EDA Assignment

Decimal Clock

Tony de Souza-Daw
Wade Tregaskis

20/9/2004

Problem

The problem put forward for the assignment was to create a digital decimal clock, with the typical functionality of any digital clock – i.e. standard display of the current time, the ability to be set the time, etc.

A decimal clock is one where there are 100 decimal seconds per decimal minute, 100 decimal minutes per decimal hour, and 10 decimal hours per day. Thus, 1 decimal second is equivalent to 0.864 normal seconds.

The parameters of the solution are defined by the Altera UP2 board used to implement the solution. Namely, there are a finite number of devices available for input and output - two push-buttons, 8 DIP switches, and 8 7-segement LED displays. The FLEX chip also has finite programming capacity, although it is accepted that this capacity is far beyond that necessary to solve this problem.

Solution

Approach

As we were inexperienced, when starting the assignment, with the software involved, we elected to attempt the design & implementation in FPGA Advantage. As such, we made use of its block diagram functionality to design top-down, then implement bottom-up. The intention was to utilise the top down approach for design to modularise the solution in an intuitive fashion. This design would then be implemented bottom up to allow for module testing and design verification, prior to implementing and integrating the complete system.

As it eventually conspired, FPGA Advantage proved too buggy and to have too many undocumented nuances to provide a suitable implementation platform. As such, a transition to Max+PLUS II was made in order to implement the design. While Max+PLUS II was itself unsuitable for the implementation, it was all that was available and did, in the end, suffice, albeit with much sweat, blood and tears.

Design

Specification

The first priority was to define how the solution should operate. It was immediately clear that the primary interface should be the two push-buttons and the 6 grouped 7-segment displays. This interface is comparable to that of a typical digital watch. While the option was available to use the 8 DIP switches for various core functionality, it was felt that this was a less elegant solution. As such, the DIP switches were relegated to debugging tasks only.

The behaviour was fairly well defined by existing digital clocks – the time would be displayed on the 7-segment displays, incrementing as appropriate. ‘10’ would be displayed as the zero-hour, much the same way 12 is displayed instead of 0 by most clocks. When in set mode, the clock would stop incrementing automatically, and the currently selected section (hours, minutes or seconds) highlighted in some fashion.

The two buttons would be defined as “mode” and “advance”, in loose terms. The ‘mode’ button would swap between normal operation and set mode, if held down for some appropriate period (e.g. 2 seconds). Furthermore, in ‘set’ mode the ‘mode’ button would, if pressed only briefly, cycle through the selected fields of the display. Finally, the ‘advance’ button would (in set mode) increment the selected field). In ‘normal’ mode it has no functionality.

Overall

The highest-level of the design can be divided into two sections – the major section being the decimal clock itself, the minor being the student number display for authenticity purposes. These two sections are entirely independent of each other. See Appendix I for a diagram.

Student Number Display

The functionality required is trivial – a DIP switch should toggle between the two numbers displayed, which need to be shown on two 7-segment displays. The 7-segment decoder would of course be implemented as a component, so it could be reused throughout the solution.

Decimal Clock

The first concern in this design, as with most designs, was in providing the necessary clocking signals for operation of each module. The clock available was a 25.175MHz crystal, providing far faster operation, and higher resolution, than the solution required. As such, it was considered a good idea to divide it down where possible, to save logic space in terms of counters and other processes.

Consequently, a clock divider module was designed to take as input the 25.175MHz system clock, and output clock signals for incrementing the clock and for the display module. It would also take as input two of the DIP switches, allowing the exact division ratio to be chosen from amongst 4 values, easing debugging and demonstration of the circuit by allowing it to run up to a thousand times faster than normal.

Next, the two button inputs needed to be debounced to prevent inappropriate behaviour. As such, a debouncer module was defined to take as its inputs merely the signal to debounce and the system clock, and to output a clean version of the input signal.

The actual clock module itself was then defined, given it now had the appropriate inputs from the clock divider and debouncers. While the clock module could, in hindsight, have been broken up into several modules, due to the implementation issues already glanced upon it was ultimately far easier to implement as a single module.

The inputs for the clock were straight-forward – the two debounced button signals and the appropriate clock from the clock divider (plus a synchronous reset signal). As it transpired the system clock was also needed, and added during the implementation phase.

The outputs from the clock module were similarly intuitive – the clock values, in hours, minutes & seconds, as well as enables for the three sets of values (to allow for flashing during set mode).

Finally, the display driver module was defined. It's purpose would be to take the values from the clock and display them on the six 7-segment displays. This task is complicated slightly by the fact that these six displays operate from a shared bus, and so cannot be hard-wired to individual decoders. The display driver thus has to operate a simple finite-state machine to output each 7-segment display one after the other, with appropriate strobes and enables to control the display latches.

Consequently, the display driver's inputs would be the values from the clock as well as a clocking signal from the clock divider, which would govern how fast the finite state machine would operate (as an aside, in the current implementation this is at full system clock speed, but could be slowed to allow for slower 7-segment display latches or other timing issues).

The outputs would be the various enable & strobe lines, along with a connection to the data bus.

Display Driver

The display driver's primary purpose is to multiplex the time values from the clock module, and output those in an appropriate fashion to the six 7-segment displays on the Altera expansion board. It does this using a simple finite-state machine, where it places the current digit onto it's outputs, along with the appropriate enable/strobe signals. The output goes via a segment decoder to the data bus.

The primary issue for the display driver is the timing issues in driving the various latches for the displays. This issue has two key elements:

1. What speed the latches can be driven at (i.e. how fast the FSM can change states and how fast the multiplexer can operate).
2. How long the data must be on the bus for it to be latched reliably (i.e. the setup & hold times)

Testing later showed that the latches were more than capable of operating reliably at the system clock speed (25.175MHz), and the setup & hold times need only be a single cycle each.

Refer to Appendix II for a diagram.

Clock

This module is the most complicated in the design, and was further complicated during implementation due to synthesis bugs and inadequacies in Max+PLUS II. Essentially the clock module needs to operate as a fairly complicated finite state machine. At the highest level it is in either 'normal' or 'set' mode. In normal mode it needs to increment the seconds at appropriate intervals, and roll over any overflow into successive digits. In set mode it needs to trigger off the advance button to increment the selected field, but *not* roll over to other fields. It also needs to cycle through the three fields when mode is pushed briefly.

In either mode it needs to detect that the 'mode' button is held down for some extended period (e.g. 2 seconds) and toggle modes.

As the design in this particular case was driven primarily by implementation constraints, the details are covered in the implementation section. Worth noting here, however, is alternative ways this module could have been designed, with the benefit of hindsight.

First and foremost, the incrementing of the clock values could have been delegated to a sub-module, which would take as it's inputs an increment signal and an overflow control, and outputs as the current value and an overflow indicator. The value would be incremented once for every edge transition (high or low chosen based on connecting logic), and the overflow control would indicate whether the overflow indicator should be used. The overflow indicator for the seconds, for example, could then be connected to the minutes to automatically increment this in the event of an overflow, but only when not in set mode (i.e. the overflow control is enabled).

The driver for the increment would then simply be a combination of the increment button, the increment clock, and the current mode, to select one or the other. The driver for the overflow control would simply be the current mode.

This design is much more elegant than that implemented, but of course hindsight is often 20/20.

Implementation

Student Number Display

The implementation here was trivial - an effective "if…else" statement selects one of the two numbers to be displayed, which are then passed to two 7-segment decoders, to be displayed on the appropriate 7-segment displays.

Segment Decoder

The segment decoder was implemented as a simple asynchronous decoder. It supports all hexadecimal values, even though we only use the decimal subset – the argument for this is slim, but goes something along the lines that it would be handy to have the hexadecimal option available for debugging purposes, or other future modifications.

Clock

The clock module was a nightmare to implement. In our naivety we assumed the Max+PLUS II synthesiser would be at least somewhat intelligent, but this is clearly not the case. It incorrectly identified multiple driver issues, performed dead-code stripping on live code, and exhibited a variety of other bugs. As such, the implementation of the clock module went through many revisions, and at some points became hideously convoluted. Believe it or not, the final revision is many times more readable than many which came before it.

Essentially, the final result was to have a primary process for performing the majority of the work, a process to monitor the 'mode' button input and change modes or field selection, and a flasher process to strobe the appropriate 'show' outputs for the selected field, if in set mode.

The latter two processes are fairly straight-forward. The modeCounter process simply counts how long the 'mode' button is pressed. If it passes a certain threshold it toggles the mode. If the button is released prior to this threshold being achieved, and the current mode is 'set', it moves the selection to the next field.

The flasher process simply toggles the selected field's show line, if in set mode, at regular intervals. An important observation is that this could be driven by a special input clocking signal from the clock divider module, and this would be a more logical design, but Max+PLUS II would not synthesise this code unless it ran on the system clock.

The third process, the clocker, simply increments the clock as appropriate (whether based on the 'advance' button being pressed in 'set' mode, or base on the increment clock). As an additional feature it also responds to the 'advance' button being held down for an extended period – initially it increments once, but then if the button is held down for two seconds it starts incrementing continuously at a rate of approximately 8Hz. This is identical to typical digital clock behaviour, and consequently intuitive for the average user.

As an aside, the original implementation had half a dozen processes, each with a well-defined purpose and contained scope. Because Max+PLUS II could not correctly identify mutual exclusions correctly, or perhaps not use tri-state logic for multiple drivers, this implementation failed.

Segment Multiplexer

As mentioned, this operates as a simple FSM – it has essentially 6 primary states, one for each of the six 7-segment displays, with each state broken into 2 mini-states. The first of these mini-states puts the appropriate data onto the bus, while the second maintains that data as well as toggling the strobe for the appropriate latch. Ignoring the [hopefully] small time delays, this places the strobe in the centre of the data on the bus, minimising the chance of the latch seeing changing data at the transition points.

Clock Divider

This very simply runs a counter in a loop, toggling the output clock signals at appropriate intervals. While its implementation in this particular case only does this for one output, effectively, the reason for having a distinct clock divider is that when more outputs are added they can in many cases be run off the same counter, minimising the logic required for the division.

Debouncer

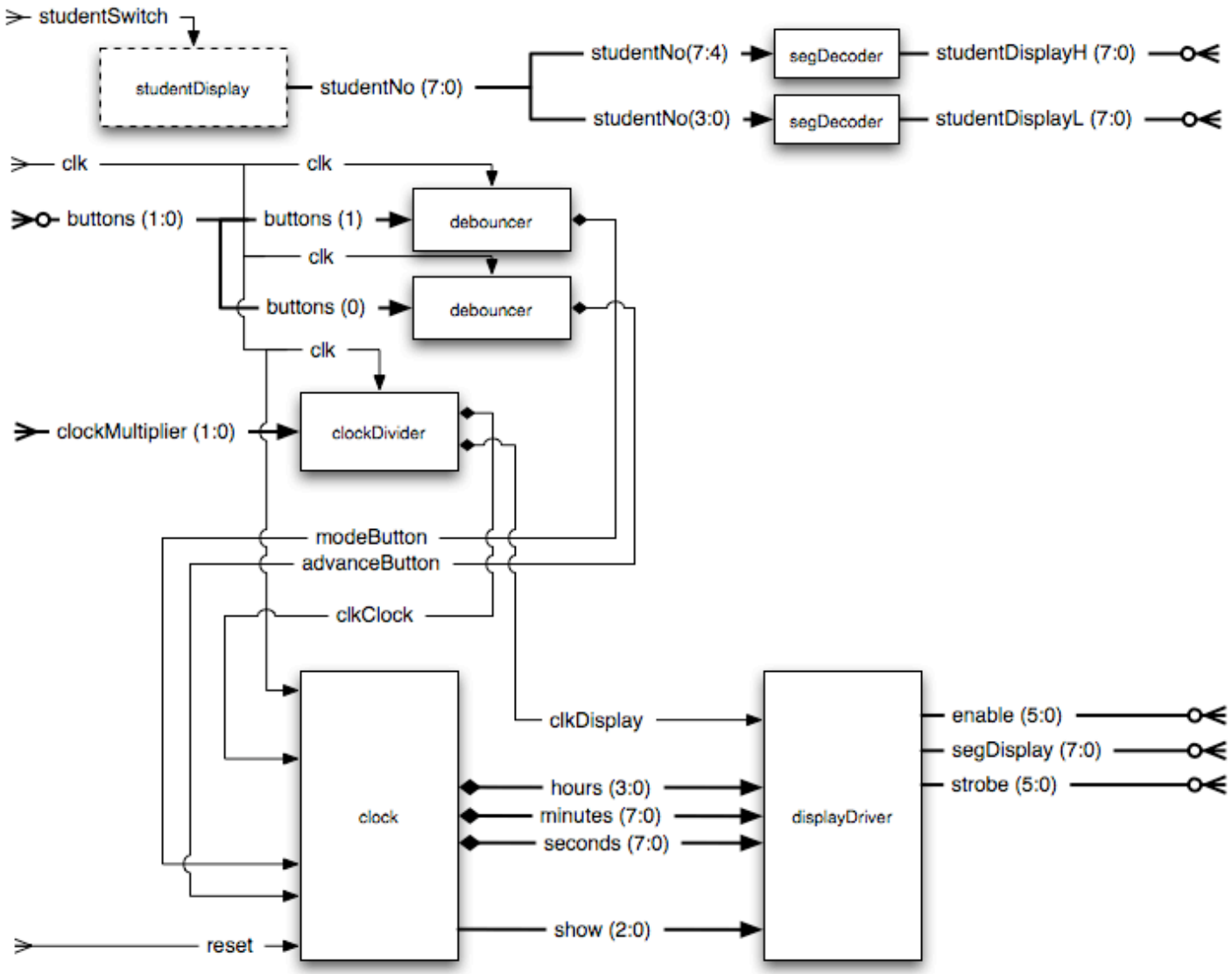
The debounce operates very simply – only when the input signal is in the same state for a certain number of clock cycles does that input get carried to the output. This works independent of edge direction, so it works equally well in debouncing button release and button press. In this particular implementation the hold period is 25ms, which we have seen in our CDP labs is more than sufficient for most types of push-button input.

Conclusion

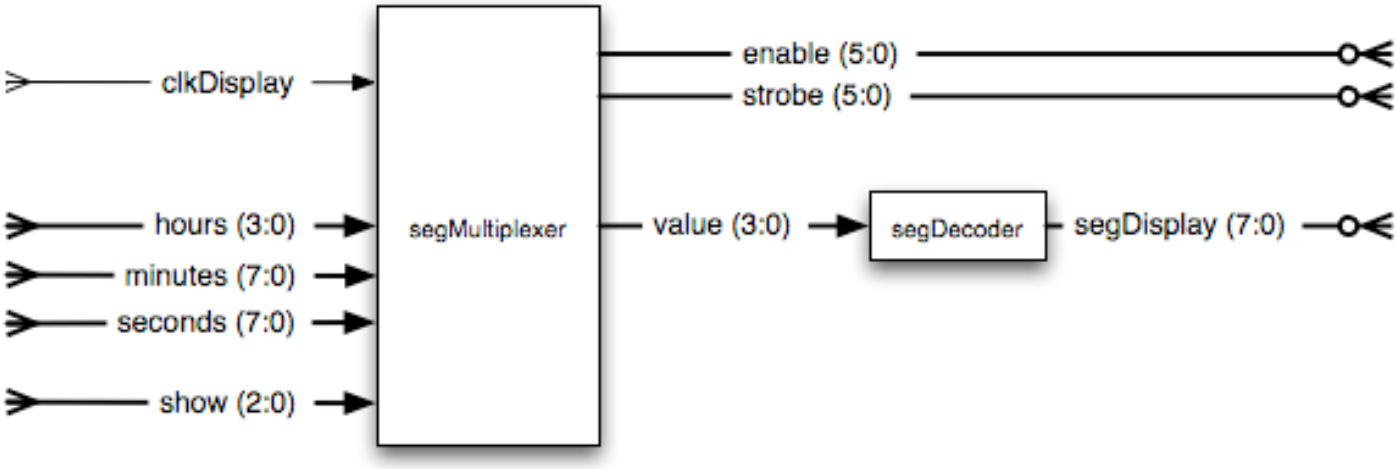
Despite the issues we had with the software, we did produce a 'perfect' product, in terms of practical operation. While the design may be somewhat convoluted, we feel it is nonetheless reasonable, and a rather good first attempt. Given that our EDA Project is very similar (a clock/stopwatch/timer/alarm/etc), we felt it better to study our first attempt (this assignment) objectively and honestly, as opposed to dressing it up, fixing the faults, and pretending we got everything right the first time through.

All in all the assignment was a good learning experience. While we are familiar with CDP and most of the methods we utilised in this assignment, this is by far the largest system we have designed to date (in VHDL), and consequently was a valuable introduction to the various design methods adopted (such as component-based and hierarchal design). And, thinking optimistically, we are now much more aware of what Max+PLUS II & FPGA Advantage can and cannot do. This knowledge will undoubtedly save us much pain in future designs.

Appendix I


studentSwitch
studentDisplay
studentNo (7:0)
studentNo(7:4)
studentNo(3:0)
segDecoder
segDecoder
studentDisplayH (7:0)
studentDisplayL (7:0)
clk
clk
buttons (1:0)
buttons (1)
debouncer
clk
buttons (0)
debouncer
clk
clockMultiplier (1:0)
clockDivider
modeButton
advanceButton
clkClock
clkDisplay
clock
hours (3:0)
minutes (7:0)
seconds (7:0)
show (2:0)
reset
displayDriver
enable (5:0)
segDisplay (7:0)
strobe (5:0)

Appendix II


clkDisplay
hours (3:0)
minutes (7:0)
seconds (7:0)
show (2:0)
segMultiplexer
enable (5:0)
strobe (5:0)
value (3:0)
segDecoder
segDisplay (7:0)