waveform

3.2 Example: NAND.vec

Open the associated "NAND.vec" file. It is divided in two main groups. The top part defines the information on vector size (in bits), name, voltage, and timing slots. The **bottom part defines the input values to be loaded into the circuit and the expected output**.

Read and verify the definition of basic vector parameters. For some parameters, you can provide **a mask bit** to specify to which circuit's nodes you apply the parameter. If the mask is not specified, the setting applies to all vectors (input, output, and bidirectional).

- **radix**: Specifies the size (in bits) of the vector. If the radix is 4, the vname can use names such as name[3:0] and name[0:3].

```
radix vector1_size1 vector2_size2 ...vector_sizeN
```

- **vname**: Assigns a name to each vector. **The name MUST match the circuit's nodes.**

```
vname name1 name2 ... nameN
```

- **io**: Defines the type of vector. *i* states for input, *o* for output and *b* for bidirectional.

```
io type1 type2 ...typeN
```

- **tunit**: Sets the time unit for all time related variables. The time unit can be one of the following: fs (femto-second), ps (pico-second), ns (nano-second), us (micro-second), and ms (millisecond). The default time unit is 1 ns.

```
tunit time_unit
```

- **period**: Indicates the time between the digital transitions in the list below.

```
period time
```

- **tdelay**: Specifies the delay time for corresponding vectors.

```
tdelay time [mask1 mask2 ... maskN]
```

- **vih/vil**: Specifies the logic high/low voltage of the input vector.

```
vih voltage [mask1 mask2 ...maskN]
*Example: vih 1.0 1 1 0 (node 'A' and 'B' have a high input voltage of 1v)
*Example: vil 0.0 1 1 0 (node 'A' and 'B' have a low input voltage of 0v)
```

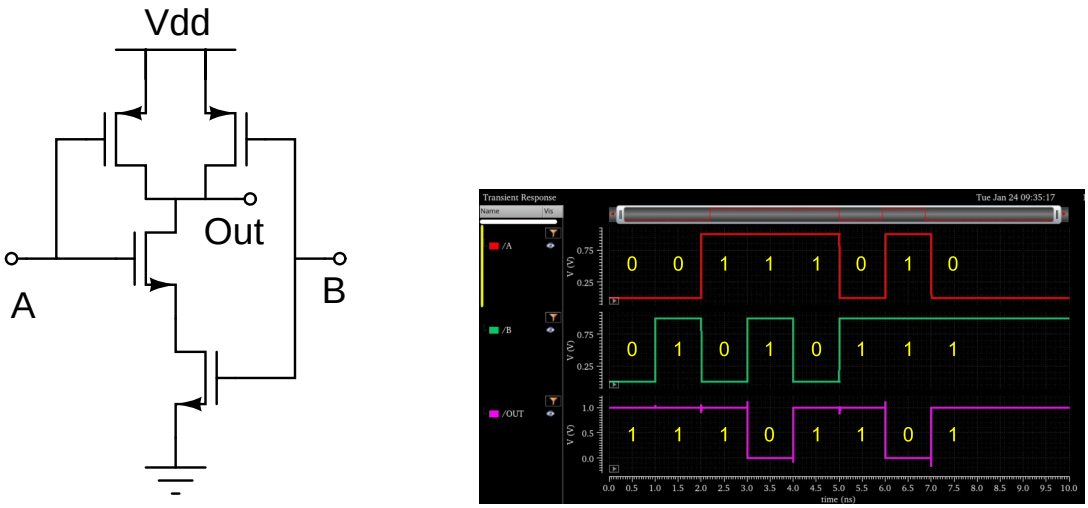
- **Vth**: **Threshold voltage to determine a transition in the output.**

```
vth voltage [mask1 mask2 ... maskN]
```

- **trise/tfall**: Specifies the rise/fall time of the input vector.

```
trise time [mask1 mask2 ...maskN]
tfall time [mask1 mask2 ...maskN]
```

4 Exercise 1: Standard NAND gate design



(a) Schematic (b) Vector pattern

Figure 2: Example for CMOS NAND gate

Design the NAND gate of figure 2a:

1. Describe the circuit (or subcircuit) of the standard NAND gate within the file "NAND.sp".
2. Fill the bit patterns (truth table) for node 'A' 'B', and 'out' in "NAND.vec" (example of pattern in figure 2b). Note: Make sure that the input and output pins of the gate match the *vname* parameter used in the .vec file.
3. **Match the driving capability by scaling the transistors appropriately.**
4. Using .captab compare the resulting input capacitance to that of an inverter.

Verify the correct behaviour of your design:

1. Make sure the .vec simulation is successful (file ".err0" is clean).
2. Plot the simulation output signals and verify that they match with your bit pattern (see figure 2b).
3. Verify the transistor's scaling by analyzing the symmetry of the rise/fall edges. You can measure it manually or use the measurement statements (.meas) as performed in previous session.

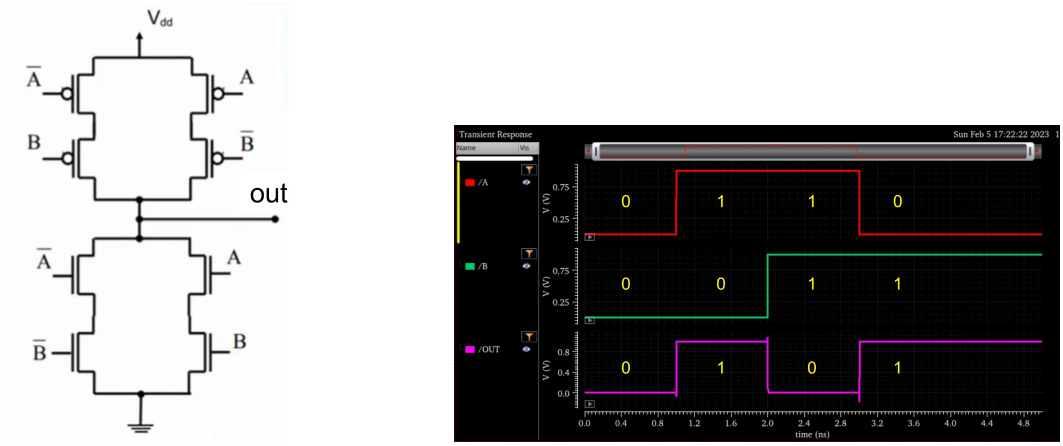
```
.MEAS TRAN Rise_Delay_A trig V(A) val='supply/2' rise=2
+ targ V(out) val='supply/2' fall=2
.MEAS TRAN Fall_Delay_A ....
.MEAS TRAN Rise_Delay_B ....
```

5 Exercise 2: Location of the inputs in the gate

As a first improvement, have a look at the location of inputs of the gates with reference to the output. In figure 2a, you can see the implementation of a NAND gate with inputs A and B. Please use the vector pattern shown in figure 2b.

1. Which one will be faster from a theoretical point of view?
2. Calculate the rise delay and fall delay in hspice using the measurements statements.
3. Run the simulation to evaluate this with the software.
4. Check the simulation results in "NAND.mt0"
5. Did you predict correctly which one was faster?
6. Create a load of 16 minimal inverters in "NAND.sp", what is the difference in speed between both inputs?

6 Exercise 3: Standard XOR CMOS gate design



(a) Schematic (b) Vector pattern

Figure 3: Example for CMOS XOR gate

1. Design a new circuit (or subcircuit) for a standard XOR gate (figure 3a) in "XOR.sp". **Match the driving capability by scaling the transistors appropriately.**
2. Create your own .vec file "XOR.vec" to evaluate the behaviour correctness (see figure 3b). Note: Make sure that the input and output pins of the gate match the naming used in the .vec file. Make sure that Vth and period times are correct.
3. **Characterize the cell in terms of speed, dynamic power and static power**. You can measure it manually or using the below measurement statements (.meas). Note: To measure the static power set a dc voltage on the input (as performed in session 1) but using the .vec file.

```
.MEAS TRAN Rise_Delay_A trig V(A) val='supply/2' rise=1
+ targ V(out) val='supply/2' rise=1
.MEAS TRAN Fall_Delay_A trig V(A) val='supply/2' fall=1
```

```
+ targ V(out) val='supply/2' rise=2
.MEAS TRAN Rise_Delay_B trig V(B) val='supply/2' rise=1
+ targ V(out) val='supply/2' fall=1

.MEAS dynpower avg power from 0n to 5n
```

7 Exercise 4: Another XOR gate topology

We are going to use the XOR topology depicted in figure 4.

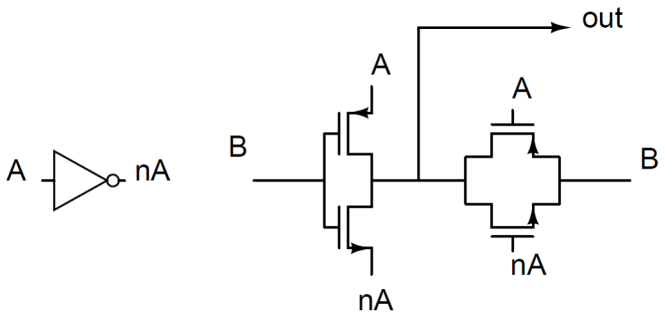


Figure 4: The 6T XOR

1. Why does it behave as an XOR gate? Draw the truth table if necessary.
2. Implement the XOR in "XOR.sp" (don't remove the previous CMOS XOR design) and verify its correctness reusing the "XOR.vec".
3. **Characterize the cell in terms of speed, dynamic power and static power**.

8 Exercise 5: XOR comparison

Comparing both XOR architectures in terms of speed and power:

1. Which are the benefits/drawbacks of each topology?
2. Which one do you think is the best?
3. When would you use each topology in a design?