Investigation of a Reconfigurable Processor for a SLAM System

by:

Roy Lycke
Ji Li
Ryan Hamor

Iowa State University

CPRE 583
December 17th, 2011

Introduction

The purpose of this project was to take a computational intensive algorithm implemented on an embedded processor, benchmark it, and then create custom instructions in reconfigurable hardware to speed that execution up. The algorithm chosen was a Rao-Blackwellised Particle Filter type SLAM algorithm. SLAM stands for Simultaneous Localization and Mapping. By recursively estimating a vehicles pose and at the same time collecting landmarks from the environment and correlating them to that pose to create a map of the environment.

For this project we choose the predict step of the particle filter to be implemented in hardware for acceleration. The predict step of the particle filter is responsible for completing the following algorithm:

Algorithm 1:

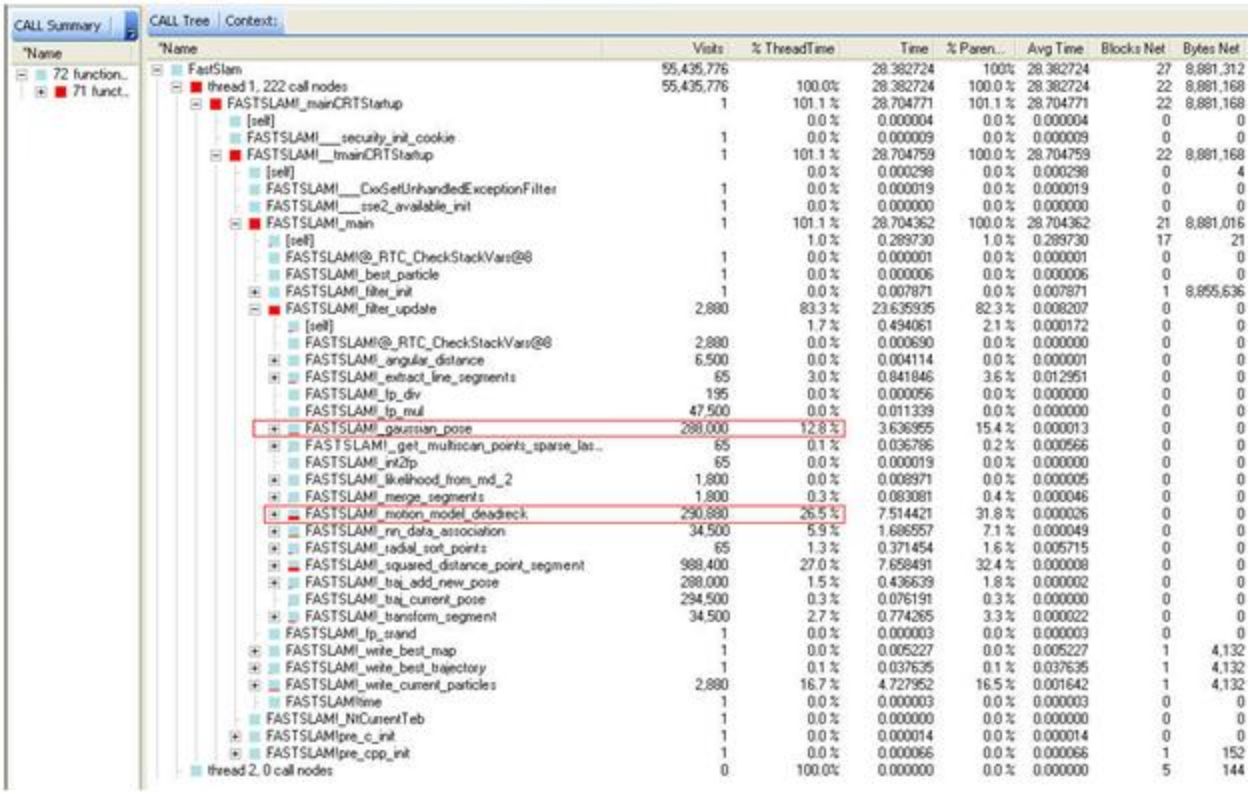
For each particle
Using the previous and new odometer readings derived an estimated control vector.
Using the control vector estimate compute a predicted pose.
The predicted pose and the measured sensor variance is used to draw a sample from a Gaussian distribution.
End For

The goal will be to benchmark algorithm computation time for the software / hardware vs. the all software version. The breakdown of the paper is as follows; section one discusses the hardware implementation of the predict step, section two gives detailed analysis of the profiling and bench marking process, section three reviews the simulation results from hardware and section four presents conclusions.

Hardware / Software Co-design of Predict Step for SLAM Algorithm

During our initial project work we found a SLAM algorithm developed by Kris Beevers for a PHD thesis called ratbot-slam. This algorithm was originally designed to run on an embedded processor using a fixed point representation of 16.16. The algorithm was modified to take in data and odometer information from encoders and laser scanners in place of the IR range finders used in the thesis. An initial run was done using "usc-sal200-021120" from www.radish.com.

In figure 1 below shows the initial bench mark results running the algorithm on a PC. The results, captured with Glow Code, show that 83.3% of the algorithms time is spent in the filter update function. Looking deeper into we targeted two other functions, gaussian_pose and motion_model_deadreck, these two functions compose the predict step in the algorithm and consume 39% of the application run time. With these numbers we are hoping to get close to a 1.66x speed up. As a result of this we targeted two main functions to be implemented in hardware and be heavily pipelined. Our analysis of the filter update function shows we could have implemented more of the algorithm in hardware; however, due to time constraints we have limited the hardware design to the two functions listed above.



CALL Summary | CALL Tree | Context:

Name	Visits	% ThreadTime	Time	% Paren...	Avg Time	Blocks Net	Bytes Net
FastSlam	55,435,776		28.382724	100%	28.382724	27	8,881,312
thread 1, 222 call nodes	55,435,776	100.0%	28.382724	100.0 %	28.382724	22	8,881,168
FASTSLAM!_mainCRTStartup	1	101.1 %	28.704771	101.1 %	28.704771	22	8,881,168
[self]		0.0 %	0.000004	0.0 %	0.000004	0	0
FASTSLAM!___security_init_cookie	1	0.0 %	0.000009	0.0 %	0.000009	0	0
FASTSLAM!__tmainCRTStartup	1	101.1 %	28.704759	100.0 %	28.704759	22	8,881,168
[self]		0.0 %	0.000298	0.0 %	0.000298	0	4
FASTSLAM!___CxxSetUnhandledExceptionFilter	1	0.0 %	0.000019	0.0 %	0.000019	0	0
FASTSLAM!___sse2_available_init	1	0.0 %	0.000000	0.0 %	0.000000	0	0
FASTSLAM!_main	1	101.1 %	28.704362	100.0 %	28.704362	21	8,881,016
[self]		1.0 %	0.289730	1.0 %	0.289730	17	21
FASTSLAM!@_RTC_CheckStackVars@8	1	0.0 %	0.000001	0.0 %	0.000001	0	0
FASTSLAM!_best_particle	1	0.0 %	0.000006	0.0 %	0.000006	0	0
FASTSLAM!_filter_init	1	0.0 %	0.007871	0.0 %	0.007871	1	8,855,636
FASTSLAM!_filter_update	2,880	83.3 %	23.635935	82.3 %	0.008207	0	0
[self]		1.7 %	0.494061	2.1 %	0.000172	0	0
FASTSLAM!@_RTC_CheckStackVars@8	2,880	0.0 %	0.000690	0.0 %	0.000000	0	0
FASTSLAM!_angular_distance	6,500	0.0 %	0.004114	0.0 %	0.000001	0	0
FASTSLAM!_extract_line_segments	65	3.0 %	0.841846	3.6 %	0.012951	0	0
FASTSLAM!_fp_div	195	0.0 %	0.000056	0.0 %	0.000000	0	0
FASTSLAM!_fp_mul	47,500	0.0 %	0.011339	0.0 %	0.000000	0	0
FASTSLAM!_gaussian_pose	288,000	12.8 %	3.636955	15.4 %	0.000013	0	0
FASTSLAM!_get_multiscan_points_sparse_las...	65	0.1 %	0.036786	0.2 %	0.000566	0	0
FASTSLAM!_init2fp	65	0.0 %	0.000019	0.0 %	0.000000	0	0
FASTSLAM!_likelihood_from_md_2	1,800	0.0 %	0.008971	0.0 %	0.000005	0	0
FASTSLAM!_merge_segments	1,800	0.3 %	0.083081	0.4 %	0.000046	0	0
FASTSLAM!_motion_model_deadreck	290,880	26.5 %	7.514421	31.8 %	0.000026	0	0
FASTSLAM!_nn_data_association	34,500	5.9 %	1.686557	7.1 %	0.000049	0	0
FASTSLAM!_radial_sort_points	65	1.3 %	0.371454	1.6 %	0.005715	0	0
FASTSLAM!_squared_distance_point_segment	988,400	27.0 %	7.658491	32.4 %	0.000008	0	0
FASTSLAM!_traj_add_new_pose	288,000	1.5 %	0.436639	1.8 %	0.000002	0	0
FASTSLAM!_traj_current_pose	294,500	0.3 %	0.076191	0.3 %	0.000000	0	0
FASTSLAM!_transform_segment	34,500	2.7 %	0.774265	3.3 %	0.000022	0	0
FASTSLAM!_fp_srand	1	0.0 %	0.000003	0.0 %	0.000003	0	0
FASTSLAM!_write_best_map	1	0.0 %	0.005227	0.0 %	0.005227	1	4,132
FASTSLAM!_write_best_trajectory	1	0.1 %	0.037635	0.1 %	0.037635	1	4,132
FASTSLAM!_write_current_particles	2,880	16.7 %	4.727952	16.5 %	0.001642	1	4,132
FASTSLAM!time	1	0.0 %	0.000003	0.0 %	0.000003	0	0
FASTSLAM!_NtCurrentTeb	1	0.0 %	0.000000	0.0 %	0.000000	0	0
FASTSLAM!pre_c_init	1	0.0 %	0.000014	0.0 %	0.000014	0	0
FASTSLAM!pre_cpp_init	1	0.0 %	0.000066	0.0 %	0.000066	1	152
thread 2, 0 call nodes	0	100.0%	0.000000	0.0 %	0.000000	5	144

Figure 1 - Initial Profiling of SLAM Algorithm

Top Level Design

As discusses above, the main object of our project is to take an existing SLAM implementation and accelerate as much of it as possible in the time limit allotted for the project. To enable us to meet this goal we need a reliable and repeatable way to benchmark the system. We choose to do this by running a log file of odometry and laser scan data over UDP to an ml507 development board with an F70T Xilinx 5 Vertex processor. The FPGA on the ml507 has an embedded PowerPC 440 processor that the bulk of our algorithm shall run on. We then implement the motion_model_deadreck and gaussian_pose functions in hardware. To communicate with the reconfigurable fabric of the FPGA we connected to the APU controller. The APU controller is a hardcore piece of hardware that allows for special instruction and data to be committed to the reconfigurable fabric via software assembly instructions. Two such functions we take advantage of are the load and store functions. Below is the top level architecture of the system.

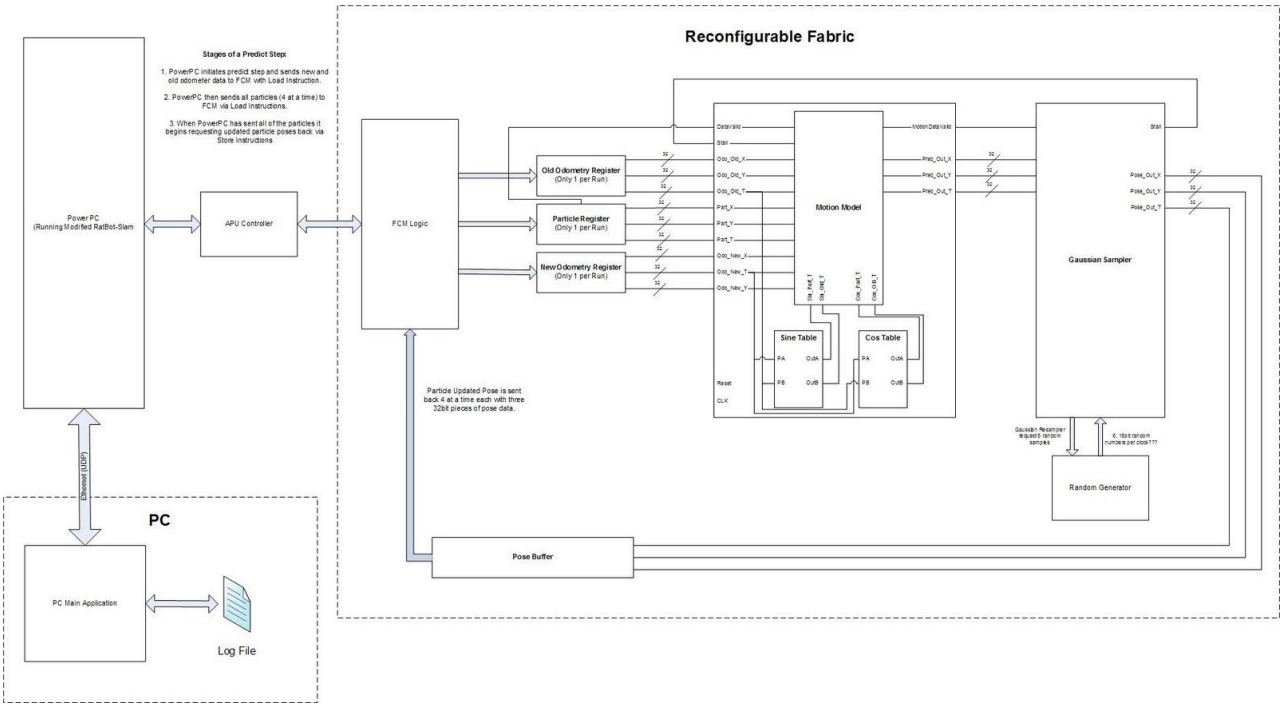


Figure 2 - Top Level System Block Diagram

The log file is preprocessed in MatLab to only use the odometry data and five out of the 180 laser scan points. The main PC program then reads this modified log file, converts it into packets, and transmits them via UDP to the PowerPC which is running the modified ratbot-slam algorithm.

The next sections of the paper shall discuss the detailed implementation of the specific parts in our system.

Software Design

The software system of this project is the code used for implementing the original SLAM algorithm on the PowerPC and the PC based data loading functions. The software described in the following section is used for loading data, communicating over a network, running the SLAM algorithm, and returning its output. This section does not describe the operation of the SLAM algorithm used in this project; that information is covered in the introduction.

PC Data Loader

The first portion of the software design is the PC data loader. This portion of the project code is run by the user on the PC and is responsible for loading the sensor data to the FPGA for running the algorithm and returning the results to the user. This code is run entirely on the PC, excluding the receiving code on the FPGA, and pulls data from input files on the PC.

The operation of the PC data loader starts by loading the input data file into structures for sending to the FPGA. The data structures for this transmission had to be formatted so that the

endianess of the two systems, PC and FPGA, did not impact communication or interpretation of values. The second step of the data transfer was sending the data the FPGA via UDP packets. Once the data was sent to the FPGA the PC data loader would wait for a request from the FPGA for more data. The PC data loader would continue to send data until the input data files were empty. Once the input files had reached end-of-file the PC data loader would send a request to the FPGA to transmit back the output files from the SLAM algorithm. As the output files would get sent back to the PC data loader they would be written into output files with no change in formatting.

SLAM Running Code

The second region of the software design is the code that runs on the Xilinx FX70 FPGA and runs the SLAM algorithm. This code takes network input from the PC data loader, manages the operation of the SLAM algorithm, and when requested sends data back to the PC data loader.

The SLAM running code runs in an infinite loop waiting for input form the network. When a request is sent to load data the SLAM algorithm reads in the input data from the network, loads the data into sensor data structures then loads each sensor data structure into the SLAM algorithm. Each time a new sensor data is loaded into the SLAM algorithm, the algorithm performs all of its calculations and processes the appropriate output maps for the environment described by the input sensor data. Once all the data is loaded into the SLAM algorithm the running code sends a request for data to the PC data loader and returns to waiting in the infinite loop.

The other case for the SLAM running code is a request for output. This is usually performed once all data from an input file is exhausted, but all needed is a request from the network. The SLAM running code will first calculate the output map files for the environment currently viewed by the SLAM algorithm based on previously input sensor data. Once these output map files are generated they are sent over the network to the requester with no formatting, the data must be loaded into an output file directly. Once all the data from the output maps files is sent the SLAM running code sends a request for data to the PC data loader and returns to waiting in the infinite loop.

The two previous elements, the PC data loader and the SLAM running code, have one weakness in that they communicate via UDP. Due to this unchecked method of communication a single dropped or mal-formatted packed could cause the program to hang. This was observed to occasionally happen after several minutes of network communication, even using handshaking techniques to control program synching. This could be addressed in future implementations of this project by using TCP rather than UDP. The decision to use UDP for this project was made because there was a previous implementation of network communication working for the target FPGA and adaptation of a working system allowed for further efforts to be spent on other elements of the design.

APU Integration

The last portion of the software design is the integration of the APU co-processor with the C implementation of the SLAM algorithm. This was performed by modifying the predict step of the SLAM algorithm to perform calls to our APU co-processor. The current implementation

uses only loads and stores to the co-processor to initialize data analysis. The operation of the APU co-processor is discussed in further detail in the remainder of the document.

FCM Module Design

As discussed above we took advantage of the APU controller interface and the special load and store instructions for our accelerated hardware. Our top level hardware component is called apu_inv after our MP3 project from the class. The base apu_inv allowed for a load instruction directed at register 2 of the FCM to invert all colors of one pixel of an image. Our modified version added additional registers to load data specific to our project.

There are three main types of data that must be loaded to the HW Predict Step; Old odometry data from the last sample, new odometry data from the current sample, and the current pose of the particle that a new pose shall be predicted. Each of the data structures are a pose type data structure which means they consist of 3 32bit fixed point numbers (16.16 bit division). The algorithm for loading the particle data is shown below.

Algorithm 2:

```
For each filter update step
        Load(Old Odometry Data, FCM Reg 0x01)
        Load(New Odometry Data, FCM Reg 0x03)
        For each Particle
                Load (Particle Pose Data, FCM Reg 0x04)
        end loop
        For each Particle
                Store(Particle Pose Data)
        end loop
end
```

As the FCM receives the load instructions it determines where to put the data based on the desired register specified in the load instruction as shown in the table below:

Load Instruction Register	FCM Register	Description
0x01	Old_Odometer_Data_reg(95 downto 0)	This register holds the previous sampled odometry
0x0395	New_Odometer_Data_reg(downto 0)	This register hold the current update step's odometry data.
0x04	Particle_Data_reg(95 downto 0)	

The next two sections shall describe the two main hardware subsystems, Motion Model and Gaussian Pose.

Motion Model

After the first three registers are filled the FCM module signals that particle data is ready to start processing by asserting the Particle_Data_Valid_reg signal to the Motion Model. A typical run for one predict step involves 100 particles that must be updated while the old and new odometry information remains unchanged. Due to this we have pipe-lined all the subsystems to continuously take in new particle pose information. The top level inputs and outputs for the Motion Model are listed in the table below:

Signal Name	Description	Input/Output
clk	Motion Model clock signal	Input
reset	Motion Model reset signal to clear all internal registers	Input
DataValid	Signal to Motion Model that all input data has been loaded and processing can begin	Input
Stall	Signal from the gaussian_pose subsystem that the Motion Model should stop processing while keeping the interal states of its registers	Input
OLD_X (31 downto 0)	Old odometry x value	Input
OLD_Y (31 downto 0)	Old odometry y value	Input
OLD_T (31 downto 0)	Old odometry angular value	Input
NEW_X (31 downto 0)	New odometry x coordinate	Input
NEW_Y (31 downto 0)	New odometry y coordinate	Input
New_T (31 downto 0)	New odometry angular coordinate	Input
Part_X (31 downto 0)	Current particle x coordinate	Input
Part_Y (31 downto 0)	Current particle y coordinate	Input
Part_T (31	Current particle angular coordinate	Input

downto 0)		
DataValidOut	Used to signal gaussian_pose subsystem that there is data to be processed	Output
PRED_T	Nominal predicted new angular coordinate for the particle	Output
PRED_X	Nominal predicted new x coordinate for the particle	Output
PRED_Y	Nominal predicted new y coordinate for the particle	Output

To create the Motion Model the modified C code was mapped to a data flow graph. The data flow graph was then analyzed to extract areas of parallelism and identify where pipelines maybe necessary. The figure below is the data flow graph for the Motion Model:

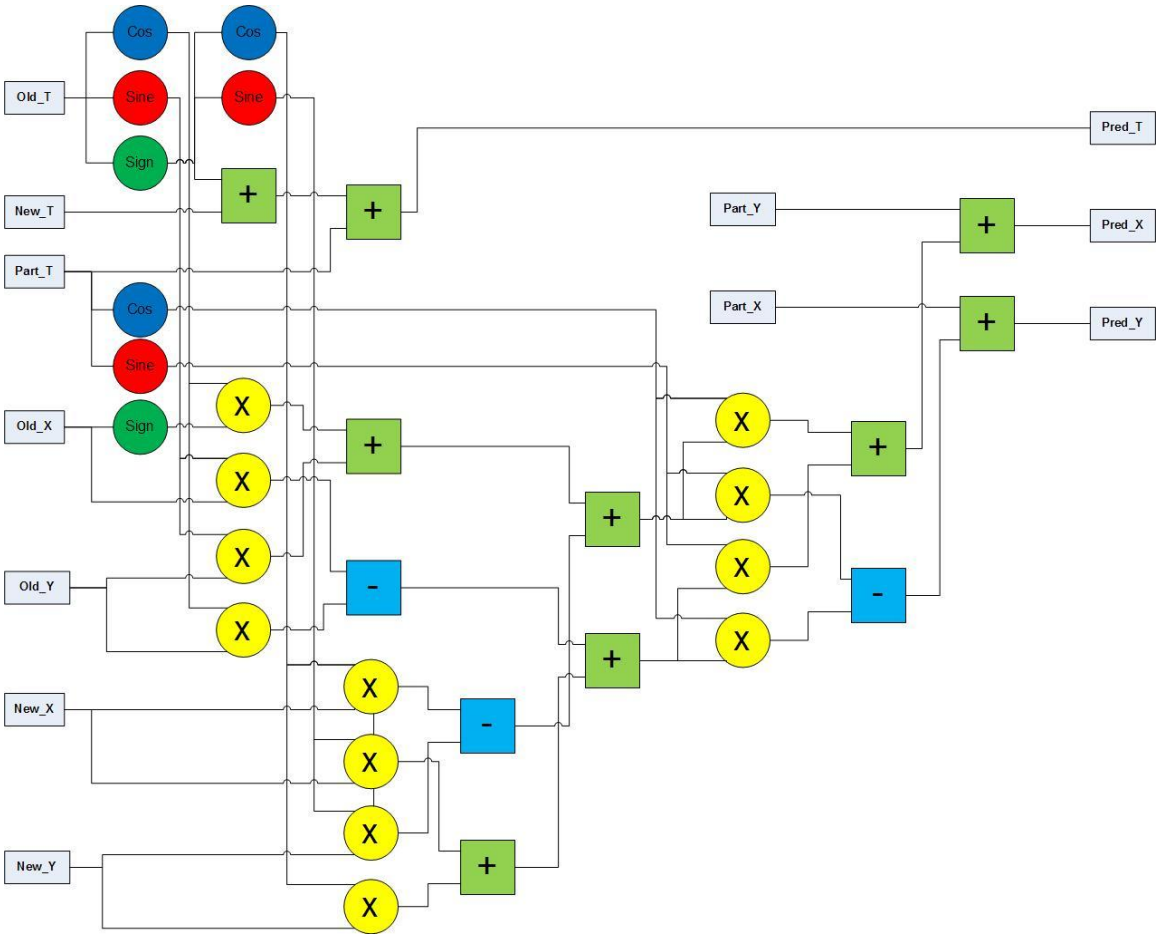


Figure 3 - Data Flow Graph of the Motion Model

From the data flow graph is apparent that not all operation can be completed in parallel. The next design decision was to determine if pipelines needed to be used. To do this the motion model was created to run in one process and the circuit synthesized to analyze timing. It was

found that the circuit timing was approximately 40ns, which meant that the clock would have to be slower than 25MHz. The original FCM module was setup to run at 100 MHz and we wanted to be able to maintain at least this speed. Based upon this, the decision was made to pipeline the Motion Model.

From the flow graph the Motion Model can be divided up into 8 stages. One additional stage is used to align the registers. Timing on this 9 stage Motion Model indicated that we were under the 10 ns requirement to run at 100 MHz. In addition to the 9 stages for the motion model logic three main components were also designed; Cosine BRAM table, Sine BRAM table, and 32bit 16.16 fixed point multiplier.

Common Motion Model Pipeline Stages

While the intermediate signals for each of the motion model stages varied there were some core signals that each pipeline stage contained to keep the data flowing. Those signals are listed in the table below:

Signal Name	**Description**	**Input/Output**
clk	Pipeline stage clock	Input
reset	Synchronous reset to clear all pipeline registers	Input
Stall	Halts the pipeline stage execution, but keeps the register states when asserted high	Input
DataValid	Enables pipeline stage execution on input data	Input
DataValidOut	Signals next pipeline stage that data is available on the pipeline stage output registers	Output

Sine and Cosine BRAM Table

The Motion Model required that the sine and cosine of the input angles be computed as part of its functionality. To facilitate this we use a look up table of 16.16 sine and cosine values stored in BRAM memory. In addition to the BRAM memory it was also necessary to add logic to perform the following tasks:

- Determine the sign of the angle and set the sign of the output sine values accordingly
- Bit shift the input angle to a look up value of 0 to 402
- Range limit the look value if the input angle was greater than pi

The input and output table of the look-ups is shown below:

Signal Name	Description	Input/Output
clk	Lookup Table clock signal	Input
Angle (31 downto 0)	16.16 fixed point angle in radians that the trig function should be evaluated on	Input
SineAngle / CosineAngle (31 downto 0)	16.16 fixed point trig-function value	Output

32bit 16.16 Fixed Point Multiplier

When working with the 16.16 fixed point format a special multiplier needed to be developed. The multiplier consists of a 32 bit signed multiply with addition shift logic to transform the result to a 16.16 fixed point number. The input and output table is given below:

Signal Name	Description	Input/Output
MultInA (31 downto 0)	First 16.16 fixed point value to be multiplied	Input
MultInB (31 downto 0)	Second 16.16 fixed point value to be multiplied	Input
MultOut (31 downto 0)	16.16 fixed point value of A x B	Output

Simulation Results for Motion Model

To verify the correct operation of the Motion Model hardware a simple C program was used to load in 3 sets of particle information. The figure below is the simulation of the Motion Model. The simulation show contains 5 consecutive loads; old odometry data, new odometry data, and 3 particle poses'. Once the first particle pose has been loaded the datavalid signal is asserted. 10 clocks later the first new particle pose estimate is output along with the datavalid out signal. The next 2 new pose estimates are output at less than 10 clocks due to the pipeline stages.

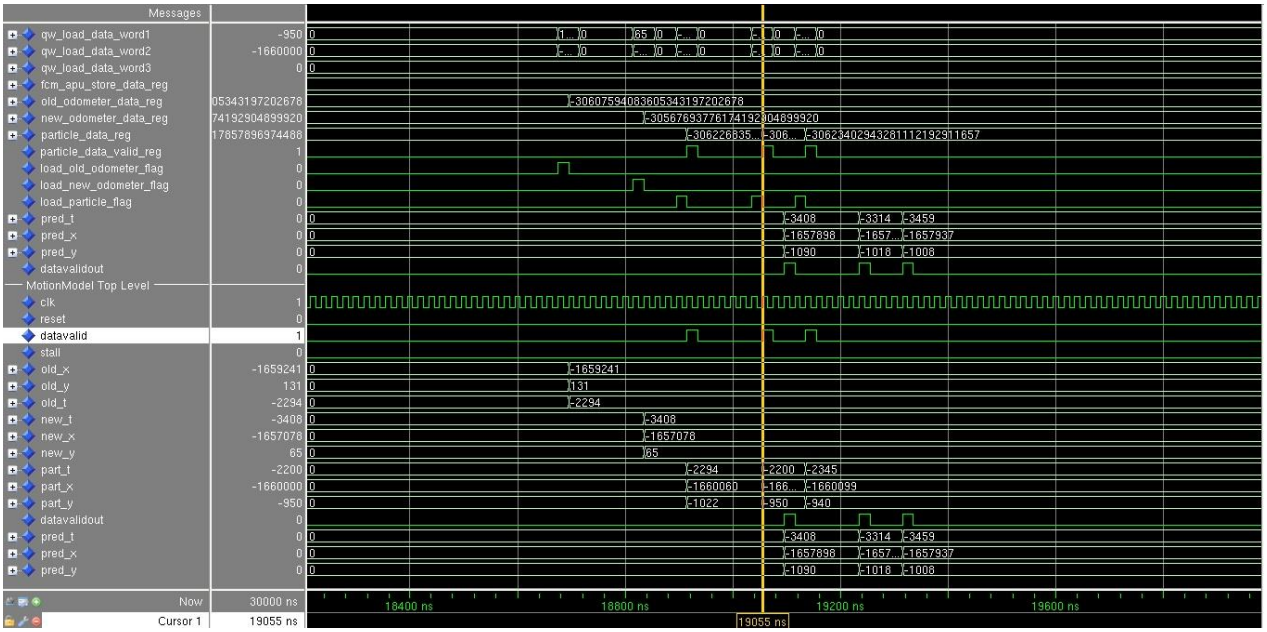


Figure 4 - Simulation of Motion Model Subsystem

Implementation of function of gaussian_pose

The C program code of *gaussian_pose* function is shown as follows. It consists of three *gassian* functions. Because this is fast version of slam algorithm, the original C program code ignores cross-variance terms, which are set to constants. So for this *gaussian_pose* function, it takes three input values, which are the predicted value of x, y, and t, and outputs three numbers processed by *gaussian* functions for them.

```
void gaussian_pose(const pose_t *mean, const cov3_t *cov, pose_t
*sample)
{
  sample->x = gaussian(mean->x, fp_sqrt(cov->xx));
  sample->y = gaussian(mean->y, fp_sqrt(cov->yy));
  sample->t = gaussian(mean->t, fp_sqrt(cov->tt));
}
```

The detail C program code of *gaussian* function is shown as below. It can work in two ways, which are determined the variable *cached*. When the static variable *cached* equals 1, this function only does one multiplication and one addition calculation. When it equals 0, it will do more calculations, including generating and selecting random numbers, look-ups to the Gaussian table, multiplications, and additions. Every time the *gaussian* function is called, the *cached* signal switches between 0 and 1.

```
fixed_t gaussian(fixed_t mean, fixed_t stddev)
{
  static int cached = 0;
  static fixed_t extra;
  static fixed_t a, b, c, t;
```

Parallelism analysis

As the code shows, two high-level parallelisms exist and can be used to accelerate the calculation speed:

1. *gaussian_pose* function consists of three *gaussian* functions, and they can run simultaneously. Thus, these three functions can be realized using multi-threads technique.
2. *Gaussian* function can be separated into two parts. The first part is to generate and select qualified couples of random numbers, and then look up the Gaussian table based on the random numbers achieved. The second part of work is to take the inputs from *motion_model_deadreck* function, random numbers and looking up result of Gaussian table to do further calculations. Actually, the first part is fully independent with the second one. While it does not need any input when it is running, the second part needs the input from previous calculation results. Thus, the *gaussian* function can be implemented as two components, which can run simultaneously. In that case, one component produces and stores random numbers without ceasing, and the other one consumes these calculation products when inputs arrive.

Techniques used for acceleration

To accelerate the calculations, three techniques are applied:

(1) Pipeline: Pipelines are heavily used for the hardware design. Even though there are high-level parallelisms exist in gaussian_pose functions. Its subcomponents still are made up with sequential operations. Because of that, multiple clocks are needed to process one set of inputs. By using pipeline techniques, new inputs can be fed before the previous ones are fully processed, and the hardware can take one input and produce one output every clock. Thus, this technique can greatly accelerate the calculations.

(2) Multi-thread: Multi-thread techniques are applied to implement the three *gaussian* functions of *gaussian_pose,* so that they can run parallel in the hardware.

High level architecture of *gaussian_pose*

The high level architecture of *gaussian_pose* entity is shown by figure 5. It consists of three *gaussian* entity, they work in parallel as multi-thread. And there are three *random_number_manager* components feed the random numbers needed to them. Each *gaussian* component produces one 32 bit output. The total three outputs of them are combined together and stored in a 96 bit FIFO to be accessed by APU later.

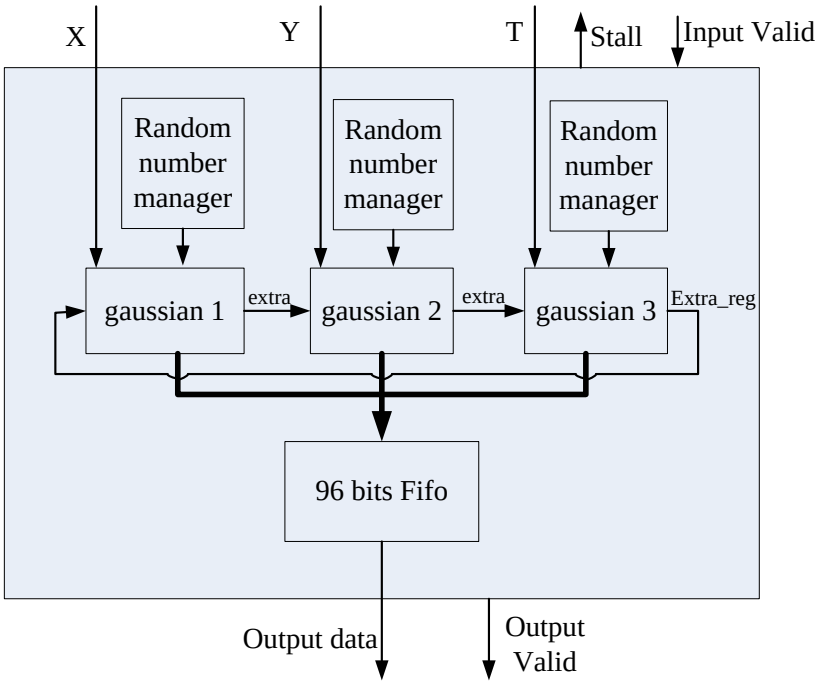


Figure 5: High level architecture of *gaussian_pose*

Actually, as shown by Fig. 5, the original g*aussian* function of the C program code is separated into two parts and implemented as two hardware components, which are named as Random_number_manager and Gaussian entities. The first component is to generate and select qualified couples of random numbers, and then look up the Gaussian table based on the random numbers achieved. The second one is to take the inputs from *motion_model_deadreck* function, random numbers and looking up result of Gaussian table to do further calculations. There are two reasons of this design:

(1) As discussed in above part, the work of *Random_number_manager* is completely independent with any other components and can work without any inputs, while *gaussian* entity needs to wait until the input from *motion_model_deadreck* is available. So it is reasonable to split their work into two parts to free the calculations of *Random_number_manager* from any other constraints.

(2) Not all the sets of random numbers generated are qualified for later calculations. One entity produces two 16 bits random numbers every clock, whose sum of square needs to be not bigger than 1^{32}. Otherwise, they should be discarded, and additional clocks are needed to generate new couple of random numbers for later procession. That means if system produces and evaluate the random numbers only when the input from *motion_model_deadreck* arrives, the later calculations need to wait indefinite clocks before one couple of suitable random numbers arrive. It has two disadvantages. Firstly, it unnecessarily wastes some clocks. Secondly, the indefinite clock needed to produce the suitable random numbers lead to indefinite clocks needed to finish the whole operations of gaussion functions. That means the multi-thread of Gaussian functions may need different clocks to finish their work, and this make it harder to synchronize them. To save the clock wasted and convenient the synchronization of multi-threads of Gaussian functions, the work related with random numbers is taken out from *gaussian* entity and is taken care by *Random_number_manager*

entity, which generates and selects the random numbers with non-stop no matter whether the inputs from *motion_model_deadreck* arrives. And the suitable ones are stored into a FIFO for later use of further calculations when needed until it is filled.

Random_number_manager

Random_number_manager entity generates and selected suitable couples of random numbers needed for later calculations. Its main structure is shown by Fig. 6. It consists of two random number generators, one Gaussian look up table, and one FIFO. Pipelines are used to implement this entity, and it has four working stages. Firstly, two 16 bits random numbers are produced by two random number generators. Secondly, the sum of square R of two random numbers is calculated. Thirdly, the system judge whether R is smaller than 1^{32}. If so, it takes 10 bits of R as the address to look up the Gaussian table, and the result will be stored into one FIFO along with the two random numbers for the later calculations.

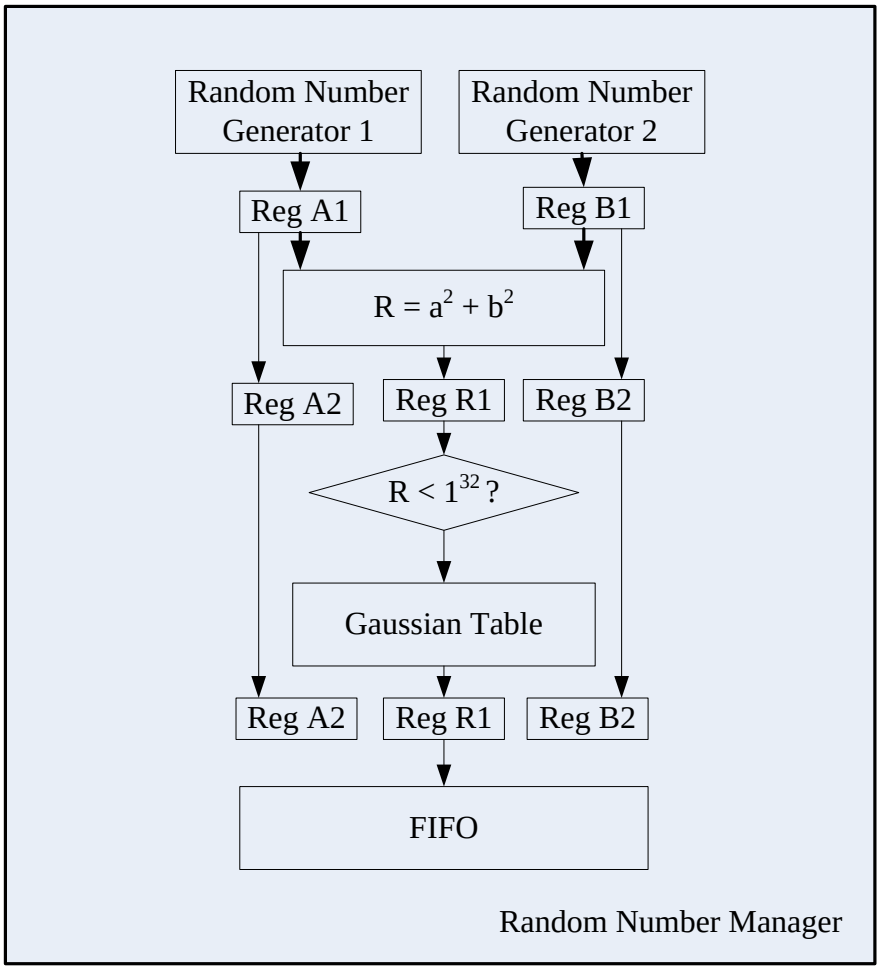


Figure 6: Diagram of Random_number_manager component

The pseudorandom random number generators implemented are called Xorshift random

number generator. They generate the next number in their sequence by repeatedly taking the exclusive or (XOR) of a number with a bit shifted version of itself. Each of them can work very fast, and can produce one random number every clock. The random number sequence produced is determined by its initial setting value and the bits chosen to do the XOR operations.
As the Gaussian function of the algorithm needs two independent random numbers every time, the two Xorshift random number generators in each *Random_number_manager* are very different. They are given different initialization values, and the different bits of them are selected to do the XOR operations. This solution can make sure the random number sequences generated by them are not correlated.

The following sample VHDL program shows how one Xorshift random number works:

```
num_seed <= num_seed(0) & num_seed(15 downto 13) & (num_seed(0) xor
num_seed(12)) &
                num_seed (11 downto 8) & (num_seed(0) xor num_seed(7)) &
                                                num_seed(6 downto 2) &
(num_seed(0) xor num_seed(1));
```

On the other hand, the Gaussian look-up table is generated by Coregen using block memory resource. Its width is 32 bits, and depth is 1024. So, it consume one 36 Kbit block memory.
In the end, one FIFO is implemented to store the selected random numbers and related result of checking Gaussian table. As the suitable random numbers are produced very fast, this design can make sure the FIFO is always almost full of data for later calculation.

Gaussian Entity

Figure 6 shows the Gaussian entities implemented. And pipeline is applied to accelerate its calculations. As discussed above, the Gaussian Entity has two working modes as shown by Table 1. In mode one, the Gaussian Entity only needs to do one multiplication and one addition which can be finished with two steps. However, in mode two, there are four steps, including: reading the data from FIFO, three multiplications, and one addition. Additionally, there are always one or two of the three Gaussian components working at one mode, and the others works at another one. This may cause some problems on synchronizing the three Gaussian components. Additionally, it may make the output of later input processed by mode one be send out earlier than the earlier input processed by mode two. To solve this problem, working mode one is also extended to four steps to complete; the first two steps do nothing but pass the input data to the registers of next stage. By doing this, the two working modes need same amount of clocks to be finished, and the right output sequence is ensured. On the other hand, as the pipeline stages of different Gaussian components working at different modes align well, their synchronization and data exchange can work pretty well. As Fig. 6 shows, the *extra* variable produced at stage 2 by left Gaussian component at mode 2 can

directly feed to the stage 3 of right Gaussian component at mode 1. They are well synchronized and cooperate well.

Table 1: Different work mode of Gaussian component

	Mode 1	Mode 2
Step 1	N/A	Read data from FIFO
Step 2	N/A	Multiplication
Step 3	Multiplication	Multiplication
Step 4	Addition	Addition

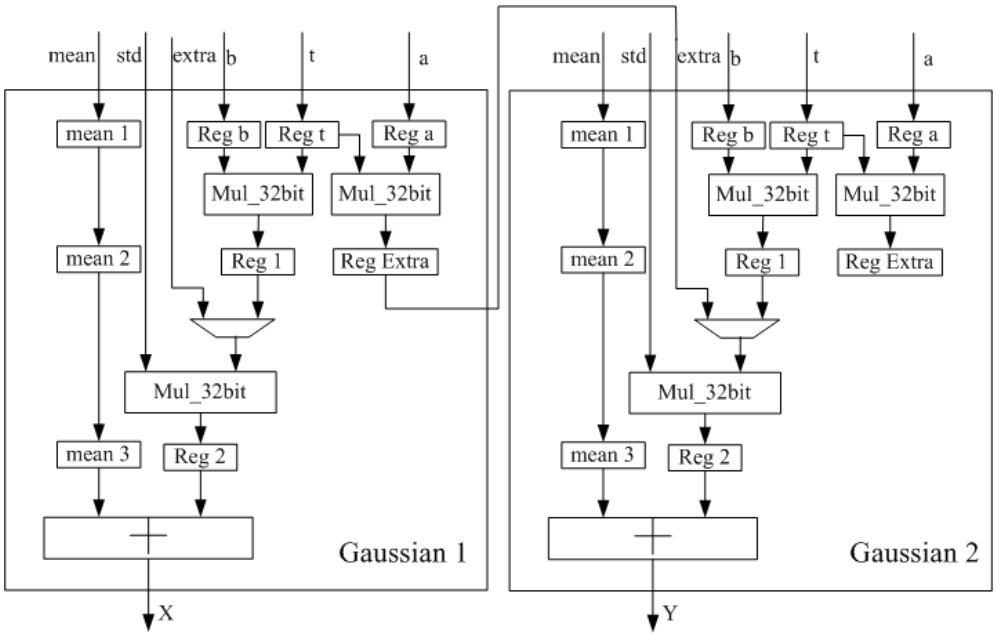


Figure 6: Diagram of two well synchronized Gaussian component

FIFO and look-up table

The system applied look-up tables to do Gaussian, sine, and cosine functions. On the other hand, FIFOs are used to buffer the output data of some components. All the FIFOs and look-up tables were generated by Coregen using block memory resource. Efforts were made to optimize these designs by careful configuration. For example, when generating 32 bits width and 512 depth sine table, if the default Minimum Area algorithm is chosen to concatenate the block RM primitives, it takes one 36K BRAM. But if 512 x 36 fixed primitives algorithm is applied instead, it only cost one 18K BRAM. On the other hand, it should be noticed that the same amount of BRAM resource is cost when generating the FIFO with different depth in some ranges. For example, both 16 depth and 512 depth of 51 bit width FIFO cost one 36K BBRAM. While the 512 depth FIFO takes full advantage of the BBRAM cost, the 16 depth one greatly wastes the BRAM

resources. Efforts were made to optimize the configuration when using Coregen in order to save the hardware resource while the performance is satisfied.

Hardware Simulation Results

The simulation of the program was performed by running the SLAM algorithm implementation both with and with out the APU coprocessor code implemented and using simulated sensor data as the input. The running environment was the Xilinx FX70 board. The run-times were obtained from clock cycle calculations and print statements to the minicom output. Both before and after coprocessor predict step averages are based on more than 400 data points each. The "Average Time in us" column describes the average time to execute the described step in microseconds. The "Present in Percentage of Runs" column describes the percentage of times the described step executed over all simulated sensor inputs.

The results of these simulations are shown in the following table.

Before APU Acceleration	Average Time in us	Present in Percentage of Runs
Predict Step	107,502	100.00%
Multiscan Step	2,487,969	2.17%
Filter Step	3,394	2.17%
After APU Acceleration		
Predict Step	12,784	100.00%
Multiscan Step	1,982,950	1.94%
Filter Step	13,291	1.94%

The results show that for the predict step there was an execution time reduction of 88.108%; 100% time reduction could only occur in the case that predict step takes no time to execute. The resulting acceleration for the whole program is 34.362%. This is considered significant acceleration for the predict step and the program in as a whole.

The results also show that the percentage of executions in comparison to number of input values by all three different steps is approximately constant between non-accelerated and accelerated implementations (within 0.25%). These results agree with the assumption that the program should operate in the same manner before and after the addition of the co-processor acceleration.

The simulation results also describe the multiscan feature extraction and data association step and the filer health evaluation and re-sample step. The average run-times for these two steps change between non-accelerated and accelerate implementations with the multiscan step accelerating by 20.29% and the filter step slowing by 74.46%. A possible reason for this is a difference in the output of the particle poses between the non-accelerated and accelerated implementations. Different value ranges could account for more computations to be performed on the particle poses in filter step, such as resampling, and fewer computations in the multiscan step. These variances do cause merit further investigation at a later time, but due to their small impact, present in only approximately 2% of total executions, their impact is viewed as minimal for the scope of this project.

Conclusions

For the VHDL design, pipelines and multi-thread techniques were successfully applied to accelerate the calculation particle pose for the SLAM algorithm by approximately 34% out of a possible 39%. The development team believes that this acceleration is significant, and multiple executions of the algorithm show this acceleration level to reliable. This was performed while saving hardware resource and meeting the performance is required; efforts were made to optimize the configuration when Coregen with generate look-up tables and FIFO components. This project shows that by using FPGA as the platform for running SLAM algorithm on, the algorithm can be run to required performance levels and accelerated to run efficently. Further research in the field could yield more efficient SLAM systems running on reconfigurable hardware.

References

1. Durrant-Whyte, Bailey, “Simultaneous Localization and Mapping: Part 1”, IEEE Robotics and Automation Magazine, June 2006, pg 99 – 108

2. Durrant-Whyte, Bailey, “Simultaneous Localization and Mapping: Part 2”, IEEE Robotics and Automation Magazine, September 2006, pg 108 - 117

3. Bonato, Peron, Wolf, Holanda, Marques, Cardoso, “An FPGA Implementation for a Kalman Filter with Application to Mobile Robotics”, Industrial Embedded Systems, 2007, pg 148 – 155

4. Bonato, Marques, Constantinides, “A Floating-point Extended Kalman Filter Implementation for Autonomous Mobile Robots”, Field Programmable Logic and Applications, 2007, pg 576-579

5. Beevers K.R., Huang, W.H., “SLAM with Sparse Sensing”, Robotics and Automation 2006, pg 2285-2290

Appendix

Source code can be downloaded via SVN at the link below:

https://source.ece.iastate.edu/projects/slam-f2011/