Ceng232 Take Home Midterm2 Project

Table of Contents

Deadline

~~Original: May 22, 2020 9.00 AM~~

Postponed: May 25, 2020 9.00 AM

Introduction

You'll be implementing a 2FA(2-Factor Authentication) key fob. A 2FA key generator's task is to periodically generate a code that is unique to a user at any given time. One way to achieve this is to use a hashing algorithm (or similar) and feed it a user specific information (your student id) and the current time. Your lead engineer have already come up with a design but unfortunately she has become unavailable due to the current Coronavirus outbreak. So it's up to you to implement the whole project and report the design procedure along with the final outcome.

Project Outcomes

- **Implementations for module**; no testbench is asked as we will use our own testbenches for automated grading, but it's highly advised that you write and test all your units. Grading of the units, including the top module will be done on a pass/fail basis and no partial grades will be given. Because in the end, it either works or not, nothing else matters.
- **A handwritten final report**; this report should detail the design procedure of all units including -but not limited to- system block diagrams, flow diagrams, state diagrams if any, design decisions, assumptions and implementation details. For each unit you're expected to explain
    i. What does this unit do? *(i.e. high level function)*
    ii. How does it achieve *(i)*? *(i.e. low level function)*
    iii. Why did you decide to implement *(ii)*?
    iv. Are there any disadvantages to *(ii)*?
    v. How did you test this unit?

Delivery Format

The codes and report will be uploaded to separate OdtuClass assignments (only the last submission will be accepted for both).

1. Zip file containing all verilog files with their original names without subfolders.
2. *PDF* of photos/scans of **handwritten** report. No *MSWord*, no *ODT*, absolutely no *macOS Pages* will be accepted. Check out Adobe Scanner (https://acrobat.adobe.com/us/en/mobile/scanner-app.html) or Open Note Scanner (https://f-droid.org/en/packages/com.todobom.opennotescanner/) if you prefer something FOSS (much less features).
3. Total page count really shouldn't need to exceed 5 pages.

Grading Structure

- 80% Code Submission
    - Display (14%)
    - Clock Divider (14%)
    - Button FSM (16%)
    - Hasher (18%)
    - Timekeeper (8%)
    - Top (10%)
- 20% Report Submission
    - Display (3%)
    - Clock Divider (3%)
    - Button FSM (5%)
    - Hasher (4%)
    - Timekeeper (2%)
    - Top (3%)

Advice

- As a general rule of thumb, try and let us understand what was going in your mind but only the ones that contributed to the final outcome.
    - i.e. your lead doesn't need to know about the 99 problems you figured out
        - if you *really really* think it's important, it can go into the appendix.
- It's always a good idea to take notes/scribbles as you go along.

Overall Function *(high level function)*

This is an always-ON device, which is why it's very important to have a very low power consumption as much possible. This approach also significantly decreases mass-production costs. Using a custom design potentially increases the security by preventing software hacks which is sometimes a possibility when microcontrollers are used. This device has a 5-digit (decimal) display and a single button. When you press the button it activates the display which then shows the current 2FA key on the display until you press the button again. If the 2FA key changes while the display is on (because it's dependent on time), display shows new key. As long as the button is not pressed again, the display stays on. The 2FA-key should change every *N* seconds regardless of the display and/or button state(not very practical compared to the conventional 60s, but will make the simulations much faster).

Overall Design and Module Definitions Supplied

Display Module

Display Module's task is to convert a 16-bit unsigned binary number to 5 BCD(Binary Coded Decimal) numbers. Its job is to generate the decimal digits where these digits will be outputted in BCD. The digits are to be found from the decimal equivalent of the input binary number. It also has a 1-bit enable input. If the enable signal is '0' then display module's all BCD outputs should be "1111", otherwise it should perform its function. Module definition is provided in *b16toBCD.v*

Example

Input:

- 1110 0010 1011 0111 (decimal 58039)

Outputs:

- D5 : 0101 (decimal 5)
- D4 : 1000 (decimal 8)
- D3 : 0000 (decimal 0)
- D2 : 0011 (decimal 3)
- D1 : 1001 (decimal 9)

Clock Divider

You'll need 2 different clocks to be able to 1. trigger events for key computation and 2. to capture button input. Key computation should happen every **N** seconds, and your lead has chosen a 500Hz clock should be OK for capturing inputs without bouncing issues from a cheap button (search *"button bouncing problem"*). Your clock divider has a 1 MHz input clock (sys_clk) and it should output 2 new clock signals one with **N** seconds period and another with 500Hz frequency. N should be a verilog *parameter* called *KEYCHANGE_PERIOD*. Our testbenches will try to override this parameter. Module definition is provided in *rtcClkDivider.v*.

Button FSM

One of the stakeholders decided he doesn't like the whole *"you have to keep the button pressed to activate the display"* so your lead decided to emulate a stateful button behaviour with a regular button (search *stateful button*). And on top of that, none of the engineering chiefs seem to like tampering with clock signals. So for this purpose you'll need to design an FSM. And to allow for fastest response to button input, it has to be a Mealy FSM. Apart from a clock input, the machine has a single input button and a single output *display_enable*. Assume button input becomes *'1'* when pressed and bounces back to *'0'* when released. On reset the output is *'0'*. When the button is pressed the output should become *'1'* **immediately**. Until the button is pressed again, the output should stay *'1'*. When the button is pressed again, the output should become *'0'* **immediately**. Then, until the button is pressed again, the output should stay *'0'*. The behaviour from here is the same as reset state. The module definition is provided in *buttonFsm.v*.

Note that clock tampering is NOT allowed and this module MUST be implemented as a Mealy FSM. Please include state diagrams and tables in your report, along with explanations.

p.s. If the button is pressed and somehow released within a clock cycle, module would reflect the action for the duration of that press, but the machine whould not change states. This is expected behaviour and indeed a relatively-less sophisticated way to handle button bouncing.

Hasher

The hasher module is the brains of the operation. This is a synchronous sequential module that should calculate the next 2FA key on each clock cycle. Its inputs are a 16-bit unsigned *time*, and a 16-bit unsigned *studentid*. Its output is a 16-bit unsigned *cur_hash*. On each clock cycle it should compute, store and output the current 2FA-key. Its output should only change every clock cycle regardless of the arbitrary input changes. On reset, *cur_hash* output should be all zeroes. As such the initial input to hashing function should also be all zeroes. The module definition is provided in *hasher.v*.

Timekeeper

Timekeeper is a simple storage that counts the number of *N-second periods* passed since the power on. In every clock cycle it should increment its internal value and output that as current time. It has no inputs, but its initial value should start from 0. It has one unsigned 16-bit output *current_time*. If *current_time* reaches its maximum value it should just restart from 0. The module definition is provided in *timekeeper.v*.

Top

This is the module that brings everything together. It should have one 1-bit input *button*, and 5x 4-bit outputs D5, D4, D3, D2, D1, respectively representing the 5th*(most significant)*, 4th, 3rd, 2nd and 1st*(least significant)* decimal digits of 2FA-key. The expected behaviour is explained in overall function section. The module definition is provided in *"top.v"*

All modules described above will be tested against various testbenches and will be graded automatically.

Other remarks

- ~~Testbench clocks will start high. ClkDivider outputs should also start high according to this.~~
- Testbench clocks will start LOW. ClkDivider outputs should also start LOW according to this (reason why) (https://odtuclass.metu.edu.tr/mod/forum/discuss.php?d=17833#p40116).
- All modules should operate only on rising (positive) edge of the clocks.
- Minimum button press will be at least 5ms excluding the bounce periods.
- Testbench bouncing duration of button will be less than 2ms after and before button press and/or release (i.e. button will NOT bounce for or more than 2ms, t<2ms). But button input will be pretty much random during this time.
- KEYCHANGE_PERIOD will be more than 0.5 seconds. You can assume this will be an integer, but would be nice if you made it work regardless.
- Division and Modulo operators ARE allowed.
- The HTML example here (https://odtuclass.metu.edu.tr/mod/page/view.php?id=100318) only simulates the asked Mealy Machine behaviour and not the bouncing. Notice how "output" (not state!) immediately changes compared to 3rd one.
- Minor details are subject to change.

Hashing Function

of your colleagues came up with a super-mega-uber-secure (not really) hash function as described below.

```
cur_hash = ((((time ⊕ student_id) - prev_hash)^2) >> 8) & 0xFFFF
```

Where **⊕** is XOR, **^** is exponent (i.e. power) and, **>>** is logical shift right. A detailed C implementation is given below for a stricter reference. The first hash value to compute the next hash should be 0.

You should also convert your student id to contain 16-bits but also without leading zeroes. For that, take the modulus of your student id from 2^16 and then replace any leading zeroes with 5s. This part should NOT be coded into system, meaning you should do this by hand (we'd like to see this calculation and definitely the resulting "studentid" in report too).

```
If student id actual is 197010. Then:
197010 % 65536 = 402
16-bit student_id to be used should be then 55402
```

Overall, the computation of next 2FA-key value should be realised with respect to the C function definition below (search *"stdint"*). Assume that first value inside cur_hash reference is zero.

```
void compute_next_hash(uint16_t* cur_hash, const uint16_t time, const uint16_t student_id)
{
    uint32_t prev_hash = *cur_hash;
    prev_hash = ((time ^ student_id) - prev_hash);
    prev_hash *= prev_hash;
    prev_hash = ((prev_hash >> 8) & UINT16_MAX);
    *cur_hash = prev_hash;
}
```