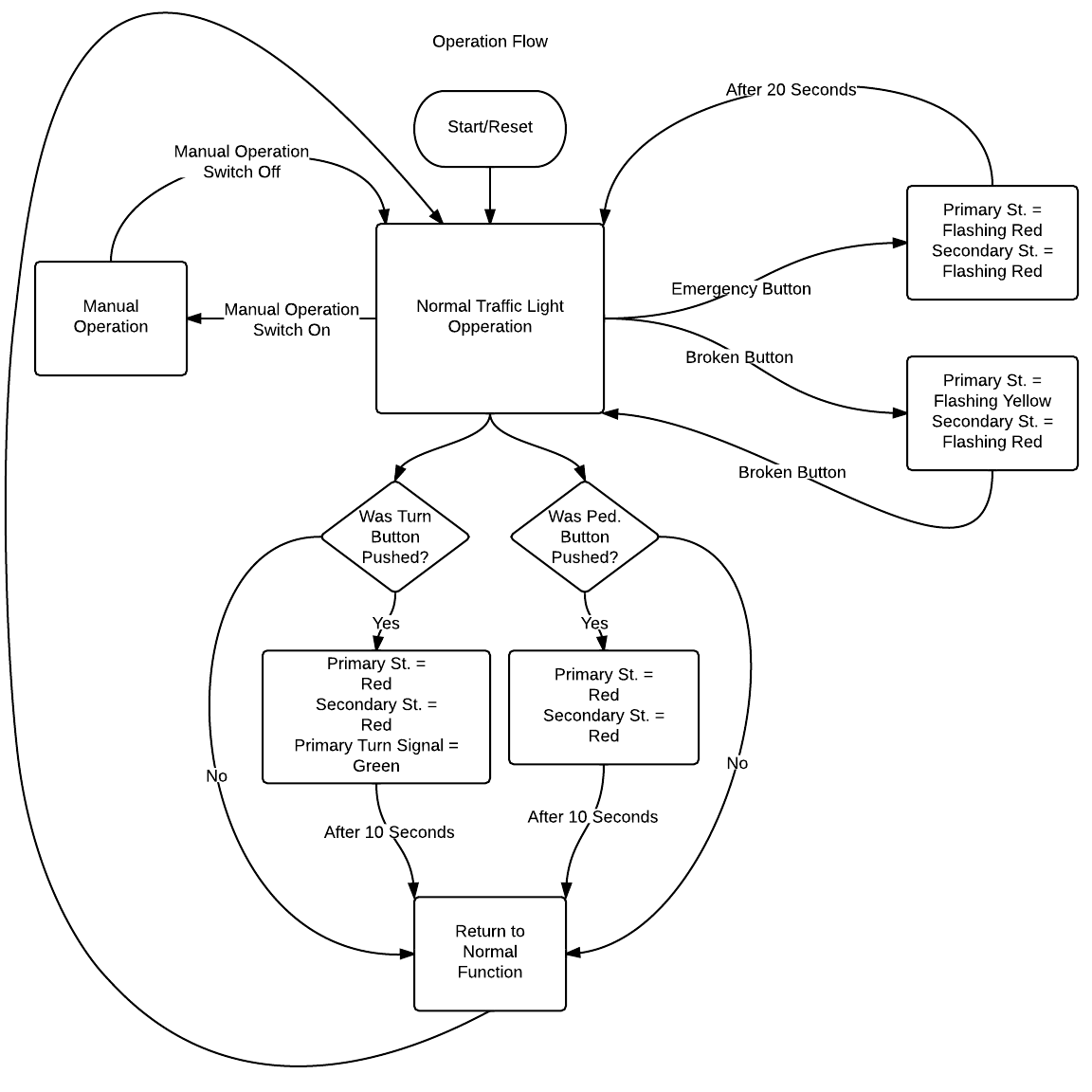
**Embedded Systems Project 4:**

**Traffic Light Controller**

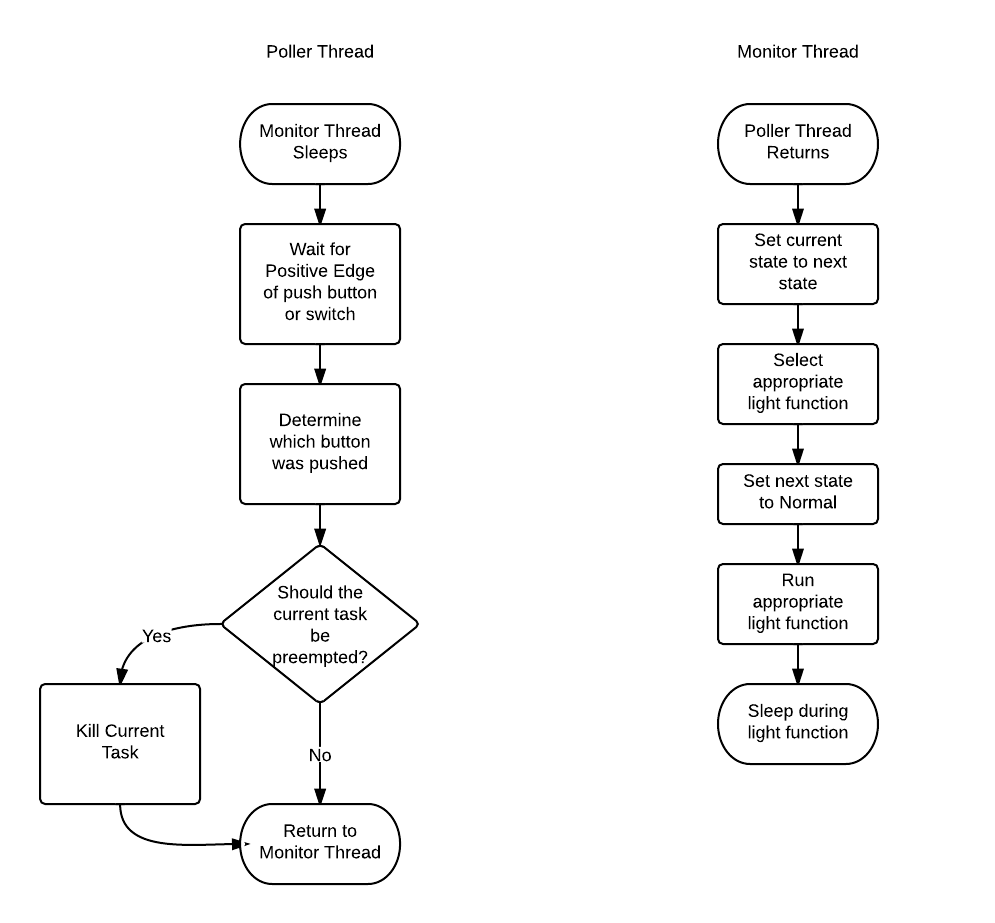
For this project we implemented a traffic light controller using an Altera FPGA that contained an embedded NIOS II processor. The traffic light has six different modes of operation that it can be in based upon the current state of the push-buttons and switches. The different modes of operation are each given a priority with which they will be allowed to run within the system. The highest priority, which will trump any of the other modes, is the “Broken” mode. When the traffic light is executing in this mode the green LED’s representing the yellow light on the main road and the red light on the secondary road flash every second, and one of the red LED’s lights up corresponding to the current mode of operation. The second highest priority mode is the “Emergency” mode. In this mode the green LED’s representing the red lights on both roads flash every second, and one of the red LED’s lights up corresponding to the current mode of operation. The third highest priority mode is “Manual” mode, which allows a user to control the traffic lights using a set of switches located on the board. The first switch turns the system into manual mode and the second switch controls which street will have the green light and which will have the red light (after transitioning through the yellow light of course).

The fourth highest priority mode is “Turn” mode, which includes the left turn light on the primary road in the sequence of lights. This mode is activated by pressing a push button and is disabled by pressing the button again. The fifth highest priority mode is “Pedestrian” mode, which turns the lights for both streets to red (after transitioning through yellow of course) and turns on an LED representing the walk signal for pedestrians. Lastly, “Normal” mode is the lowest priority which simply cycles through the lights, giving each street 10 seconds of time on the green light and 2 seconds for each of the transitions from yellow to red and red to green the other direction. We chose these times to make it easier to test (we did not have hours to sit around watching the lights change). We chose to make the times for the green traffic lights longer since this is what would occur in a real-life scenario. In order to make the traffic light usable in a real-world situation a few parameters in the code would need to be changed to account for the increased times.

A flow chart showing the operation of our system as viewed from an external user’s standpoint is shown below.



In order to implement this functionality for our system we used a multithreaded approach. There are two main threads that execute within the processor – the Poller and the Monitor. The poller thread is used to constantly poll the switches and buttons to see what state they are in and if there have been any changes in state since the last polling. It is constantly running and when it detects a positive edge on a pushbutton or a switch it determines whether the current task should be preempted based on the priority of each mode that was given above. The second thread is the monitor thread. Due to the fact that the poller is continually running, the monitor thread is given higher priority within the processor so it is not starved by the poller and is able to execute when it needs to. When the monitor finishes a task he determines which task is scheduled to be executed next, with the default being normal mode of operation. When the poller determines that one of the higher priority modes of operation should be executed it tells the monitor to terminate the current task and run that one.



**Testing Strategy**

The testing strategy was largely driven by the need for independent testing. Due to the fact that the system is a large entity that functions together as one unit, we were not able to test the individual modules and connections between them. The coding was done in a style that largely resembled extreme programming. Tom was the main coder, with Joe sitting by his side as we worked through the various aspects of our implementation. In order to design the project we first laid out a flow chart in order to account for various testing points. As will be explained further in the design choices section below, we began with an approach the involved hardware interrupts, but midway through the implementation phase found that it was in our best interest to switch to a polling implementation for the buttons and switches. Therefore, we had to do a redesign of our flowchart and included new testing points.

Additionally, as we would finish implementing each mode of operation for the traffic light we would test the operation completely and ensure it did not break any other aspects before moving on to implementing the next mode. Once the entire project was implemented we did a full system test and ensured that all priority requirements were met successfully.

**Interesting Design Choices**

The most interesting design choice that we had to make was to switch from an interrupt driven system to one that relies on polling to monitor the button pushes and switches. When we first began to try and implement the interrupt method of using the switches and button presses we were having trouble getting the hardware to work correctly. Eventually we found a method of using interrupts, but by that time we had already devised a poller system that worked effectively with our multithreaded approach and were well on our way to finishing the project. With more time we could have gone back and re-explored the interrupt based solution.

Our use of a monitor thread with a higher priority than the poller thread was also an interesting design decision that allowed us the preempt the poller whenever the monitor needed to run, but gave the poller enough time on the microprocessor to be able to react without a delay to changes in the buttons and switches.

**Team Member Contributions**

|  |  |
| --- | --- |
| **Name** | **Contributions** |
| Tom Dickman | Code writer and debugger |
| Joe Lovelace | Code debugger and tester |
| Andrew Steller | Final report editor |
| Ben Zerhusen | Final report writer, Implementation Investigation, Code tester |

**PowerPlay Power Analysis:**

PowerPlay Power Analyzer Status Successful - Mon Dec 10 15:22:50 2012

Quartus II 32-bit Version 12.0 Build 178 05/31/2012 SJ Web Edition

Revision Name de1\_tutorial

Top-level Entity Name de1\_tutorial

Family Cyclone II

Device EP2C20F484C7

Power Models Final

Total Thermal Power Dissipation 79.67 mW

Core Dynamic Thermal Power Dissipation 0.66 mW

Core Static Thermal Power Dissipation 47.37 mW

I/O Thermal Power Dissipation 31.63 mW

**Maximum Speed:**

50 MHz

**Fitter Resource Usage Summary:**

Total logic elements 2,953 / 18,752 ( 16 % )

-- Combinational with no register 1035

-- Register only 524

-- Combinational with a register 1394

Logic element usage by number of LUT inputs

-- 4 input functions 1276

-- 3 input functions 720

-- <=2 input functions 433

-- Register only 524

Logic elements by mode

-- normal mode 2241

-- arithmetic mode 188

Total registers\*   1,986 / 19,649 ( 10 % )

-- Dedicated logic registers 1,918 / 18,752 ( 10 % )

-- I/O registers 68 / 897 ( 8 % )

Total LABs:  partially or completely used 244 / 1,172 ( 21 % )

User inserted logic elements  0

Virtual pins 0

I/O pins 99 / 315 ( 31 % )

-- Clock pins  6 / 8 ( 75 % )

Global signals  7

M4Ks 14 / 52 ( 27 % )

Total block memory bits 44,032 / 239,616 ( 18 % )

Total block memory implementation bits 64,512 / 239,616 ( 27 % )

Embedded Multiplier 9-bit elements 0 / 52 ( 0 % )

PLLs 1 / 4 ( 25 % )

Global clocks 7 / 16 ( 44 % )

JTAGs 1 / 1 ( 100 % )

ASMI blocks 0 / 1 ( 0 % )

CRC blocks 0 / 1 ( 0 % )

Average interconnect usage (total/H/V) 7% / 8% / 7%

Peak interconnect usage (total/H/V) 24% / 24% / 24%

Maximum fan-out node nios\_system:u0|nios\_system\_altpll\_0:altpll\_0|altpll:sd1|\_clk0~clkctrl

Maximum fan-out 1114

Highest non-global fan-out signal nios\_system:u0|nios\_system\_nios2\_qsys\_0:nios2\_qsys\_0|W\_alu\_result[2]

Highest non-global fan-out 94

Total fan-out 15843

Average fan-out 3.29