Rojobot31
Theory of Operation
Revision 3.0

Table of Contents

Introduction

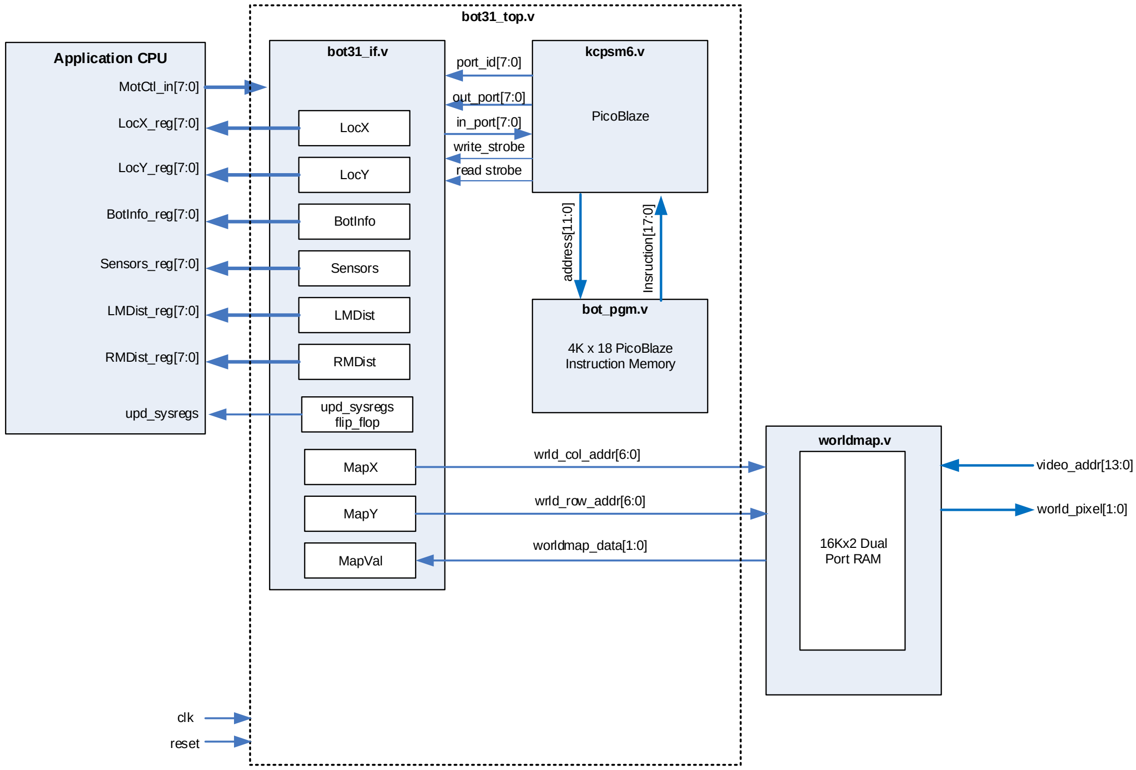
The Rojobot (**Ro**y and **Jo**hn's **Bot)** models a simple robot moving through a simple environment. The robot is based on the UP1-Bot described in Hamblen/Furman's *Rapid Prototyping of Digital Systems – A Tutorial Approach*, Kluwer Academic Publishers, 2001. As described in Hamblen/Furman, the robot is a platform with two wheels, each driven by an independent motor. A Teflon skid serves to stabilize the platform.

This document describes the internal operation of the Rojobot version 3.1. This document is made available to those of you who are interested in learning more about how the Rojobot is implemented but the information contained in this document is not required to incorporate the Rojobot into your design. In fact, Rojobot 3.1 is packaged as a Xilinx Vivado IP block which can be treated as a "Black Box" in your project. If you're curious about how it was implemented, though…read on.

Theory of Operation

Rojobot31 Architecture

The Rojobot models a simple two-wheeled mobile platform that is capable of moving around a terrain consisting of open space, obstructions, and a black line. The emulation is implemented in Picoblaze Assembly language running on a Xilinx Picoblaze softcore CPU with the supporting logic written in Verilog. The Rojobot should be thought of as a peripheral to an external CPU. As with many slave peripherals, the CPU communicates to the Rojobot through I/O registers (defined in the *Rojobot31 Functional Specification*). The terrain is modeled in a separate dual-port block RAM (called the *worldmap*) which connects to both the Picoblaze emulating the Rojobot and a video controller that can be used to display the map. The block diagram on the next page describes the Rojobot's structure and interfaces.



BOT31_IF.V

bot31_if.v is the Verilog module that implements the I/O interface between the PicoBlaze emulating the Rojobot and a separate application CPU, and also to the world_map BRAM. *Bot31.v* responds to INPUT and OUTPUT instructions from the PicoBlaze and either returns the value of an input port (PicoBlaze INPUT instruction) or writes a value to one of its registers (PicoBlaze OUTPUT instruction). Since the PicoBlaze can only perform one I/O instruction at a time and there are several registers in the interface to update simultaneously, *bot31_if.v* supports a two phase approach for updating the registers. The emulator PicoBlaze performs OUTPUT instructions to a set of state registers internal to bot31_if.v and then performs two sequential writes to a special I/O port that cause the internal registers to simultaneously updates the ports that are visible at in the interface. This synchronous update insures that the entity (hardware or software) using the registers receives a consistent view of the Rojobot's status.

WORLD_MAP.V

world_map.v is the Verilog model that implements the map of the RojoBot's world. It consists of a 16K x 2 dual-port RAM that returns the type (open, obstruction, black line, reserved) of that location. The location types are encoded as follows:

Encoding (binary)	Location Type
00	Open space
01	Black line
10	Obstruction
11	*RESERVED*

The world map is arranged as a 128 x 128 grid which is addressed by a 14-bit address arranged as {row address[6:0], column address[6:0]}. Column 0 is the leftmost location in a row and column 127 is the rightmost location in a row. Row 0 is the topmost location in a column and Row 127 is the bottommost location in a column.

world_map.v also provides a second independent interface to address the memory containing the map. This second interface is intended to be used by video generation logic to display the map on a monitor.

KCPSM6.V

kcpsm6.v is the PicoBlaze module. It is instantiated in *bot31_top.v*.

BOT_PGM.V

bot31_pgm.v contains the instruction memory for the Rojobot's PicoBlaze CPU. It is generated by the kcpsm6 assembler (*kcpsm6.exe*) and is included in the rojobot31 IP block.

Rojobot31 Program (bot_pgm.psm)

The emulator is implemented as an Assembly language program for the PicoBlaze. The program is about 500 PicoBlaze instructions. This section does not describe the program in detail but contains an overview of how the key functions work.

> The source code for the simulator program is not included in the distribution package for the projects but is available upon request. There is plenty of room to add your own enhancements for, say, a final project. The code is modular (lots of functions) and heavily commented.

MAIN LOOP

Like most embedded system programs, the Rojobot emulator does its processing in an infinite loop. It is not, however, interrupt driven as many embedded programs are. Instead the main loop reads the Motor Control inputs, emulates the Rojobot's movement, updates the interface registers (including its location in the emulated world, "sleeps" (executes a busy-wait loop) for a period of time and repeats the loop. The pseudo-code for the main loop is as follows:

```
while (1)  {
      get motor control inputs
      determine Bot's movement (fwd, rev, slow or fast turn, etc.)
      simulate the rojobot and update the distance counters
      update the interface registers
      if (distance change >= move threshold {
            calculate new rojobot orientation
            calculate new rojobot location coordinates
            update rojobot's location and orientation
            get new sensor values
            clear rojobot distance counters
      }
      busy-wait for next sample interval
}
```

It is worth noting the *if* statement in the main loop since it is key to changes in the Rojobot's location or orientation. The distance counters in the Rojobot emulation are updated roughly every 50msec, but practically speaking, the wheels on a physical robot would not move very far in a twentieth of a second unless they were turning very quickly. The BotSim accumulates changes in the distance counters until they pass a "move threshold." The move threshold simulates enough wheel movement to cause the Rojobot to move to its next location or change orientation.

NOTE: THE MOVE THRESHOLD AND SAMPLE INTERVAL CAN BE CHANGED IN THE ROJOBOT'S EMULATOR SOURCE CODE AND BY CONFIGURING THE EMULATION USING THE BOTCONFIG REGISTER. SEE THE ROJOBOT31 FUNCTIONAL SPECIFICATION FOR DETAILS.

EMULATING THE ROJOBOT

The Rojobot is implemented as a function call in the BotSim. The function reads the motor control input and increments one or both of the motor distance counters if their speed is greater than 0 (on). This function, in conjunction with the calc_movement() function in the emulator program determines what movement (or action) the Rojobot is performing.

CALCULATING THE BOT'S ORIENTATION AND LOCATION

The Rojobot's orientation (direction it is facing) is dependent on its current orientation and what it's been told to do. The orientation of the Bot changes only when the Bot is turning to left or right. A slow left or right turn causes the Rojobot to turn 45 degrees every time the move threshold is passed. A fast left or right turn causes the Rojobot to turn 90 degrees in the same period of time. For the sake of simplifying the algorithm all orientation changes happen in place (e.g. the Rojobot rotates around its center). The only time the Rojobot moves to a different location in its world is when it is moving forward or reverse.

The Rojobot's new location is dependent on its current location and orientation. A Rojobot moving North or South decrements (North) or increments (South) the row (Y-coordinate) portion of the location address and leaves the column (X-coordinate) portion unchanged. A Rojobot moving East or West increments (East) or decrement (West) the column (X-coordinate) portion of the location address and leaves the row (Y-coordinate) portion unchanged. A Rojobot moving Northeast, Southeast, Southwest, or Northwest will change both the X and Y coordinates appropriately.

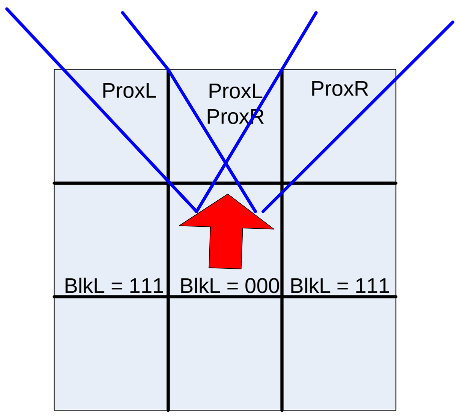
IMPORTANT: The Rojobot will not move onto a location that is obstructed. Instead it will stay in its current location with its simulated motors turning, simulated battery wearing out, simulated friction heating the simulated tires and wearing out the simulated motor brushes….ok, not really, but it will continue to try, and fail, to move onto the obstructed location until it is told to stop or change the direction it is moving.

CALCULATING THE NEW SENSOR INFORMATION

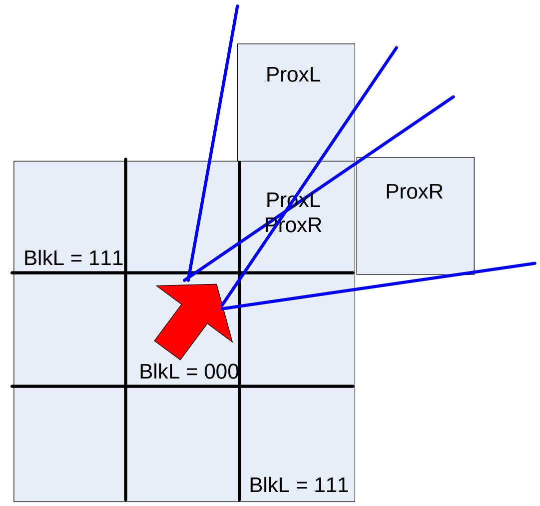
The Rojobot's proximity sensors are looking ahead one location and its black line sensors are looking underneath the front of the Bot. As a result, calculating the new sensor values involves examining both the location under the Bot to get the black line sensor value and several locations in front of the Bot.

Collecting the black line sensor value is simple. If the Rojobot's location is of type = 2'b01 (black line) then {BlkLL, BlkLC, BlkLR} is assigned a value of 3'b000. Otherwise it is assigned a value of 3'b111.

Collecting the proximity sensor value requires reading different locations surrounding the Rojobot. This is illustrated in the following two figures:



North



Northeast

The first figure illustrates the range of the proximity sensors when the Rojobot is facing North. The get_sensors() function reads the 3 locations to the left front, directly in front, and right front of the Rojobot. ProxL is set to '1' if either the left front or directly in front location type is 2'b10 (obstruction). ProxR is set to '1' if either the right front or directly in front location type is 2'b10 (obstruction).

The second figure illustrates the range of the proximity sensors when the Rojobot is facing Northeast. For this case the get_sensors() function reads the location directly in front of it and then two locations above and to the right of that location. The other locations to read can be abstracted from these illustrations.