**ECE 6213 – Design of VLSI Circuits**

**Fall 2022 – The George Washington University – Dr. Jerry Wu**

**Osama Yousuf – HW 5**

**Problem 1 – Storage Device Comparison – 10 pts**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Primary Memory** | | **Secondary Memory** | | |
| **Random Access Memory (RAM)** | **Read Only Memory (ROM)** | **Hard Disk Drives (HDD)** | **Flash Memory** | **Solid State Drives (SSD)** |
| A computer’s RAM is volatile – data is lost if the system is powered off. | ROM is non-volatile – data is maintained even if the system is powered off. | All of these secondary memory devices are used for storage purposes and are thus non-volatile. | | |
| RAM is stateless. It is used to store information that needs to be used quickly. | ROM is stateful. It is often used by system designers to store critical OS information that cannot be electronically modified. | A hard disk drive uses traditional moving disks to store data. | Flash memory utilizes memory chips to store data. These chips do not move. | Technically, SSDs still use flash storage technology, but a more advanced version that can handle daily computer demands. |
| An example of a RAM chip is the DDR5 RAM memory. DDR (double data rate) is a technology allowing simultaneuous multiple file transfers. | An example of ROM is the bootloader on a system, often embedded inside a processor chip. | Since there are moving parts, data transfer rates are relatively slow. | Data transfer rates are faster than HDDs, because there are no extra moving parts. | SSDs are typically much faster than HDDs as well as flash storage because of advancements over the previous technologies. |
| HDD Storage is relatively cheap. | Flash drives are generally more expensive per storage capacity compared to HDDs. | SSD are the most expensive technology compared to HDDs and flash drives. |
| Primary memory is faster than secondary memory. | | Secondary memory has much higher latencies in general compared to primary memory. | | |

**Problem 2 – Programmable Logic – 20 pts**

**FPD/PLD:** In the most general terms, a field programmable device (FPD) or a programmable logic device (PLD) refers to any type of integrated circuit that can be used for implementing digital hardware. Opposite to ASICs, these devices can be configured by the end user to realize different designs.

**Classification of Programmable Devices**

**PLA vs. PAL**

* These are both simple PLD devices, or SPLDs.
* PLA stands for a Programmable Logic Array, whereas PAL stands for Programmable Array Logic.
* Both are relatively small FPDs.
* PLAs contain two levels of programmable logic: the first is an AND-plane, and the second is an OR-plane. By comparison, PALs have a programmable AND-plane but a fixed OR-plane.
* Logic expanders can be used to increase the flexibility of PALs but result in significant propagation delay.
* Because of higher programmability, PLAs are more expensive to fabricate compared to PALs.

**CPLD vs. FPGA**

* These are both more complex PLD devices compared to PLAs and PALs.
* FPGAs offer higher logic capacity than CPLDs.
* CPLDs consist of an arrangement of multiple PAL-like blocks on a single chip with a programmable interconnect to connect these blocks. On the other hand, FPGAs (or Field-Programmable Gate Arrays) consists of an array of programmable basic logic cells surrounded by programmable interconnect.
* Applications of CPLDs are to implement glue logics and circuits that can exploit a large number of AND/OR logic gates. Applications of FPGAs are much wider, ranging from prototyping, digital signal processing, and even image processing and network modelling.

SPLDs have the lowest number of equivalent gates, ranging from 0 ~ 200.

CPLDs are higher, ranging from 200 ~ 12,000.

FPGAs offer the highest logic capacity, where equivalent gates can be anywhere from 1,000 – 1,000,000.

**Problem 3 – Finite State Machines**

**Mealy State Machine**

Following is a list of key points for a Mealy state machine.

* The output is a function of the current state as well as the input
* The input can directly cause a change in the output, which might violate time requirements.
* Compared to a Moore state machine, a Mealy state machine often has less total number of states.

**Code:** *mealy.v*

Text

Description automatically generated

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

**Testbench:** *mealy\_tb.v*

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Testbench Features**

* My testbench is self-reporting
* It starts with a reset to the *WhiteState*, and checks for the validity of every state transition sequentially.
* I implemented events for:
  + Resetting the state machine
  + Updating testbench cycles
  + Watchdog for exiting the simulation
  + Looping over the various states of the state machine and checking for validity
* I implemented logic for counting errors in every test case, as well as for counting total errors in the simulation
* I also implemented a simple task for printing the status of the errors encountered for self-reporting

The final text output of my testbench is given below; all test cases pass without any errors:

A picture containing text

Description automatically generated

Waveform snippets for a few cases from my testbench are also displayed below:

Graphical user interface

Description automatically generated

* Firstly, the last 4 signals in my waveform indicate the state encodings for each of the four states. These are:
  + WhiteState *white\_state* 00
  + RedState *red\_state* 01
  + GreenState *green\_state* 10
  + BlueState *blue\_state* 11
* It can be seen that the asynchronous reset at the start means that both current and next states are white.
* When the red input is asserted, the next state becomes red, and the output of the FSM (*mealy\_out*) is asserted, both at the exact same time. This is correct expected behavior since we have a mealy FSM; the input causes a direct change with the output.
* When the red input is made 0 again, the FSM correctly cycles back to the white state.
* A similar thing happens for other transitions, namely from white state to green state (and back), and white state to blue state (and back).

**Moore State Machine**

Following is a list of key points for a Moore state machine.

* The output is a function of only the current state.
* As a result, the output changes only after a state transition has occurred, contrary to a Mealy state machine.
* Compared to a Mealy state machine, a Moore state machine often has higher total number of states.

**Code:** *moore.v*

Text

Description automatically generated

Text

Description automatically generated

**Testbench:** *moore\_tb.v*

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

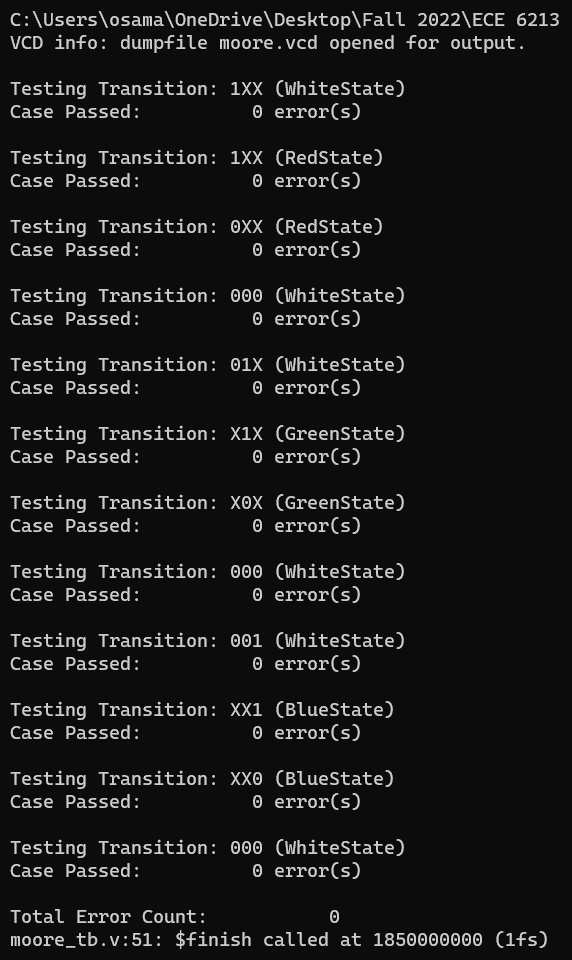
Text

Description automatically generated

**Testbench Features**

* My testbench is self-reporting
* It starts with a reset to the *WhiteState*, and checks for the validity of every state transition sequentially.
* I implemented events for:
  + Resetting the state machine
  + Updating testbench cycles
  + Watchdog for exiting the simulation
  + Looping over the various states of the state machine and checking for validity
* I implemented logic for counting errors in every test case, as well as for counting total errors in the simulation
* I also implemented a simple task for printing the status of the errors encountered for self-reporting

The final text output of my testbench is given below; all test cases pass without any errors:



Waveform snippets for a few cases from my testbench are also displayed below:

Graphical user interface

Description automatically generated

* Firstly, the last 4 signals in my waveform indicate the state encodings for each of the four states. These are:
  + WhiteState *white\_state* 00
  + RedState *red\_state* 01
  + GreenState *green\_state* 10
  + BlueState *blue\_state* 11
* It can be seen that the asynchronous reset at the start means that both current and next states are white.
* When the red input is asserted, the next state becomes red, but the output of the FSM (*moore\_out*) is not asserted at the same clock. This is correct expected behavior since we have a Moore FSM; the input does not cause a direct change to the output. The output is asserted in the next clock cycle.
* When the red input is made 0, the FSM correctly cycles back to the white state.
* A similar thing happens for other transitions, namely from white state to green state (and back), and white state to blue state (and back).

Note: Instead of $fopen and $fclose, I used the “>” operator to directly re-direct terminal output to a text files for both of my FSMs (Mealy and Moore), as follows:

**For Mealy output:**

* iverilog -o mealy mealy.v mealy\_tb.v
* vvp mealy > mealy\_output.txt

**For Moore output:**

* iverilog -o moore moore.v moore\_tb.v
* vvp moore > moore\_output.txt

The end result are identical text files that contain all of the $display information from the two test benches, as shown in the snippets below:

Graphical user interface, text

Description automatically generated

The waveform files are also provided in the fsm folder (in .vcd format).