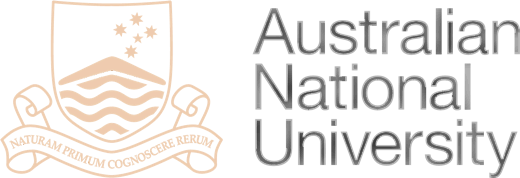
******

***Overview***

In this report, it shows our answer of Part 1 fundamentals and the report part of Part 2 Reaction Timer. The implementation of Part 2 is using the Digilent Nexys4 DDR. All the codes are written via Vivado 2017.4. The version we handled are fully tested and it could be synthesis, implemented and generating bitstream with no bugs.

Acknowledgement for the following people who gave us a help during the assignment:

Tutors:

* Yuxin Liu (Toby)
* Jordan Smith
* James Thomas Spollard
* Zheyuan Liu (David)
* Usama Elahi

Lecturers:

* Lyle Roberts
* Jonghyuk Kim

Laboratory:

* Xianjun Zheng

***Assignment One***

***Fundamentals & Reaction Timer***

u5870415 Haolei Ye

u5698699 Fangxiao Dong

# Part 1: Fundamentals

# Part 2: Reaction Timer Summary

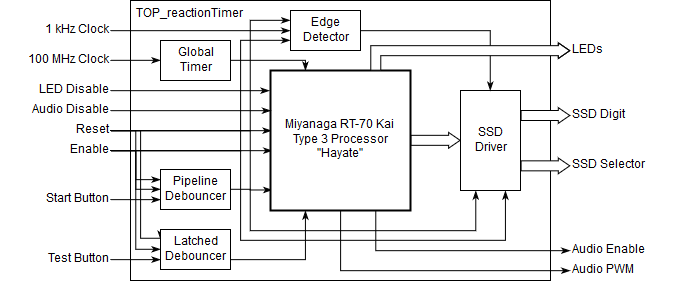
In this part, it shows the main feature list and high-level design of our reaction timer implementation. It also explains the details of how we implement these features. At the last of this report, we discuss the mean we applied testing during our development.

# Features

* Implement the four period of the Reaction Timer specification on Nexys4 DDR
* [Advanced] Shows the best record time and LED animation at the idle state
* [Advanced] Automatically return to idle state after display the reaction time for 10 seconds
* [Advanced] Display the reaction time as a 7-digit decimal floating point number in second
* [Advanced] Using Linear Congruential Generator and Mersenne Twister MT19937 as the random number generator to set random event time
* [Advanced] The recent reaction time would be to flash if it is the best time
* [Advanced] If tester pressed the button before the reaction signal, the reaction timer will mark this as a failure
* [Advanced] Support mono audio output as the optional reaction signal

# Architecture

The following diagram shows the top level design of the reaction module.



The input wires of the global module are 4 switches, 2 buttons and system 100MHz clock. One switch uses as reset, one uses as enable, one uses as audio output disable, and the other one uses as LED output disable. One button uses as the test start button and the other one uses as test button. The output wires are 16 LEDs, 7-segement display and mono audio outputs.

The top module architecture is designed for processing the input and output signal directly to the FPGA board. All the input and output pins are chosen to be used CMOS 3.3V standard. The clock used in this design is the 100MHz clock provided by the crystal on the board. An 1kHz clock generated from the 100MHz clock is used as the clock for the Seven-Segments Display to select the digit.

The edge detector has the same logic as the one we used in Lab 4.

## Modules at Top Level

### Input Signal Debouncing

For all the switches used in this reaction timer, it doesn’t need any debouncer. But for those two buttons, they need different debouncing methods.

For the start button, it is only used as a signal for starting the test. So the debouncer for start button is just the sample pipeline debouncer as we implemented in the lab. The pipeline is described as a single register. The different from the lab version is the pipeline level could be modified by a parameter. If we still use the way as what we did in the Lab, the value used to detect the button pressed would be 2n+1-1 in decimal. To reduce the work of FPGA, we store the inverse of the button instead of its original value. When the button pressed for a while, the value of the pipeline should be 0. The debouncer module also uses logic equal to avoid the high impedance value comparison. Due to the limitation of the integer is 32-bit as default, hence this module could support up to 32-level pipeline. In the implementation of reaction timer, this level has been set to 5.

For the test button, it needs for lower latency to get accurate result. Here, we use a latched debouncer for the test button, which allows us to reduce latency within 10ns. The edge detector would add another 10ns delay to the signal. Hence, the total latency of the test button pressed is 20ns for these two modules. However, this won’t affect the precise of our final result. It will explain further later.

### Global Timer

The global timer module is a simple counter start running at the very beginning. It works like the UNIX timestamp. There is a 32-bit unsigned register and initializes as 0. This register increase 1 for each 100MHz rising edge. Hence, it would warp around every 42,949,672,950 nanoseconds, which is around 43 seconds. This value doesn’t reset neither reset signal is high nor enable signal is low.

This global timer would be used as the seed of the randomized number generator. This makes the possibility of the same interval as its previous trial reduce to every 42 seconds.

### Seven-Segments Display (SSD) Driver

The SSD driver have 13 inputs: eight 4-digit Enhanced Binary Coded Decimal (EBCD) number for each digit, one 8-bit dots display expression number, 1kHz clock rising wire, 100MHz system clock, reset and enable. Only 2 outputs are need for this module: one 8-bit output for the digit display with the point and one 8-bit for the digit selector. In the module implementation, the dot output wire separates from the digit expression.

There is a 3-bit register used as the digit selector. This number increases at each 1kHz clock rising edge. The digit selected integer could be calculated with index (displayIndex) with expression ~(8'd1 << displayIndex).

The EBCD is based on the standard Binary Coded Decimal (BCD) and defines some characters for the reaction timer. It expresses a character with a 4-bit unsigned integer. The following table shows the definition of the EBCD.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EBCD | | Glyph | EBCD | | Glyph | EBCD | | Glyph | EBCD | | Glyph |
| 0000 | 0 | 0 | 0100 | 4 | 4 | 1000 | 8 | 8 | 1100 | 12 | A |
| 0001 | 1 | 1 | 0101 | 5 | 5 | 1001 | 9 | 9 | 1101 | 13 | F |
| 0010 | 2 | 2 | 0110 | 6 | 6 | 1010 | 10 | - | 1110 | 14 | I |
| 0011 | 3 | 3 | 0111 | 7 | 7 | 1011 | 11 | blank | 1111 | 15 | L |

The first 10 characters are the same as the normal BCD. Letters ‘A’, ‘F’, ‘I’, ‘L’ are defined for showing the word ‘FAIL’ when user failed to pressed the button within 9.9 seconds. The minus character is used to display the best record number when there is no valid record. Eight minus characters would be shown on SSD.

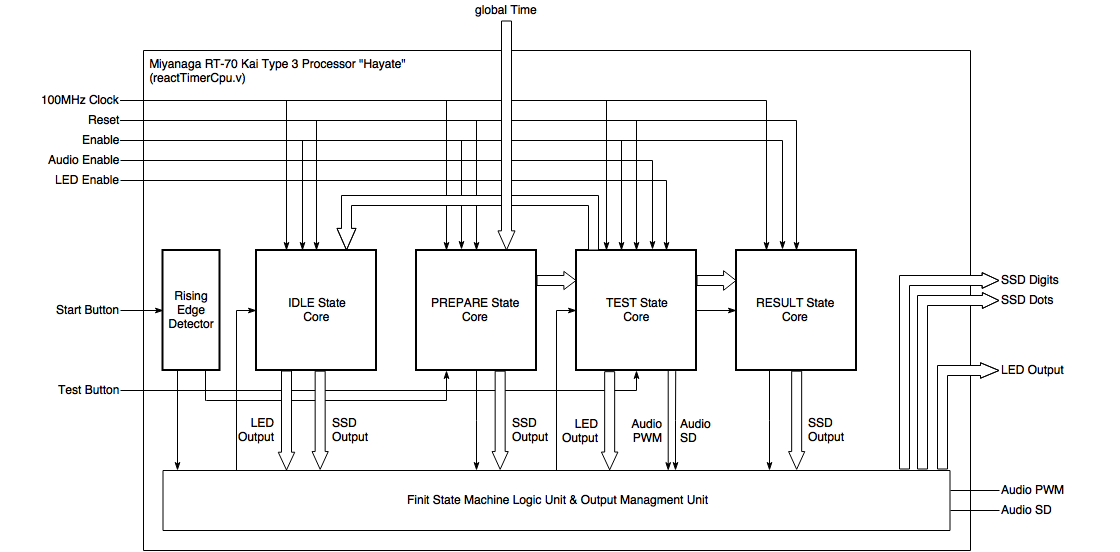
The EBCD code mapping is implemented in the SSD driver module as a single case statement. When the driver is at reset or disable state, it would output blank character with no dot displayed.

## Central Logic Unit (Miyanaga RT-70 Kai Type 3 “Hayate”, M7T3)

This is the module (**M7T3**) which implements the finite state machine (FSM) of the main reaction timer logic. It contains quad cores inside the module for each state of the FSM and a unified FSM logic switching logic and output management unit. The entire module works as a multiplexer which output the correct signal of the state core module according to the current state of the FSM.

The input signal of **M7T3** are quite simple. Besides the standard 100MHz system clock, reset, enable wires, only four other wires are needed: start test button, test button, audio hint enables and LED hint enables. Outputs are just LED, SSD and audio outputs.

The following diagram shows the inner structure of the **M7T3** processor.



The FSM logic unit inside **M7T3** manages the state switching of the state. There are four states as the specification describes. The responsibility of each state and the policy that transfer between these states are

* IDLE: display the best score and waiting for the start pulse signal.
  + Start button pressed: move to PREPARE state.
  + Or else: play the LED animation.
* PREPARE: generate the random number and show the countdown animation on SSD.
  + Wait both the countdown animation and random number generation complete.
  + Or else: output the random number to TEST state and move to TEST state.
* TEST: according to the random number generate by the PREPARE state, give out the hint signal, wait for user pressing the test button.
  + Play the audio sound hint and/or lighten the LEDs hint after the random number time given by PREPARE state.
  + Test button pressed:
    - If the test button pressed before the hint occurs or not being pressed in 10 seconds, the result is FAIL and move to RESULT state.
    - If the test button pressed in 10 seconds, the result would be recorded and move to RESULT state.
* RESULT: show the test result of the TEST state.
  + After 10 seconds, go to the IDLE state.

If reset signal is on, the state would be back to IDLE state. A 2-bit register is used to store the current state. This register is checked at each 100MHz system clock rising edge by the FSM logic unit. The FSM logic unit is implemented as an *always* block. Within this block, when the state is not changed, the output signal would be refreshed from the output of the state core. If there is no output for one type signal, for instance, there is no audio output for IDLE state, the audio output data would be reset to default value.

### IDLE state core

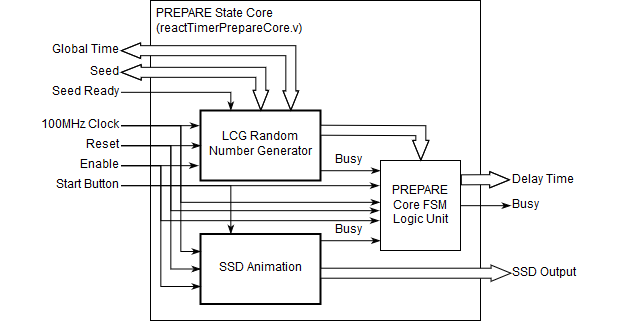
The logic of this core is pretty simple. It stores the binary of the best time. The test result comes directly from the TEST state core. There is also an LED animation playing module embedded in this core.

The outputs of this core are LED animation result and SSD signals.

### PREPARE state core

PREPARE state core implements in a small two state FSM. One state is idle and the other one is busy. For the idle state, this core just waits and detects the start signal rising. The busy state is waiting for the random number generator to generate the random number and SSD animation module to finish playing the countdown animation.

The following diagram shows the PREPARE core architecture.



Once the start button is at rising edge, PREPARE core FSM logic unit would change the state from idle to busy. LCG random number generator and SSD animation module would start to work concurrently. The LCG and SSD animation module both have a busy wire output which shows whether their mission is finished. Once all of these wires changed to 0, this module would change the busy output from 1 to 0.

The output of SSD animation module directly connects to the output of PREPARE core. The busy state of LCG generator is checked by the FSM Logic Unit at every 100MHz clock cycle rising edge. Once the falling edge of the LCG wire is detected, the output 32-bit random number would be modular within 700,000,000 that could guarantee that the waiting seconds are limited to maximum 7 seconds.

#### Random Number Generator

In M7T3, it contains two random number generator: Liner Congruential Generator and Mersenne Twister MT19937 standard implementation. All the pseudorandom number generators used in **M7T3** needs 32-bit seeds and could generate 32-bit output. The time deadline for generating a random number is 4 seconds because the animation duration of the countdown is 4 seconds. Even if the random number generator cannot meet this requirement, it is still okay to be used. However, the duration of the PREPARE state would be longer and this is transparent to tester.

These two pseudorandom number generators are managed under an Advanced Elective Generator Integrated System (AEGIS) unit. This system is embedded in the always block of the FSM logic unit in the PREPARE state core. All these generators are using the same port interfaces, which allows us to add and remove pseudorandom number generators easily on source code level. When output the random number result, AEGIS system would limit the output number less than 805,306,366 (32’h2FFFFFFE, calculated by ), which is around 8 seconds. This allows the tester won’t wait too long.

LCG generator is used as the random generator. The equation used in this implementation is , which is the same as the LCG equation used in Microsoft Visual C/C++ standard library. This equation could be completed by Artix-7 within 20ns, hence two system 100MHz clock cycles to meet the time requirements. According to the report, in **M7T3** implementation, it needs 10.568ns to complete the multiply and addition at the same time (7.336ns for logic and 3.232ns for route). So in the implement, this calculation has been separate into 2 stages (modular is implemented by using 32-bit register). The seed is chosen using the global timer counter for each time running the generator.

Mersenne Twister is a pseudorandom number generator widely used in multiple language compilers. It was introduced by Dr. Matsumoto and Dr. Nishimura in 1997. The original paper could be found at <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf>. It has a standard implementation called MT19937. The Mersenne Twister module of our implementation in **M7T3** is compatible with MT19937 standard.

To implement a Mersenne Twister generator, the following parameters are needed in the following description

* – the length of the vector. This number is 624 in **M7T3** implementation.
* – the bits of the final output random number generator. This number is 32 in **M7T3** implementation.
* – an -length long row vector construct with 0 and 1. Hence a 32-bit unsigned binary.
* – the number a bit from the rightmost of .
* – integer in range .
* – integer in range .
* – a matrix.

* – a concatenation of higher bits of and filling 0 with lower bits. In Verilog, it could be expressed as {x[w:r], r{0}}.
* – a concatenation of lower bits and filling 0 with higher bits. In Verilog, it could be expressed as {(w-r){0}, x[r:0]}.
* – a concatenation with as higher bits and as lower bits.

To initial the MT19937 generator, first prepare a -length long row vector with the following equations:

According to the paper, the result of is

In the implementation, this is exactly using the Linear Feedback Shifting Register (LFSR) as we mentioned in the assignment paper. The description above is actually a level LFSR: take the higher bits of the  and the lower bits of the , then goes through a linear transformation . However, the equation above is actually take rounds of LFSR. It won’t affect the LFSR period if  and are relatively prime.

With this equation, this allows us to extend the period to , which is reason of the name contains 19937. To extract the value from the result of the data, we only need to post multiply an invertible matrix. With integer parameters (, , and ) and two -width bit mask integers ( and ), this multiplication could be expressed by

And the result is the output of one round. Summarized the MT19937 implementation, there are three states: seed initialization, twister and extraction.

In M7T2, we tried to implement this MT19937 with a multiple level FSM. However, it would take 20,118 look up tables and 21,276 flip-flops which takes a very long time to synthesis and implementation (and even gets an unknown error from Vivado). So a new pipelined optimization has been introduced to **M7T3** implementation. With this implementation, the number of look up tables has been reduced to only 304. Flip-flops usages has been reduced to 165, which makes MT19937 could be fit in the implementation.

In the pipeline implementation, there is still one FSM has been used to distinguish the idle and busy state. In the busy state, it would go through the entire three states of the MT19937 standard and update all the values of the 624 vectors to the final result. This would take 624 clock cycles to complete the entire computation. Then update all the values every 624 times access for asking the next random number.

In each clock cycle of busy state (or the SEED state in the implementation), it would complete one round calculation within 10ns (100MHz system clock period) and meet the time constraints without providing a negative WNS.

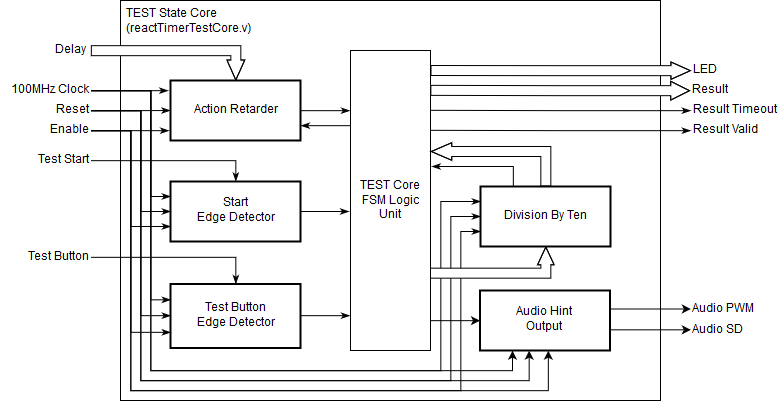
#### SSD Animation

The animation of SSD is implemented by an increase counter which increase 1 at when the timer clock rising. The clock is set to be 1Hz and increase the frame index and update the output frame according to the frame index at the rising edge of system 100MHz clock. The frames are stored as a 32-bit integer array inside the module.

### TEST state core

After the PREPARE core provides the delay duration, the TEST state core start working. This core is responsible for showing the hint after a specific period of time, this is implemented by an action retarder. Then it would measure the gap of the hint provided to the test button pressed. Last, it calculates the data to be output.

The following diagram shows the inner structures of the TEST state core.



The inner framework of the TEST state core is a FSM as well. There are 3 states for this core:

* Idle state: waiting for the test start signal. When the start signal pulse comes, the state switch to wait state. At the same time, the FSM Logic Unit would give action retarder a pulse. The action retarder would give another pulse after the time provided by the PREPARE state core.
* Wait state: At this state, the action retarder would hold the pulse for a while, and output the pulse after several clock cycles, then move to test state. FSM Unit detects this pulse for each 100MHz clock cycle rising edge, and then move to test state. If tester hits the test button at this state, this test would be marked as ‘FAIL’ and move to finish state.
* Test state: According to the configuration, the LED would be shown and/or the audio would be played. At the same time, counter starts to count from 0 and increases 1 for each 100MHz clock cycle. Once the user hits the test button, the counter would stop to count and then move to the finish state. If the counter reaches the limitation (by the default, is 999,999,999, which is actually 9.999 seconds), this test would also be marked as ‘FAIL’, then move to finish state.
* Finish state: the result of the counter is a 9-digit integer in decimal. The number goes into a division module which could divide a number by 10. If the result is not ‘FAIL’, it would wait until the divider finish and then mark the result as valid. Or else set the result to be 0, set the timeout signal as 1, then mark the result as valid.

Action retarder is just a 2-state FSM which detect the input pulse and then wait by increasing a counter as it reaches the number provided by the TEST state core FSM logic unit. It would ignore all the other pulses while holding the first signal it detects.

#### Division by Ten Module

The precise of the counter is 10ns. However, we have only 8 digits which allows us to display the number at 100ns. So we have to divide the counter result by 10. We originally use counter / 10 here to calculate this value. However, it fails to reach the time constraints (provides a negative WNS). Instead of using the division calculation directly, we are using an 8 clock cycles delay divider to do this calculation.

In this algorithm, we need two parameters: and . Suppose the number we want to divide by 10 is . First step, we can get () by

next, get () by

after that, get () by

then, get () by

afterwards, get () by

This number is actually very closed to the data as what we want. To make the data more precise, we need to calculate and check it is greater than 9 or not by

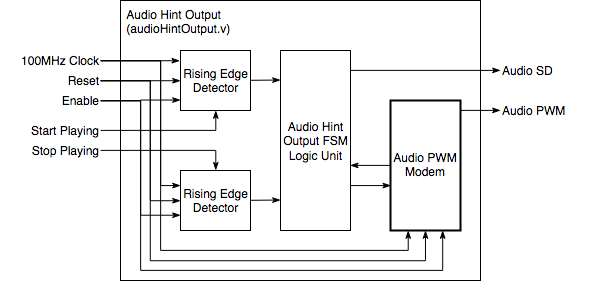
and finally get the result

which is correct in the 32-bit integer value for division by 10. This module is implemented by a 9-state FSM, which means that it adds a 9-cycle delay to the system at 100MHz.

In the implementation, there is a busy wire which marked whether the output data is ready. The TEST core unit would detect the falling edge of the divider. Once it gets the result, it will move on to the next state.

#### Audio Hint Module

The following diagram shows the architecture of the audio output module.



Inside this module, it stores the hardcoded audio samples. The PWM modem would provide a rising edge to update for the sample. The whole module is implemented by a 2-state FSM: idle state and playing state. When the start playing input wire occurs a rising edge, the state machine would move from idle to playing state, and sending audio samples to audio PWM modem. The state would be kept until the stop playing signal occurs a rising edge.

For the PWM modem using in the implementation, we decided to use 8-bit output for the sample. In the original plan, this should be 16-bit. However, the 100MHz system clock would limit the signal. To express a 16-bit number using PWM, we need 65,536 clock cycle for one number. As the system could provide a 100MHz clock, it allows us to express 1,525 samples only. To express a sine wave, we need at least 4 samples to express a cycle (which is 0, 1, 0 and -1), hence the limitation of the 16-bit PWM module with 100MHz as the input clock could only generate a 381Hz sine wave. However, the Stuttgart pitch (A440) is 440Hz, this could even fail to meet this requirement. If we are using an 8-bit PWM module with 100MHz system clock, it allows 390,625 samples per second, and this limitation would be 97,656Hz for the 4-sample sine wave. It allows use to generate A11 which is a 56,320Hz sine wave. Hence, the resolution for the PWM is decided as 8-bit.

The PWM modem is implemented by an 8-bit counter, which counts from 0 to 255 and then warp back to 0. The PWM output is a simple comparator of the counter and the sample number. When this counter start counting, it would ignore the change of the input during the counter count from 0 to 255. It would update the current sample at each time the counter become 0. It would give signal back to FSM logic unit to switch to next sample as a rising edge when the counter reach 128.

In **M7T3**, it uses a counter to control the frequency of the output signal. There is another counter which counts from 0 to a pre-set limitation parameter. This counter is called as frequency counter. Once the frequency counter reaches the limitation, the signal counter increase 1. For example, when the limitations set as 1, the signal counter increase 1 only when the frequency counter reach 1. The frequency of output PWM has been reduced to its half.

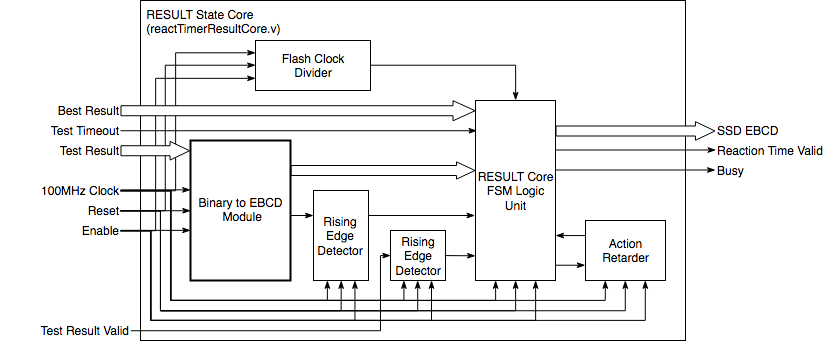
According to the Nexys4 DDR reference manual, the PWM is using the time of 1 divided by the entire period the get the value. As the document mentioned, we should output the data as PWM to express the data as what we want. However, we output the sine wave data to the audio pin, the sound is very strange. We finally choose to output a constant value which is 8’h80 which is exactly half of the duty cycle. And performs very well. Using Moku, we could actually find out that the PWM wave output directly without transfer the PWM to analog signal as the DAC module output. This is one of the thing that not mentioned in the reference manual. **M7T3** could output sine wave, but for better usage, we decide to output square wave.

Another thing which is not mentioned in the reference manual, is the existence of the AUD\_SD pin. In section 16, i.e. Mono Audio Output, it only mentioned about the PWM pin (A11), but not mentioned the AUD\_SD pin. AUD\_SD connects to the FPGA pin D12. This pin connects to the amplifiers of the low pass filter in the Figure 29 of the reference manual (Sollen-Key Butterworth Low-Pass 4th Order Filter), which controls the enable of the amplifiers. Hence, AUD\_SD is actually controls the enable of the low pass filter. In **M7T3** implementation, the AUD\_SD pin is hard coded to output high which means that it is always enable for output data.

### RESULT state core

RESULT state is implemented as a FSM. It contains 3 states: idle state, convert state and wait state. It starts at idle state, waiting for the test result from TEST state available. Then, RESULT state core would transfer the reaction time result from binary number into EBCD, and then according to the result recorded from the IDLE core, check whether the current result is the best or not. This stage is called convert state. After that, it switches to wait state. If the result is the best result, the result would be flashed for 10 seconds, or just display the time or ‘FAIL’ for 10 seconds. Then back to idle state.

The framework of the RESULT state core is shown in the following figure.



A clock divider is introduced to be used as the flash frequency. In **M7T3**, this clock has been set to be 1Hz. The SSD output signal is updated based on the divided clock from the clock divider at each rising edge of the 100MHz clock.

A busy wire is used as the signal for the core is at none-idle state. **M7T3** FSM logic unit checks this signal at every 100MHz rising clock. When the falling edge of busy is detected, **M7T3** state goes back to IDLE state.

An action retarder is introduced with a fixed parameter for waiting 10 seconds. It would start working right after the Binary to EBCD module finishes its work. Once the delayed signal pulse is detected by the FSM logic unit, the busy output would change from 1 to 0.

#### Binary to EBCD

To display the number on SSD, we have to separate the number from binary to each digit of decimal. Generally, we could divided by and modular by 10 to get the th digit of the number. However, this would give a very bad performance on WNS using division. Hence, we need another method to get each digit.

First, we need a way to express each digit of the number. There are 10 numbers of each decimal digit (0 to 9), which needs for 4 bits to store all of them. This is called Binary Coded Decimal (BCD), or actually 8421 coding. In **M7T3**, we are using an enhanced version called Enhanced Binary Coded Decimal (EBCD) which is compatible with BCD. The method of transferring binary to BCD could be used in our design.

The method we are using is called Shift-Add Binary to BCD transfer. In **M7T3**, we will finally display an 8-digit decimal. Hence, we need 32 bits to store the BCD result. The algorithm can be display as the following

1. Prepare two shift registers. One 28-bit for storing the reset of the original binary bits. The other one 32-bit for storing the transferred BCD result. Initial the 28-bit register as the original data. Initial the 32-bit register with a 32-bit decimal 0 (32’d0). Initial a counter to 28.
2. Check each decimal half-byte of the 32-bit register, if all the digit is greater than 4, increase this half-byte by 3.
3. Left shift the 32-bit register, and append the most-significant bit of the 28-bit register to the right-most bit. Left shift the 28-bit register. Decrease the counter by 1. If the counter is 0 now (left shift for 28 times), the transfer is finished, or else go to step 2.

Here is an example of transferring decimal 255 (binary 0000,0000,0000,0000,0000,1111,1111) to 8421 coding BCD with this algorithm. For the first 20 bits, there are no different because they are all 0. The diagram shows from step of 20 to 28. Step dot 2 means Shift operation from 28-bit register to 32-bit register and step dots 3 means the Add 3 operation for checking each decimal digit.

|  |  |  |
| --- | --- | --- |
| Steps | 28-bit register | 32-bit register |
| 0 | 0000 …… 0000 0000 1111 1111 | 0000 0000 0000 0000 0000 0000 0000 0000 |
| … | …… | …… |
| 20 | 1111 1111 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0000 0000 |
| 21 | 1111 1110 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0000 0001 |
| 22 | 1111 1100 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0000 0011 |
| 23.2 | 1111 1000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0000 0111 |
| 23.3 | 1111 1000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0000 1010 |
| 24.2 | 1111 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0001 0101 |
| 24.3 | 1111 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0001 1000 |
| 25 | 1110 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0011 0001 |
| 26.2 | 1100 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 0110 0011 |
| 26.3 | 1100 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0000 1001 0011 |
| 27.2 | 1000 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0001 0010 0111 |
| 27.3 | 1000 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0001 0010 1010 |
| 28 | 0000 0000 0000 …… 0000 0000 | 0000 0000 0000 0000 0000 0010 0101 0101 |
| Result | 0000 0000 0000 …… 0000 0255 | 0000 0000 0000 0000 0000 0002 0005 0005 |

The reason for shift 28 times is we have to shift all the bits from the raw data to BCD expression. Using left shift and take the MSB would be simple for the implementation by taking the bit and shift for one bit.

To calculate the BCD of the binary, we first consider one decimal digit case. For 4-bit binary, it would increase 1 to the higher 4 bits when it reaches 16. For 4-bit 8421 coding BCD, it would increase 1 when it reaches 9. Hence, to transfer 4-bit binary to 4-bit 8412 coding could be processed by plus 6 to the binary.

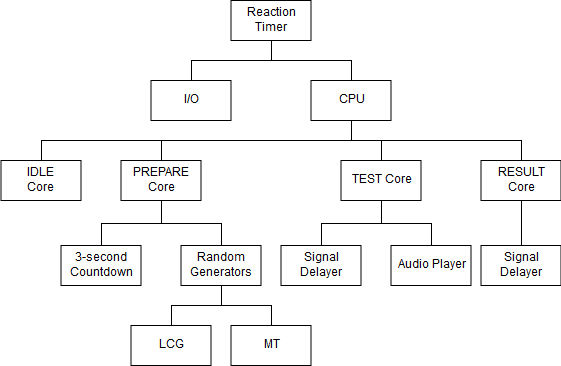
Then we consider the way to transfer multiple digit decimal into BCD. For example, decimal 30 (in binary 11110). It could be get by left shifting decimal 15 (in binary 1111). The 8421 coding BCD of 15 is 0001 0101. Shifting left 1 bit we could get 0010 1010. Check each half byte data of the BCD expression, 1010 is invalid, so we increase it for 9, the BCD is now becoming 0011 0000, which is exactly 30.

However, using increase 9 would affected the entire number. Instead of calculate after shift, we could actually check the BCD 8421 coding before we shift the bits. To increase one in binary, it needs the number to be 16. However, in 8421 coding BCD, this number should be 10. Left shift means multiply the number by 2. When the BCD digit is greater than 4, for example 5, shift left would let it to be 10, and need to increase 1 in the higher 4-bits and reset the current 4-bit to 0, and express this action is actually binary 1,0000, which is 16. In other words, the 5 here is actually 8 as what we want for the shift. Hence, before we shift the number, we check each digit is greater than 4 or not. If so, plus 3 to make this number could increase 1 in the higher 4 bits. Then do the shift and append the new bit. And this is the method that shown before and used in **M7T3**.

This algorithm needs 28 rounds for one transfer. Each round has 2 stages. Hence, 56 clock cycles to calculate the BCD from a 28-bit binary. The module is implemented by a 3-state FSM. A busy wire is present as an output for showing the signal of whether the transfer is completed. The FSM logic unit of the RESULT state core is going to check this wire for each rising edge of the 100MHz clock cycle.

# Design Verification and Validation

During the development, we are using test-driven development and go through the entire development period. We first determined the high level design. Next, we separated the function of each state, and drew the top level design diagram. The right diagram is the one that we draw before we started our implementation.



Then, we wrote each leaf-level modules, and write a test bunch for each of them as unit test. The All the unit tests are using strong robust equivalence classes black box testing. During this stage, we found the following two critical problems when we write the simulation test bunch for unit testing:

The first problem is some modules needs to execute very long time if we are using the real clock settings in the simulation. For example, the countdown animation module using in the PREPARE state core need a simulation of at least 4 seconds. Doing 1 second simulation would take a very long time on a Intel Core i7-3770 processor (4 cores 8 threads at 3.7GHz). To solve this problem, we introduced the dynamic parameter for each module. All these parameters are providing a default value which would be used as implementation configuration. But inside the test bunch, we are using a different parameter which allows us to do the simulation in a different condition, but see how the module works. This method is widely deployed in **M7T3** implementation.

The other problem is simulation cannot simulate the time constraints of the real board. In the simulation, if we use division and modular, it would still provide the same result and correct output. However, division cannot reach the requirement of the Nexys4-DDR board constraints. For 10ns period of time, it cannot complete these calculations. This data would only be available when the implementation stage completed.

The Worst Negative Slack (WNS) and Total Negative Slack (TNS) show whether our design is valid to the board and meet the time constraints. According to Lyle’s explanation, we found the information within the timing summary log in the reports (impl\_1\_route\_report\_timing\_summary\_0). Inside the report, it has a section named “Design Timing Summary”. If there is a negative WNS had been found in our design, it will list it here. In our design, the following parts are mention failed to meet the time constraints:

1. Separate the binary number into decimal digits with division and modular directly. (Solved by replacing the original method with Binary to EBCD module)
2. LCG result calculation. (Solved by introducing a FSM and separate the multiplication and addition into two steps)
3. Test result divided by 10. (Solved by introducing a new module which is especially designed to divide a number by 10)

Now, all the module designs specified timing constraints are met according to the implementation reports. The maximum delay path appears at the multiplication in LCG module. The data path delay is 9.579ns, which is very closed to the 10ns requirement. The final design takes 1,059 look up tables and 1,047 flip-flops after implementation optimization.

In the design, most of the function has been packaged into a single module. Some of the small common code pieces are gathered into a task. This is the light-weight cohesion code solution, which allows us to improve the robust of our implementation.