The Thesis committee for Thomas Charles Lauzon
certifies that this is the approved version of the following thesis:

Suitability of FPGA-based Computing for Cyber-Physical Systems

APPROVED BY

SUPERVISING COMMITTEE:

Derek Chiou, Supervisor

Aloysius Mok, Supervisor

Suitability of FPGA-based Computing for Cyber-Physical Systems

by

Thomas Charles Lauzon

THESIS

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN
December 2009

Dedicated to my parents, Véronique and Charles and my sister Catherine.

Suitability of FPGA-based Computing for Cyber-Physical Systems

Thomas Charles Lauzon, M.S.E.
The University of Texas at Austin, 2009

Supervisors: Derek Chiou
Aloysius Mok

Cyber-Physical Systems theory is a new concept that is about to revolutionize the way computers interact with the physical world by integrating physical knowledge into the computing systems and tailoring such computing systems in a way that is more compatible with the way processes happen in the physical world. In this master's thesis, Field Programmable Gate Arrays (FPGA) are studied as a potential technological asset that may contribute to the enablement of the Cyber-Physical paradigm. As an example application that may benefit from cyber-physical system support, the Electro-Slag Remelting process - a process for remelting metals into better alloys - has been chosen due to the maturity of its related physical models and controller designs. In particular, the Particle Filter that estimates the state of the process is studied as a candidate for FPGA-based computing enhancements. In comparison with CPUs, through the designs and experiments carried in relationship with this study, the FPGA reveals itself as a serious contender in the arsenal of

computing means for Cyber-Physical Systems, due to its capacity to mimic the ubiquitous parallelism of physical processes.

Keywords: Cyber-Physical Systems, Particle Filter, Electro-Slag Remelting, FPGA, CPU, Sequential Monte-Carlo Methods, Pipelining, Bluespec.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Research Context

1.1.1 Cyber-Physical Systems

The context of this research is the solution space exploration for the implementation of algorithms used in cyber-physical systems. Cyber-physical systems are the combination of control, information technology and communication. In such systems, timing is critical and processes evolve concurrently. Examples of systems that will benefit from this technology, some of which proposed by Lee [8], are:

Transportation

Automated shipyard management, advanced automotive systems, avionics, air traffic control;

Medical

Robotic surgery, assisted living, patient monitoring systems;

Utilities and infrastructure

Electric power, environmental control, water resource management, communications;

Physical virtuality

Telepresence, telemedicine;

Manufacturing

Process control, resource management;

Civil Engineering

Structure health monitoring, robotics assisted construction, smart structures.

1.1.2 Plant Model

The physical elements in a cyber-physical system may be modeled in two parts: a physical model, f and a measurement model, h. The physical model is a set of equations that describe the dynamics of the system, whereas the measurement model is a set of equations that describe the link between the system's state and the sensor signals. In most complex (i.e. high-fidelity) models these equations are often non-linear.

The physical model outputs a new state based on the command signals and the current state. Usually, noise is present in the command signals. The measurement model outputs a new observation vector from the new state from the physical model. The observation vector also contains measurement noise.

The basic form of the plant model is demonstrated in Fig. 1.1 .

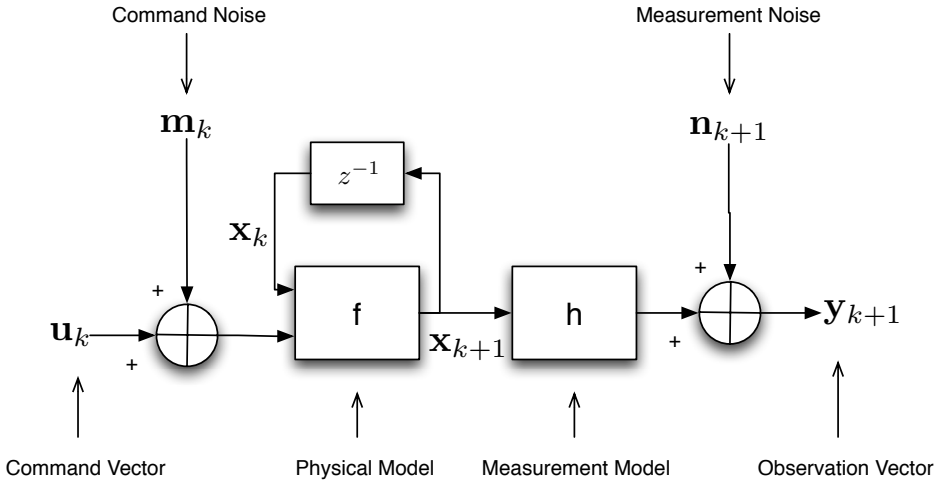


Figure 1.1: Basic Form of a Manufacturing Process.

1.1.3 Specific Applications: Gas Metal Arc Welding and Electro-Slag Remelting

As an example of a process that can benefit from the Cyber-Physical Systems paradigm, our research group has selected the Gas Metal Arc Welding (GMAW) process. This process is similar to a previously studied process called Electroslag Remelting (ESR.) The study of the ESR was performed by Ahn for his Ph.D. [1] . Although the speed of the GMAW process is much faster than the ESR, it is assumed that the general forms of the control and observation algorithms are similar. This research is therefore a follow-up of Ahn's research.

1.1.4 State Estimation for the Electroslag Remelting Process

One particular issue that arises in plants such as the ESR is the presence of noise on both the control signals and the observation signals. This is problematic for the purpose of controlling the process, since the state of the system may be quite different from the raw observations. If these raw observations were to be used directly, this would lead to improper process control. To resolve this problem, the state has to be inferred by a state estimator.

Due to the non-linear nature of the high-fidelity stochastic physical and measurement models of the ESR process, one estimation algorithm under investigation is the Particle Filter also known as Sequential Monte Carlo Methods (SMC.) Ahn's [1] study of estimation methods that inlcuded the Linear Kalman Filter, the Extended Kalman Filter and the Unscented Kalman Filter and the particle filter, has shown that although Kalman Filters provide good estimates in the Unscented Kalman Filter, the accuracy of these estimates doesn't match the accuracy of the Particle Filter for the ESR in terms of root mean squared error. This is due to the fact that if well designed and if the number of samples is large enough, the estimates tend toward the optimal Bayesian estimate.

1.1.5 Particle Filtering Applied to Electroslag Remelting

Particle Filtering is also known as Sequential Monte Carlo Methods. A detailed explanation of Particle Filtering would be beyond the scope of this document.

1.1.5.1 Overview of Particle Filtering

To understand the concept behind Particle Filtering, one may see each particle as a possible (or hypothetical) physical state of the plant. Additionally, each particle has a weight that represents how likely the plant may be in that state. The weights are normalized such that the sum of all the weights is equal to one. The particles with the heaviest weights are the ones that are most likely to be equal to the real plant state. After applying a certain command vector to the plant, simulations are done using the particles as initial states. These simulations provide new hypothetical states. The likelihood of each of these simulated states is obtained by further simulating their measurements and comparing their simulated measurements with the real measurements made on the real plant. The most likely states (i.e. the heaviest particles) are retained for the next round of simulations, hence the filtering. Ultimately, after a couple rounds, the particles should converge to the real state of the plant if the estimator is well designed.

1.1.5.2 Theoretical Background

The basic objective of Sequential Monte Carlo Methods is to estimate recursively in time the posterior distribution of a random variable and the expectations for some function of interest.

In the case of cyber-physical systems, the random variable that needs its posterior to be estimated is the current state of the physical system. Knowing the current state is necessary to command the system to the next step on the

trajectory that will lead the system into a desired final state. Typically, this signal is not directly available. It needs to be inferred from sensors, that may or may not directly sense the values that represent the state of the physical system. Furthermore, these sensed values often include significant noise.

Formally posed, the problem is stated as follows: [5]

The unobserved signal $\{x_t; t \in \mathbb{N}\}, x_t \in \chi$, is modeled as a *Markov Process* of prior distribution $p(x_0)$ and transition $p(x_t|x_{t-1})$. $\{y_t; t \in \mathbb{N}^*\}$, $y_t \in \Upsilon$ is a set of independent observations of the process with marginal distribution $p(y_t|x_t)$.

If we define $x_{0:t} = \{x_0, \ldots, x_t\}$ and $y_{1:t} = \{y_1, \ldots, y_t\}$ being all the signal values up to time t, *Bayes' theorem* gives at any time t:

$$p(x_{0:t}|y_{1:t}) = \frac{p(y_{1:t}|x_{0:t})p(x_{0:t})}{\int p(y_{1:t}|x_{0:t})p(x_{0:t})\mathrm{d}x_{0:t}} \tag{1.1}$$

This equation can be written in recursive form:

$$p(x_{0:t+1}|y_{1:t+1}) = p(x_{0:t}|y_{1:t})\frac{p(y_{t+1}|x_{t+1})p(x_{t+1}|x_t)}{p(y_{t+1}|y_{t+1})} \tag{1.2}$$

The marginal distribution, what we are looking for is obtained with these two steps:

Prediction:

$$p(x_t|y_{1:t-1}) = \int p(x_t|x_{t-1})p(x_{t-1}|y_{1:t-1})\mathrm{d}x_{t-1} \tag{1.3}$$

Updating:

$$p(x_t|y_{1:t}) = \frac{p(y_t|x_t)p(x_t|y_{1:t-1})}{\int p(y_t|x_t)p(x_t|y_{1:t-1})\mathrm{d}x_t} \tag{1.4}$$

The probability density functions can be discretized.

For further details, the reader may refer to *Sequential Monte Carlo Methods in Practice* by Arnaud Doucet, Nando De Freitas, Neil James Gordon, Neil Gordon. [5] The following description comes from this source.

1.1.5.3 Sampling Importance Resampling - Presentation

As stated in [2] the key idea behind Carlo Methods is to represent the posterior density function by a set of random samples with associated weights which can then be used to compute estimates. In [2], the observation vector $\mathbf{y}$ is called $\mathbf{z}$. In the literature, we find both uses, although usually $\mathbf{y}$ stands for the output, whereas $\mathbf{z}$ is the observation, which are often the same, but not necessarily. For instance $\mathbf{y}$ could be the raw data coming out of sensors, whereas $\mathbf{z}$ could be the interpreted value.

$$p(\mathbf{x}_{0:k}|\mathbf{z}_{1:k}) \approx \sum_{i=1}^{N} w_k^i \delta\left(\mathbf{x}_{0:k} - \mathbf{x}_{0:k}^i\right) \tag{1.5}$$

where the weights w_k^i are defined recursively as:

$$w_k^i \propto w_{k-1}^i \frac{p\left(\mathbf{z}_k|\mathbf{x}_k^i\right) p\left(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i\right)}{q\left(\mathbf{x}_k^i|\mathbf{x}_{k-1}^i, \mathbf{z}_k\right)} \tag{1.6}$$

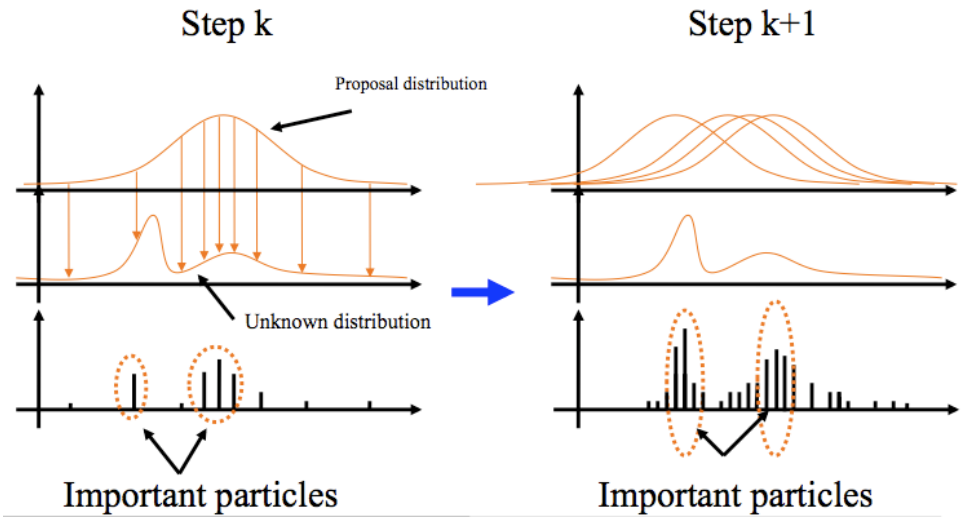


Figure 1.2: The Particle Filter: recursively improving the estimate of the posterior distribution by selecting the most important particles for the next iteration

The Sampling Importance Resampling (SIR) algorithm is the version that is used to for the Electroslag Remelting estimator. As the name suggest, it is composed of three steps:

Sampling This consist of drawing samples from the proposal distribution $\tilde{\mathbf{x}}_k^{(i)} \sim q\left(\mathbf{x}_k^{(i)} | \mathbf{x}_{k-1}^{(i)}, \mathbf{z_k}\right)$

Importance This is the computation of the samples weight using equation (1.6)

Resampling Which eliminates particles with small weights to concentrate on the most important one in order to reduce the *sample impoverishment* effect.

At each step, n simulations are done by adding noise to to the physical and measurement models. The likelihood of each observation is then computed based on the real plant observation. The particles that generated the most likely observations are kept and replicated for the next round of simulations. The number of replicates depends on the importance (high likelihood) of the particle.

1.1.5.4 Sampling Importance Resampling - Algorithm

As given in [2] and summarized in [1] the SIR particle filter algorithm is as follows:

$$\left\{x_k^{(i)}, w_k^{(i)}\right\}_{i=1}^{N} = ParticleFilter\left[\left\{x_{k-1}^{(i)}, w_{k-1}^{(i)}\right\}_{i=1}^{N}, \mathbf{z}_k\right]$$

- For $i = 1 : N$
  - Draw $\tilde{\mathbf{x}}_k^{(i)} \sim q\left(\mathbf{x}_k^{(i)} | \mathbf{x}_{k-1}^{(i)}, \mathbf{z_k}\right)$
  - Evaluate $w_k^{(i)} = p\left(\mathbf{z}_k | \mathbf{x}_{k-1}^{(i)}\right) w_{k-1}^{(i)}$
- For $i = 1 : N$
  - Normalize weights $\bar{w}_k^{(i)} = \frac{w_k^{(i)}}{\sum j{=}1^N w_k^{(j)}}$
- Calculate $N_eff = \frac{1}{\sum i{=}1^N \left(w_k^{(i)}\right)^2}$
- If $N_{eff} < N_{thr}$
  - $\left\{\mathbf{x}_k^{(i)}, w_k^{(i)}\right\}_{i=1}^{N} = \text{Resample}\left[\left\{x_k^{(i)}, w_k^{(i)}\right\}_{i=1}^{N}\right]$

- Else
  - $\left\{\mathbf{x}_k^{(i)}, w_k^{(i)}\right\}_{i=1}^{N} = \left\{x_k^{(i)}, w_k^{(i)}\right\}_{i=1}^{N}$

1.2 Problem to Solve

The initial problem we set ourselves to solve was the problem faced by Ahn. His implementation of the Particle Filter was unable to compute a sufficient amount of simulations under the process control deadline. Without studying the problem in detail, we opined that FPGA or ASIC would be a potential answer to this computation speed problem.

1.3 Preliminary Study: Influence of the Quantity of Particles on the Precision of the Estimate

An intuitive idea is that to improve the state estimates one only needs to increase the number of simulations. As a preliminary study, the influence of the number of particles on the precision of the estimates was studied in order to set a performance goal for the ESR estimator. The result of this study is that there is a limit to the maximum precision that can be achieved. In other words, that mean that there is a threshold after which increasing the number of particles has no effect on the precision of the estimates. Furthermore, a reassuring result is that increasing the number of particles (with the given models) always reduces the error of the estimate. While some state variable estimates are poorly sensitive to the number of particles others steadily improve up to 200

particles. After 200 particles, the improvements in precision are negligible (for this set of physical and measurement model.

1.3.1 Experiments, Analysis and Results

1.3.1.1 Evaluation Means

In order to evaluate the performance of the estimator, the root mean squared error (RMSE) between the true state of the plant (which is directly available from the plant model) and the estimated state is computed for each state variable. The objective is to have the smallest RMSE possible.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (\mathbf{x}_{true,i} - \mathbf{x}_{estimated,i})^2}{n}} \tag{1.7}$$

The estimator was tested in open loop configuration with no controller. The process and measurement noise were from a gaussian distribution with parameters that were measured by Ahn at the plant.

1.3.1.2 Experimental Setup

These results were computed using the C implementation. Using the original implementation under Matlab would yield the same results.

Three types of resampling techniques were used.

- Systematic resampling
- Residual resampling

- Multinomial resampling

This simulation was run over 1000 seconds divided in 7500 steps and repeated 10 times for statistical results.

The RMSE was computed for different numbers of particles for each state variable.

1.3.1.3 Results

The results are found in figure (1.3) to (1.7)

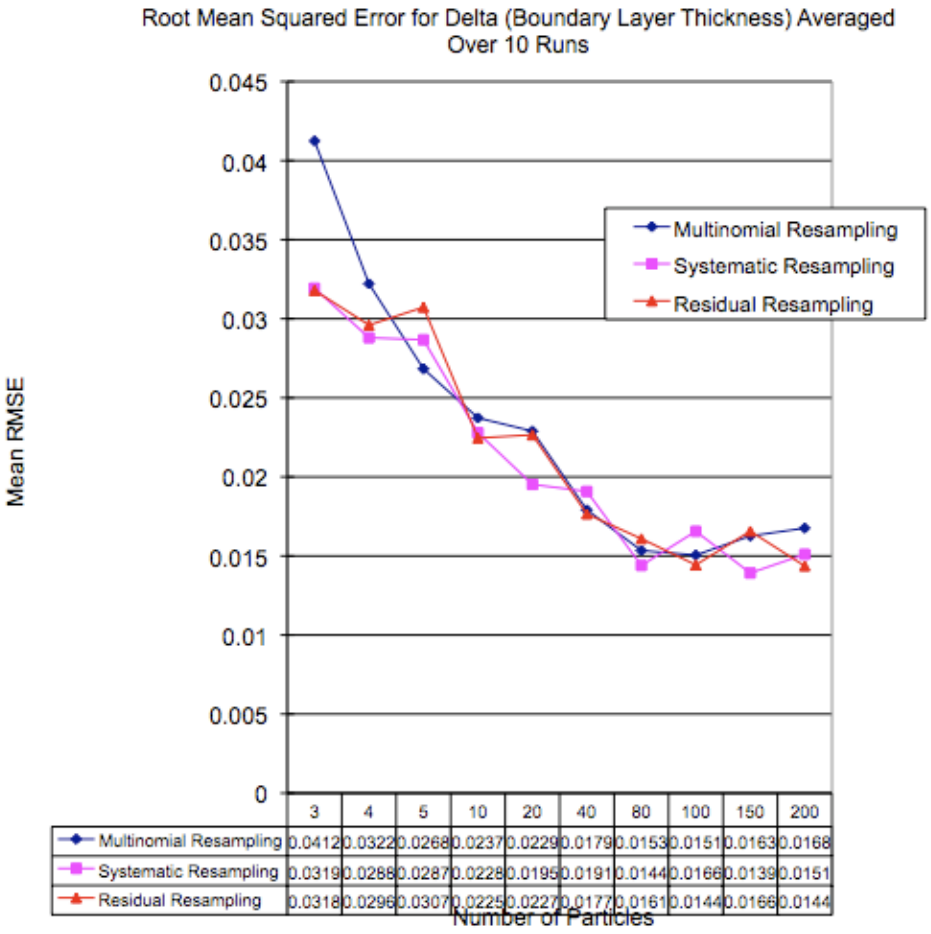


Figure 1.3: RMSE Δ

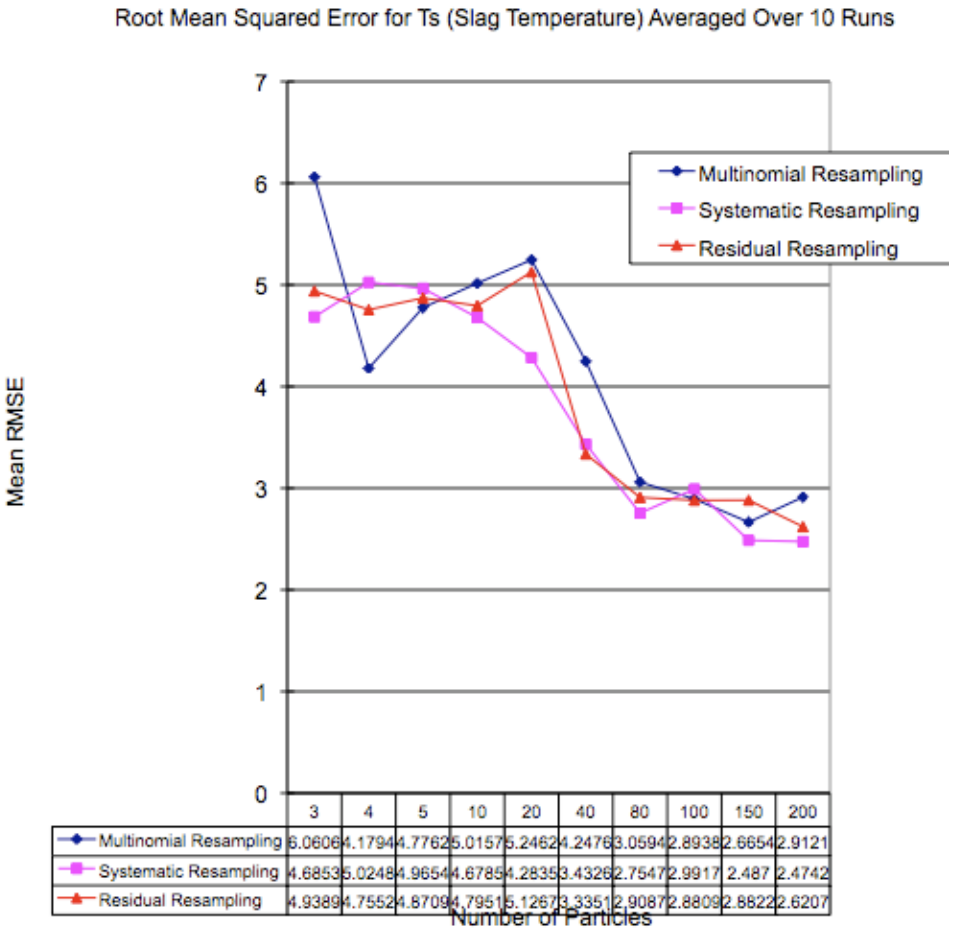
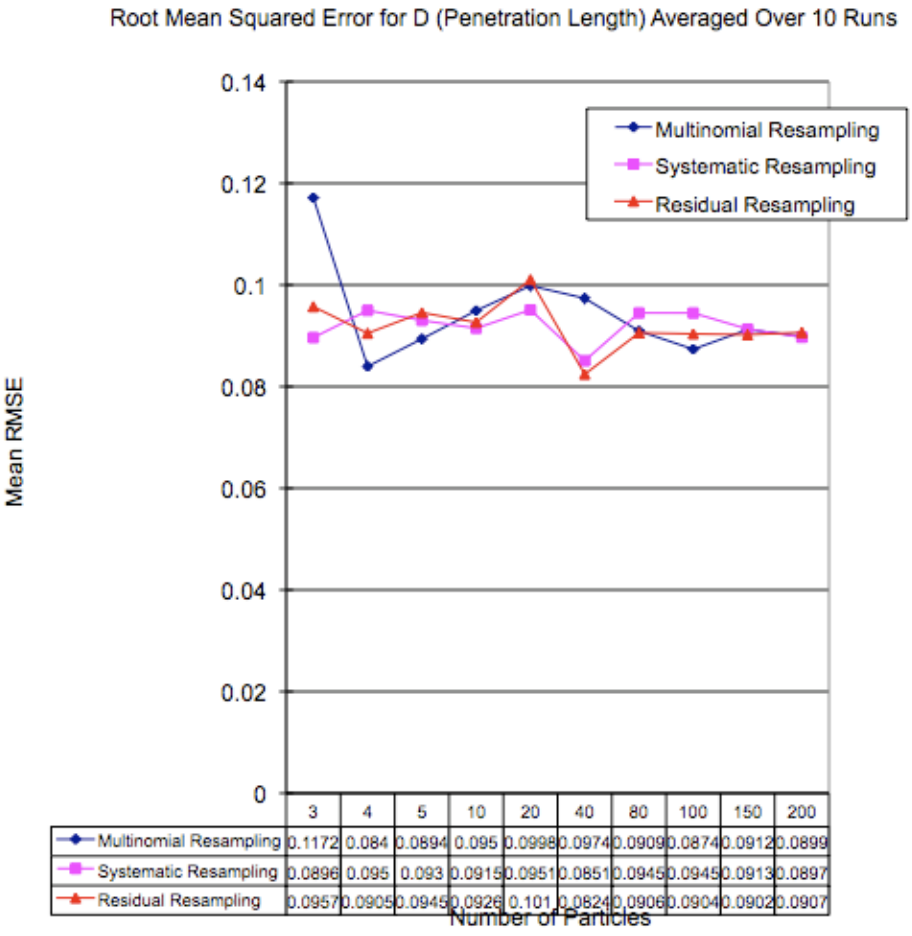
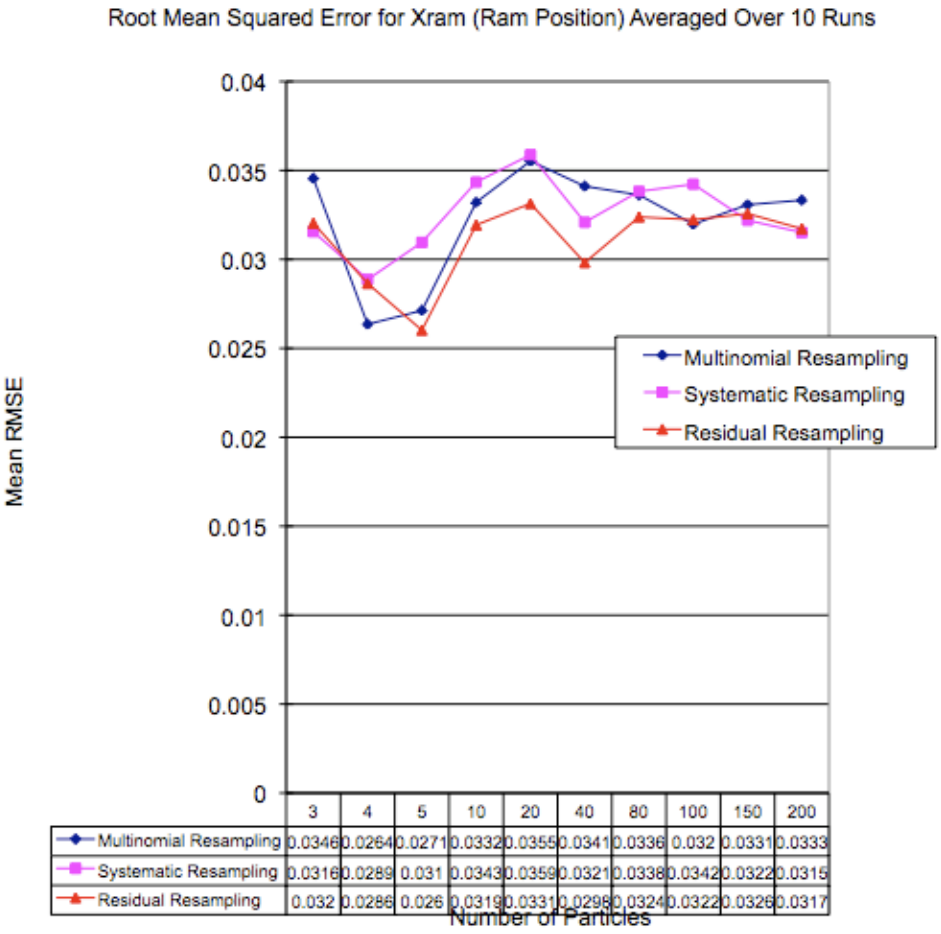


Figure 1.4: RMSE T_s



	3	4	5	10	20	40	80	100	150	200
Multinomial Resampling	0.1172	0.084	0.0894	0.095	0.0998	0.0974	0.0909	0.0874	0.0912	0.0899
Systematic Resampling	0.0896	0.095	0.093	0.0915	0.0951	0.0851	0.0945	0.0945	0.0913	0.0897
Residual Resampling	0.0957	0.0905	0.0945	0.0926	0.101	0.0824	0.0906	0.0904	0.0902	0.0907

Figure 1.5: RMSE d



	3	4	5	10	20	40	80	100	150	200
Multinomial Resampling	0.0346	0.0264	0.0271	0.0332	0.0355	0.0341	0.0336	0.032	0.0331	0.0333
Systematic Resampling	0.0316	0.0289	0.031	0.0343	0.0359	0.0321	0.0338	0.0342	0.0322	0.0315
Residual Resampling	0.032	0.0286	0.026	0.0319	0.0331	0.0298	0.0324	0.0322	0.0326	0.0317

Figure 1.6: RMSE X_{ram}

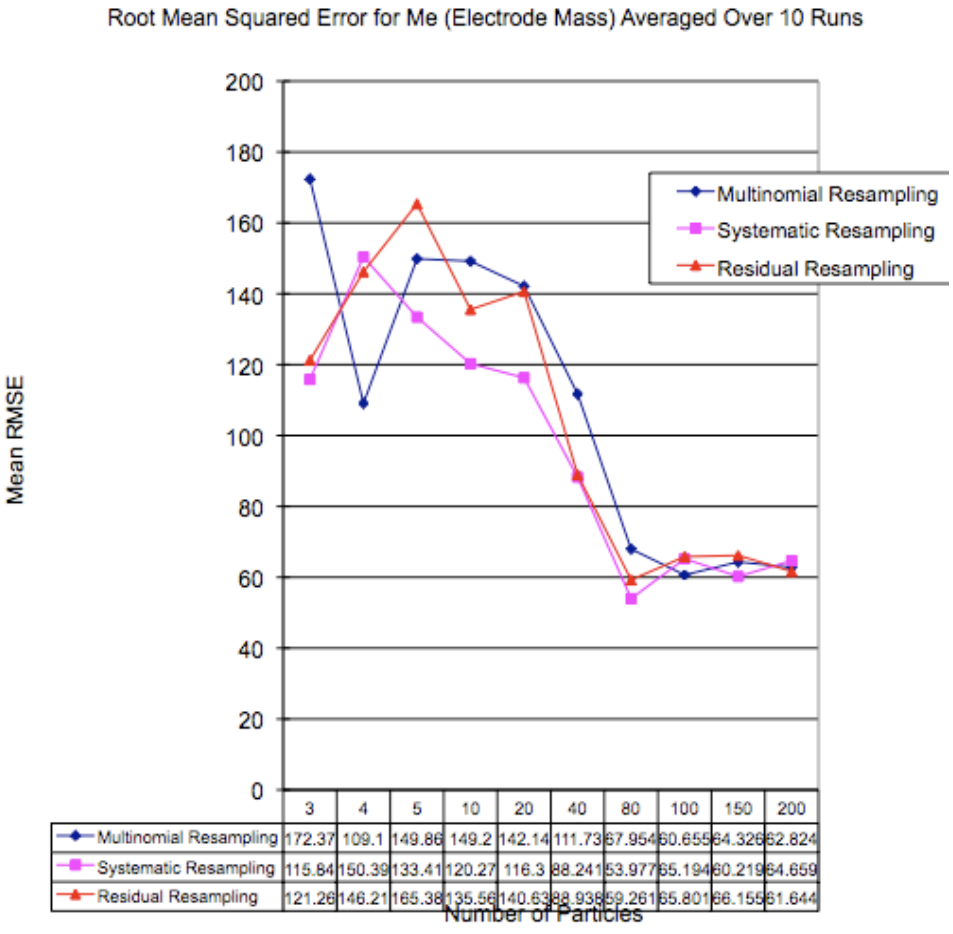


Figure 1.7: RMSE M_e

1.3.1.4 Conclusion

From these experiments it is concluded that 200 particles is the threshold after which the estimates have reached their maximum precision.

1.3.2 Summary of the Problem to Solve

Based on the preliminary experimental results and the fact that the given sampling rate is 133 ms the goal to achieve is the computation of 200 particles under 133ms.

1.4 Thesis Statement

Based on the fact that the physical world is ubiquitously parallel, i.e. processes occur concurrently, it seems that parallel computing devices would be the best suited devices for real-time inline simulation of the physics of a cyber-physical system. Supporting this argument is the fact that modern estimation techniques, such as Particle Filtering, make use of independent probabilistic sampling. Since the sampling is independent, it can be done in parallel, bolstering the use of parallel devices, such as FPGAs. Although modern CPUs are now multicore and work at Gigahertz frequencies, FPGAs appear to be a better platform due to their intrinsic parallel architecture.

An FPGA based solution would take advantage of the fact that not only the physical simulations done in Particle Filtering are independent and can be done in parallel but also of the fact that the physical processes within the plant occur concurrently themselves. As a result, a significant amount of parallelization is possible, for which FPGAs are well suited.

1.5 Objectives

The objective is to figure out how to evaluate the performance of computing devices for the purpose of assisting the task of designing cyber systems that provides adequate computing power for controlling a specific physical process. We have identified and selected three types of computing devices that differ in nature and present potential applicability to the domain of cyber-physical systems. The first type is the Central Processing Unit (CPU) type, the second is the Graphical processing Unit (GPU) type (not discussed in this Master's Thesis) and the last one is the Field Programmable Gate Array (FPGA) type. These types may be characterized by the parallelism present in their computing architecture. CPUs present the lowest order of parallelism. Current CPUs on the market typically offer 4 or 8 fast cores that can be used in parallel. In contrast, GPUs provide a higher number of parallel processing units, in the order of the hundreds. On the other end of the spectrum, FPGAs are inherently parallel devices which can be tailored to concurrently process a large number of elementary operations.

1.6 Approach

Using a well known computer controlled manufacturing process, Electro-slag Remelting (ESR), an application that is mature for experimentation, we designed an FPGA based solution and compared its performance with that of a reference CPU based implementation. From both implementation we derived computation speed models that give a macro level view of the performance

attainable on both platforms.

Chapter 2

Study of the Particle Filtering Algorithm as Applied to Electro-slag Remelting

In this chapter, the algorithm is described in the special case of the Electroslag Remelting process. After this, the proposed method for accelerating the algorithm through the use of parallelization is presented, followed by a study of the data and data types and how this is a potential barrier to implementation and speedup potential

2.1 Particle Filter Algorithm for the Electro-slag Remelting Process

In the previous chapter, the general form of the particle filter has been introduced. However, this form is somewhat abstract. In fact, the question is "how do we draw particles?" As said earlier, the particles are drawn from a proposal distribution $q\left(x_k|x_{k-1}, z_k\right)$. Since this distribution is difficult to obtain, the transition prior $q\left(\mathbf{x}_k|\mathbf{x}_{k-1}\right)$ is chosen as a proposal distribution. This distribution is only dependent on the previous state $\mathbf{x}$. In many cases this is acceptable.

We would like to be able to sample from the transition prior by run-

ning a particle through the physical model and adding process noise to the new state. The problem is that it is difficult to know the parameters of the distribution of $\mathbf{x}$ since we cannot measure it directly by experiment. Instead, particles are drawn by adding noise to the control vector, $\mathbf{u}$, running those noisy input values through the physical model and the measurement model, and then adding the measurement noise. The resulting values are the particles.

$$\mathbf{x}_k^{(i)} \sim q\left(\mathbf{x}_k^{(i)} | \mathbf{x}_{k-1}^{(i)}\right) \sim f\left(\mathbf{x}_{k-1}^{(i)}, \mathbf{u}_k + \mathbf{v}_k^{(i)}\right) \tag{2.1}$$

where: $\mathbf{x}_k^{(i)}$ is the particle (i) at step k, f is the physical model, $\mathbf{u}_k$ is the command vector at step k and $\mathbf{m}_k$ is the generated process noise vector at step k for particle i. This noise may come from any distribution.

The next step is to evaluate the weights. This is done recursively by multiplying the previous weights by the likelihood of each observation. The likelihood function in the case of the ESR is a gaussian.

For each particle, the simulated observation is:

$$z_k = h\left(\mathbf{x}_k\right) + \mathbf{n}_k \tag{2.2}$$

where: h is the measurement model, $\mathbf{x}_k$ is the drawn state and $\mathbf{n}_k$ is the measurement noise vector. The noise vector is random and different for each particle. As can be seen from this equation, the observation is a direct function of the state and the noise.

The likelihood is then:

$$p\left(z_k|x_k^{(i)}\right) \propto e^{\left(-\frac{1}{2}\left(z_k-z_k^{(i)}\right)^t R_r\left(z_k-z_k^{(i)}\right)\right)} \tag{2.3}$$

R_r is the covariance matrix, z_k is the observation from the plant at step k, $z_k^{(i)}$ is the simulated observation for particle i.

$$w_k^{(i)} = p\left(z_k|x_k^{(i)}\right) w_{k-i}^{(i)} \tag{2.4}$$

$w_k^{(i)}$ is the weight of particle i at step k, $w_k^{(i)}$ is the weight of particle i at step *k-1*.

Once all the weights have been computed, they must be normalized so that the sum of the weights is equal to 1.

The last part is the resampling of the particles. This part is not covered in detail in this document in detail as it is not the target of this study. The basic mechanism is that the particles that are the most important (highest weights) are selected for the next step. The reader may consult [2] for further reference.

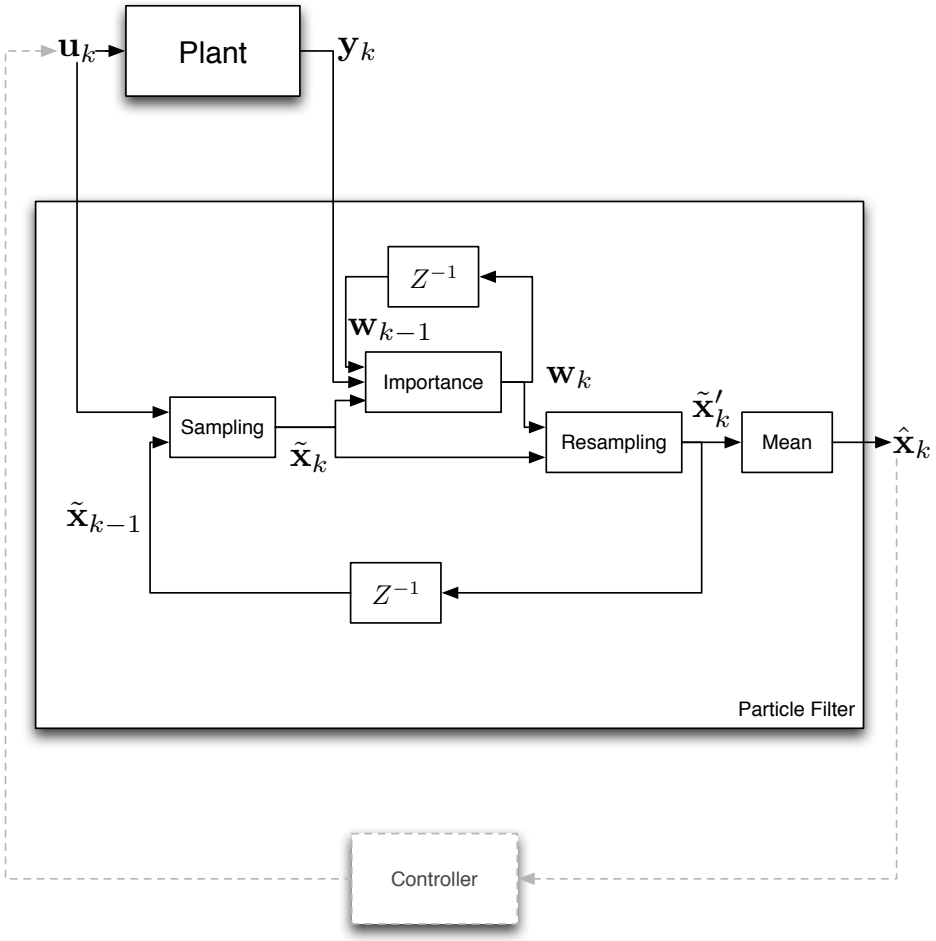


Figure 2.1: General Form of the Particle Filter Based Estimator.

2.2 Parallelization Potential

One specificity of the Particle Filter is that the sampling part can be parallelized since each particle is drawn independently from another. It is sup-

posed that the sampling part may be accelerated through parallelization on an FPGA for example. Not only can the sampling of each particle be done in parallel, many of the internal computations, which are the physical and measurement models of the concurrent processes, can also be done in parallel. As seen in the estimator diagram, the likelihood function can also be computed in parallel. But the parallelization potential stops here, as the next step (normalization) requires all the weights from all the particles. Finally, the resampling may potentially benefit from internal parallelization, but this study has not been done yet and it is believed at this point that the speed improvement from this parallelization will not be as significant as the parallelization of the sampling stage, the mechanism of which remains the same independently of the physical and measurement models used. The equations for the models come from Ahn's Dissertation [1].

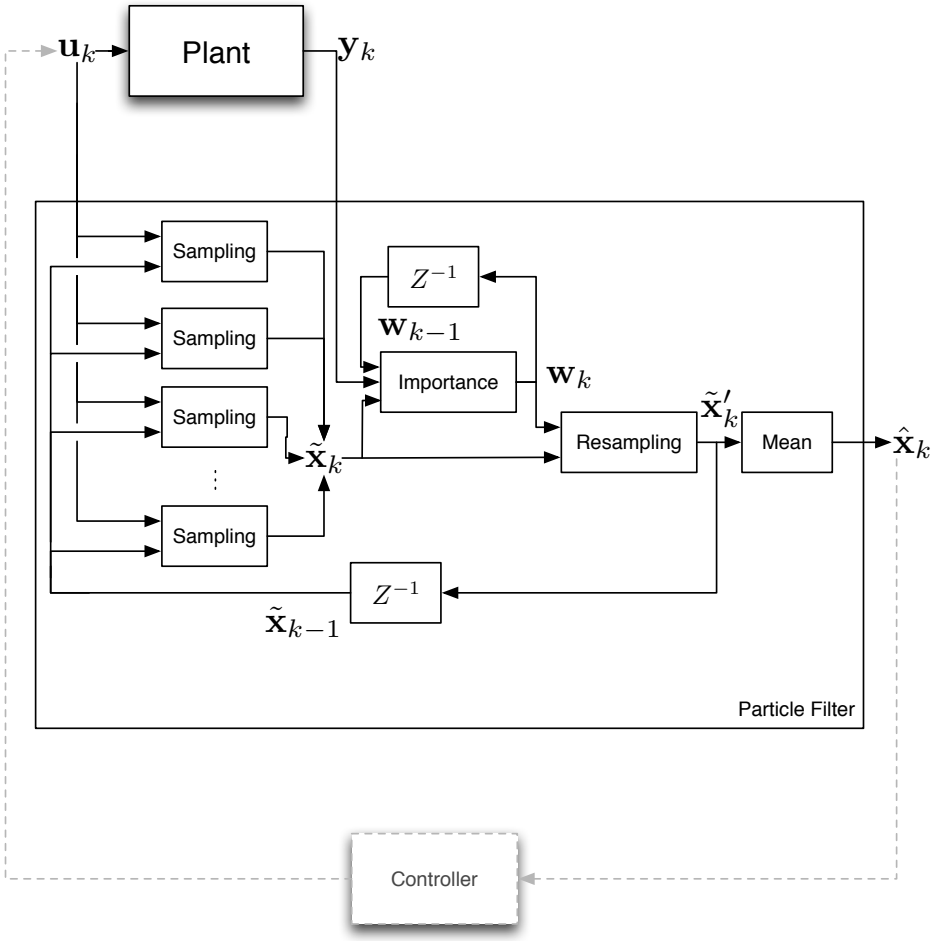


Figure 2.2: Particle Filter with Multiple Sampling Stages in Parallel

2.2.1 Physical Model

The physical model is a set of differential equations that describe the dynamics of the process. Here are the equations of the model for the Electro-slag Remelting process.

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{\Delta} \\ \dot{T}_s \\ \dot{d} \\ \dot{X}_{ram} \\ \dot{M}_e \end{bmatrix} = f(\mathbf{x}, \mathbf{u}) = \begin{Bmatrix} \frac{C_{\Delta\Delta}\alpha_r}{\Delta} - \frac{C_{\Delta p}}{h_m} p_{m,r}(T_s) \\ \frac{1}{\rho_s \nu_s C_s} \{p_{in,r}(T_s, d, I_c) - Q_m(T_s) - Q_s(T_s)\} \\ \frac{C_{s\Delta}\alpha_r}{\Delta} - \frac{C_{sp}}{h_m} p_{m,r}(T_s) + \frac{1}{a}(V_{ram,c} + V_{ram,b}) \\ V_{ram,c} + V_{ram,b} \\ -\rho A_e \dot{s}(\Delta, T_s) \end{Bmatrix} \tag{2.5}$$

where:

$$\begin{cases} \dot{s}(\Delta, T_s) = -\frac{\alpha_r C_{s,\Delta}}{\Delta} + \frac{C_{sp}\, p_{m,r}(T_s)}{h_m} \\ p_{m,r}(T_s) = (1 + \mu_r)\frac{Q_m(T_s)}{A_e} \\ Q_m(T_s) = H_e A_e (T_s - T_m) \\ Q_s(T_s) = H_s 2\pi r_i h_{s0}(T_s - T_{ss}) \\ p_{in,r}(T_s, d, I_c) = Volt(T_s, d, I_c) I_c \\ Volt(T_s, d, I_c) = R(T_s, d) I_c \\ R(T_s, d) = R_d(d) e^{(} - A_{elect}(T_s - T_{s^*}) \\ R_d(d) = \begin{cases} R_1 - m_0 d & d < d_i nflection \\ R_1 - m_1 d & d \geq d_i nflection \end{cases} \end{cases}$$

While it is important for the the design of these models to know what all these names mean, for the problem of studying the computation what they stand for is insignificant. It is however important to realize that all the values that are not part of the state vector $\mathbf{x}$ or the command vector $\mathbf{u}$ are constants, i.e. the state vectors contain all the variables of these equations. All other values are parameters that can be pre-computed.

The discrete version is obtained by integrating $f(\mathbf{x}, \mathbf{u})$ over the sampling period T.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + f(\mathbf{x}_k, \mathbf{u}_k).T \tag{2.6}$$

Finally, the command noise $\mathbf{m}_k$ is added to the command vector, yielding:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + f\left(\mathbf{x}_k, \mathbf{u}_k + \mathbf{m}_k\right).T \tag{2.7}$$

2.2.2 Measurement Model

In the case of the electroslag remelting model, the measurement equations are rather simple. The output vector $\mathbf{y}$ contains the penetration depth d, the ram position X_{ram}, the current I_r, the weight read on the load cell LC and the voltage $Volt$

$$\mathbf{y} = h(\mathbf{x}) = \begin{Bmatrix} d \\ X_{ram} \\ I_c \\ LC \\ Volt \end{Bmatrix} \tag{2.8}$$

where:

$$\begin{cases} R = R_d e^{(-A_{elect}(T_s - T_{s^*})} \\ R_d = \begin{cases} R_1 - m_0 d & d < 0 \\ R_1 - m_1 d & d > 0 \end{cases} \\ LC = \begin{cases} M_e - \rho_s A_e d & d > 0 \\ M_e & \text{otherwise} \end{cases} \\ Volt = RI_c \end{cases}$$

Finally, the measurement noise $\mathbf{n}_k$ is added to the command vector, yielding:

$$\mathbf{y}_k = h\left(\mathbf{x}_k\right) + \mathbf{n}_k \tag{2.9}$$

2.2.3 Likelihood Function

The weights are finally computed through the use of the likelihood function based on the simulated (drawn) observations. The likelihood function is the following:

$$l = e^{-\frac{1}{2}\left(\mathbf{y_{plant}}-\mathbf{y_{particle}}\right)^T \Sigma^{-1}\left(\mathbf{y_{plant}}-\mathbf{y_{particle}}\right)} \tag{2.10}$$

Chapter 3

Implementation

A working version of the estimator for the ESR process has been written under Matlab by Ahn. Both CPU and FPGA perform the same functions as this Matlab code. In fact the Matlab results were used to validate both implementations. The CPU implementation was done in C, whereas the FPGA implementation was done using Bluespec System Verilog.

3.1 CPU implementation in C

As discussed in the introduction of this chapter, the CPU implementation was done in C. The primary reason for this choice is the fact that many modern real-time control systems using CPUs use this language due to its simplicity and its robustness. Although C doesn't natively provide many types (e.g. no fixed-point type, which can be useful for speeding up some of the computation,) C natively supports double precision data types and comes with many mathematical functions such as the exponential function, which is required to compute the resistance values in the ESR's models. In order to verify that the C code performs the same tasks as the Matlab code, the Process and Measurement noise have been stored in text files that are both accessible

by Matlab and C code. In the end it was verified that the C implementation provides the same results as the original Matlab code.

3.1.1 Code

Only the physical model, f, and the measurement models are included, since we are only studying the sampling part.

```
void f   (
         /*IN:*/
         double x[SIZEOFX],
         double u[SIZEOFU],
         double noise[SIZEOFU],
         double ts,
         /*OUT:*/
         double newX[SIZEOFX],
         double * I
         )
{
         double tdelta,Rd,R,Volt,P,Qm,pm,Qs,Vram,
         ... Sdot,deltadot,Tsdot,ddot,Xramdot, Medot;

         tdelta=ts;

         if(X_D<dInflection)
                  Rd=R1-m0*X_D;
         else
                  Rd=R1-m1*X_D;

         R=Rd*exp(-Aelect*(X_TS-Tsstar));
         *I=U_IC+Ib+noise[0];
         Volt=R* *I+Voltb;
         P=Volt* *I;
         Qm=He*Ae*(X_TS-Tm);
         pm=(1+mur)*Qm/Ae;
```

```
        Qs=Hs*2*M_PI*ri*hs0*(X_TS-Tss);
        Vram=U_VRAMC+Vramb+noise[1];
        Sdot=-alphar*Csd0/X_DELTA+Csp*pm/hm;


        //rate equations
        deltadot = alphar*Cdd/X_DELTA-Cdp*pm/hm;
        Tsdot = (P-Qm-Qs)/rhos/Vs/Cs0;
        ddot = alphar*Csd0/X_DELTA - Csp*pm/hm + Vram/a;
        Xramdot = Vram;
        Medot = -rhom*Ae*Sdot;

        //Output
        newX[0] = X_DELTA + deltadot*tdelta;
        newX[1] = X_TS + Tsdot*tdelta;
        newX[2] = X_D + ddot*tdelta;
        newX[3] = X_XRAM + Xramdot*tdelta;
        newX[4] = X_ME + Medot*tdelta;
}

void h  (
        /*IN:*/
        double x[SIZEOFX], double u[SIZEOFU],
        ... double noise[SIZEOFY],
        /*OUT*/
        double y[SIZEOFY]
        )
{
        double Rd,Ir,R,Volt,LC;

        if(X_D<0)
                Rd = R1-m0*X_D;
        else
                Rd = R1-m1*X_D;

        Ir = U_IC + Ib;
        R = Rd*exp(-Aelect*(X_TS-Tsstar));
```

```
        //estimated resistance
        Volt = R*Ir + Voltb;

        if(X_D>0)
                LC = X_ME - rhos*Ae*X_D;
        else
                LC = X_ME;

        y[0]=X_D+noise[0];
        y[1]=X_XRAM+noise[1];
        y[2]=Ir+noise[2];
        y[3]=LC+noise[3];
        y[4]=Volt+noise[4];
}
```

3.2 FPGA Implementation

3.2.1 FPGA technology

Field Programmable Gate Arrays (FPGA) are device that can be configured to perform sequential logical functions. FPGAs are the bigger type of logic devices in the programable device family. One of the advantages of FPGA is that the logic architecture can be designed specifically for a certain application, in particular, the architecture can be designed such that several computations are made in parallel, which can create a significant speedup in comparison with CPUs. One of the drawbacks is that an FPGA is more complicated to program compared to a CPU.

In our case, we hope to achieve a significant speedup through the parallelization of the sampling process. As discussed earlier, the sampling part

of the algorithm can be done in parallel due to the fact that the samples are independent.

3.2.2 ESL tool: Bluespec System Verilog

Electronic System Level design. ESL design is the creation of hardware from an algorithmic description. Bluespec has been identified as a tool that has a level of abstraction that is close to the algorithmic description, whilst at the same time being capable to give the designer control over the architecture of the hardware. Bluespec is capable of generating Verilog code that is directly usable in synthesis tools such as Xilinx ISE.

The first problem to consider when designing a system at the Electronic System Level is the set of operators that are available. At its most basic level, the FPGA is capable of performing logic operations on bits. On Xilinx FPGA this is done through the use of 6 bit Look-up tables (LUT) . Xilinx FPGAs also feature DSP48 multipliers which are 48×48 bit multipliers that return a 48 bit result. The Bluespec language provides additional operators, mostly for bits, or sets of bits such as integers.

The models, however, require rational numbers. Two options exist to represent rational numbers: fixed-point or floating point representation. Two points give fixed-point the advantage over floating-point. The first point is that fixed-point operations usually require less hardware, therefore are faster and require less physical resources. The second is that Bluespec already comes with a synthesizable fixed-point library whereas it currently provides no support for

IEEE 754 floating point natively.

The Bluespec fixed point library [3] handles additions, subtractions and multiplications. It also provide function for bit field width extension or truncation and a multiplication function that returns data with the proper bit width. Unfortunately, the division and the exponential functions required for the algorithm are missing. They need to be implemented. On the upside, this allows for performance tradeoffs.

3.2.3 FPGA Implementation Techniques

For the FPGA implementation, it is convenient to have a flow graph representation of the algorithm. The algorithm is separated in two parts: the computation of the physical model F and the measurement model H. Both models contain stages, which are separated by doted line in the diagrams. Operators within the same stage function in parallel. Furthermore, the design is made to operate in a pipelined fashion, therefore each stage is a stage of the pipeline. One final important detail that is absent in the figures for the sake of overcrowding the illustration are the memory queues between operators separated by multiple stages. The depth of these queues are naturally to be equal to the number of stages separating the operators minus one.

It is essential to note that the main operations natively provided are logic operations (AND, OR, NOT...) and 48 bit multiplications (on the Xilinx Virtex V FPGA). Any other operation needs to be provided by libraries or custom made. The first technique used is the pre-computation of parame-

ters. Unlike C in which the compiler precomputes constant parameters (such as constant resistance values) automatically, parameters in the FPGA modules should be precomputed to avoid using operators unnecessarily. Furthermore, a nice feature of performing the pre-computation of parameters is that complicated operations such as divisions are avoided, saving many cycles and hardware resources.

The second technique is the use of non native operations in the form of modules. For the physical and measurement models of the ESR, two non native operations are required: divisions and exponentiations. These operations can be implemented in the form of modules. These modules may require several cycles to complete their functions. When this is the case, the length of the stage that contains such modules is equal to the length of the longest module - or longest chain of modules - in terms of number of cycles.

3.2.4 Mixed control and data flow chart of the Algorithm

3.2.4.1 Physical Model (F)

The ESR's physical model has been divided in ten stages. While eight of these stages last only one cycle, two of them require multiple cycles due to the use of non-native operations. These non-native operations are presented later. The physical model computes the rates of variation of each state variable (i.e. Δ, T_s,d, X_{ram} and M_e) based on the current state and the command vector. Additionally the command noise is added to the command vector within this modules, although this could have alternatively been done externally.

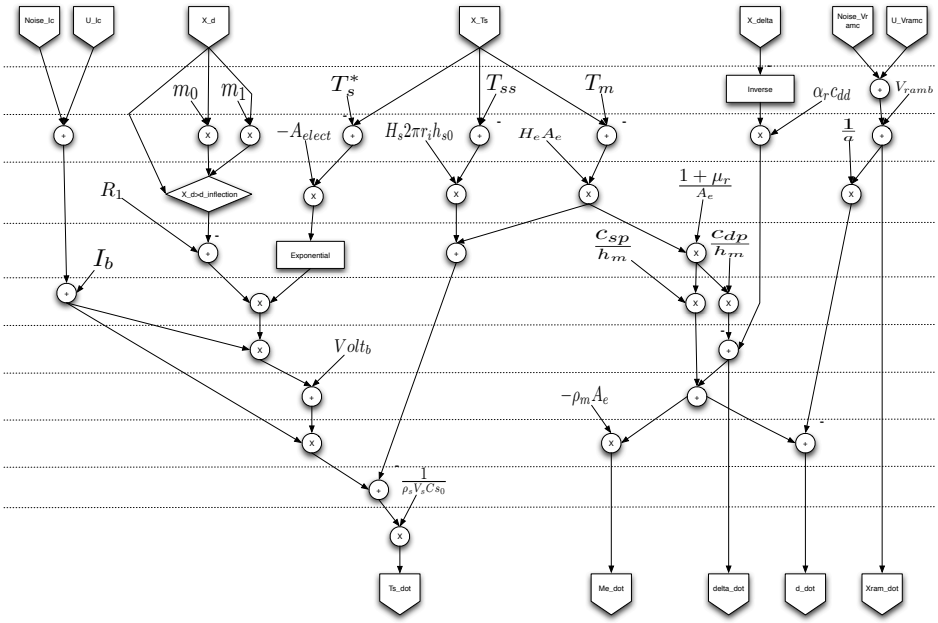


Figure 3.1: Physical Model Flow Chart

For the FPGA implementation it is interesting to decompose the physical model of the ESR plant into three parts: inversion, dynamics and integration. The reason for this separation from the rest of the physical equations is that it takes many more steps to compute the inversion and during this time, some of the computations can be done in parallel so that once the inversion is complete the rest of the computations can be completed in a fully-pipeline fashion, with a result at each cycle. In order to keep the pipeline full, the number of copies of parallel inverters need to be equal to the number of step it take to complete an inversion. For the sake of comprehension, we will call the physical model without the inversion the dynamics model, for lack of a better

term.

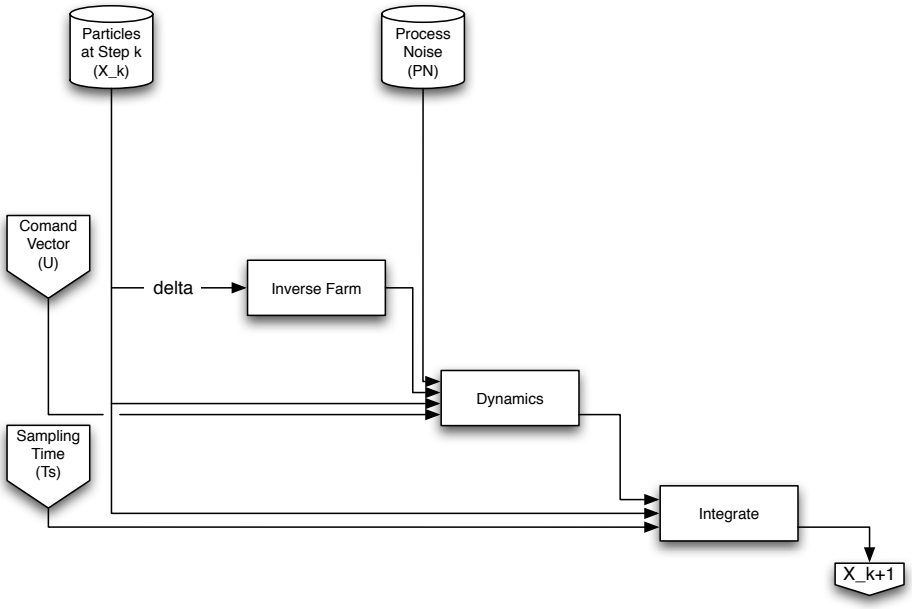


Figure 3.2: Full Physical Model Flow Chart

The resulting dynamics model can be found in figure 3.3.

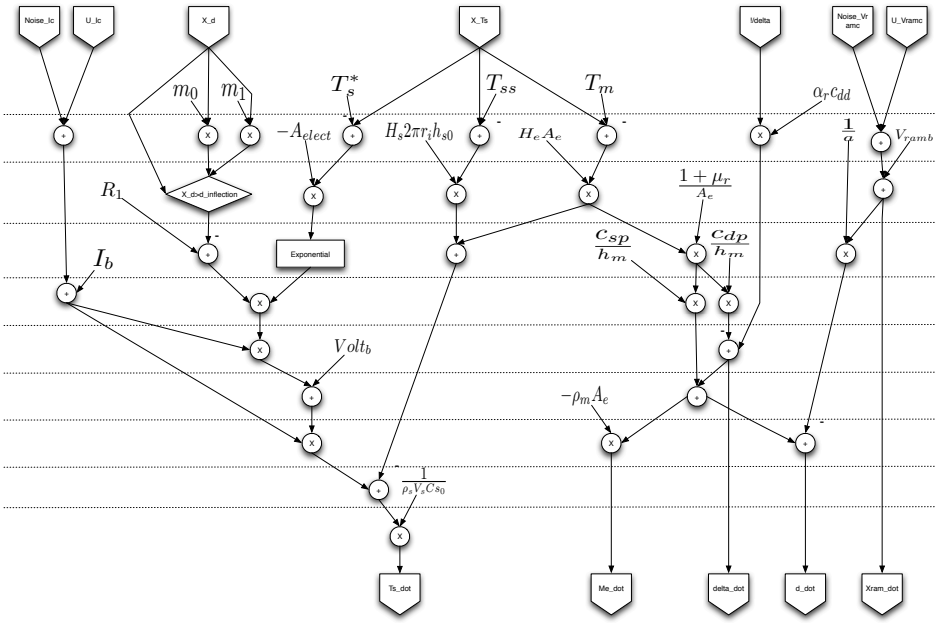


Figure 3.3: Physical Model Flow Chart - Inversion Excluded

3.2.4.2 Measurement Model (H)

The measurement model is rather straightforward compared to the physical model. The only non-native operation is an exponential used for the computation of the ESR's resistance. The observation vector is computed from a state. Noise is added to the measurement in the model.

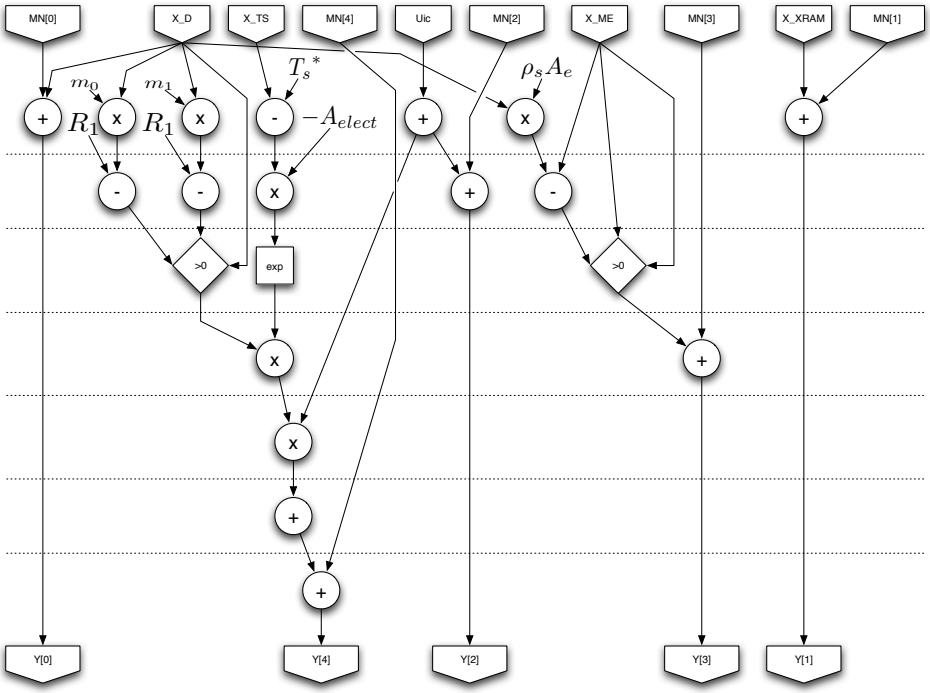


Figure 3.4: Measurement Model Flow Chart

3.2.4.3 Whole Simulator

The whole simulator includes the physical and measurement models. Figure 3.5 shows how they are connected.

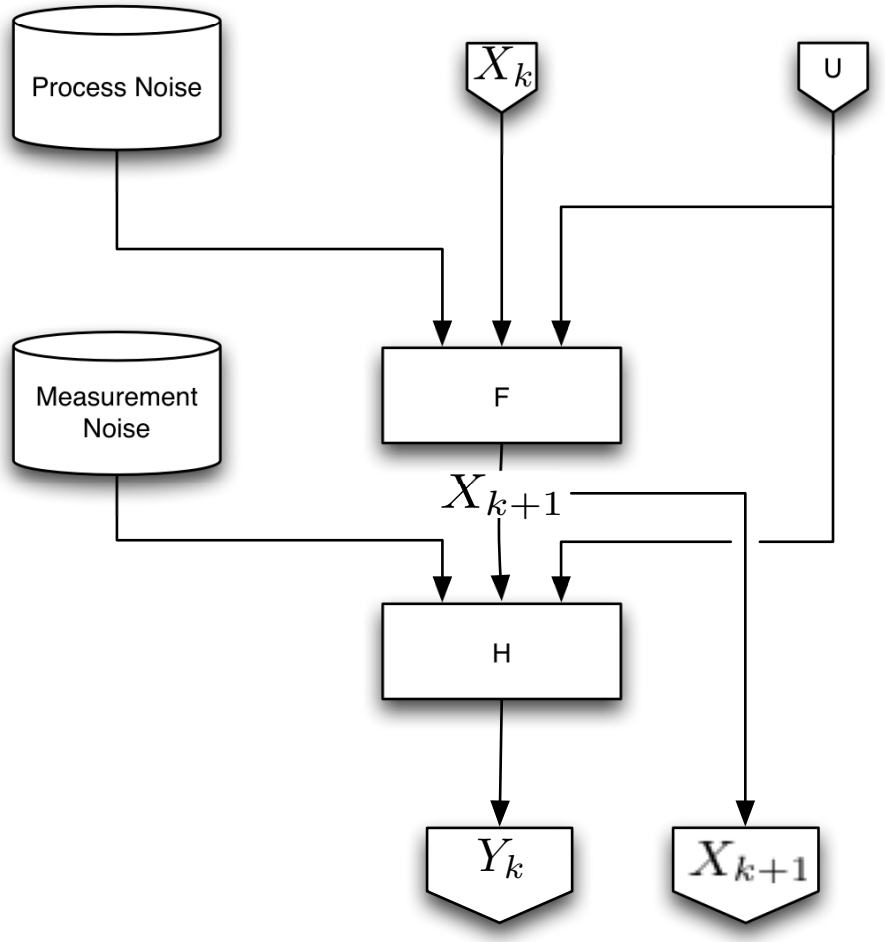


Figure 3.5: Whole System Flow Chart

3.2.5 Customized Operators

As discussed in the previous section, division and exponentiation are not natively supported by the Virtex 5 FPGA. Therefore, they have to be

designed. This is a point on which performance may be gained (or lost if poorly executed).

3.2.5.1 Division

According to [9] they are two categories of algorithms for division : recurrence and convergence. Recurrence approaches are slow as they produce one bit of the quotient per iteration, whereas convergence divisions produces several bits of the final quotient at each iteration through approximation, but require fast multipliers.

In this study, the recurrence approach has been chosen as a first approach, especially since only 1 division is present in the algorithm after factorizing. Were this approach too slow for other models, the convergence approach may be further studied and implemented.

The base version of the algorithm is the paper and pencil method that is common to most people. We are looking for $Q = \frac{N}{D}$ through the use of the recurrence equation [9] :

$$P_{k+1} = rP_k - q_{n-k-1}D \text{ for } k = 1, 2, \ldots, -1 \tag{3.1}$$

where:

P_k is the partial remainder after the selection of the kth quotient digit

$P_0 = N$ (subject to the constraint $|P_0| < |D|$)

r is the radix

q_{n-k-1} is the kth quotient to the right of the binary point

D is the divisor.

During this research it was observed that a mix between the restoring and non restoring method could be implemented on an FPGA by making use of parallelization. At each step, the partial remainder is shifted to the left and the n-kth bit of the Numerator is concatenated to it. If the partial remainder is positive, then the Numerator is effectively removed from the partial remainder and the $(n-k)^{th}$ bit of Q is set to one. If not, the partial remainder stays unchanged and the $(n-k)^{th}$ bit of Q is set to 0.

```
package DivFix;

import FixedPoint :: * ;

interface Div_IFC#(numeric type ai, numeric type af,
   numeric type bi, numeric type bf,
        numeric type ci, numeric type cf);
        method Action                   start (
           FixedPoint#(ai, af) a, FixedPoint#(bi, bf) b)
           ;
        method FixedPoint#(ci, cf)      result();
        method Action                   acknowledge();
endinterface

module mkDivFix(Div_IFC#(ai,af,bi,bf,ci,cf))
provisos(
        Arith#(FixedPoint::FixedPoint#(ci, cf)),
        Bitwise#(FixedPoint::FixedPoint#(ci, cf)),
        Add#(TAdd#(bi, bf), 1, TAdd#(TAdd#(bi, bf), 1)),
        Add#(1, a__, TAdd#(ai, af)),
```

```
Add#(1, b__, TAdd#(bi, bf)),
Bits#(FixedPoint::FixedPoint#(ai, af), TAdd#(ai,
    af))
);

Integer maxnum=valueOf(ai)+valueOf(af)+valueOf(
   cf);
Integer maxa=valueOf(ai)+valueOf(af);
Integer maxb=valueOf(bi)+valueOf(bf);
Integer maxc=valueOf(ci)+valueOf(cf);
Integer shift=valueOf(af)-valueOf(bf);


Reg#(Bool)
   available        <- mkReg(True);
Reg#(Bit#(TAdd#(ai,af)))
   n        <- mkReg(0);      //Numerator
Reg#(Bit#(TAdd#(TAdd#(bi,bf),1)))
   d        <- mkReg(1);      //Divisor
Reg#(Bit#(TAdd#(TAdd#(bi,bf),1)))
   r        <- mkReg(0);      //remainder
Reg#(Bit#(TAdd#(TAdd#(ai,bf),cf)))
   q        <- mkReg(0);
Reg#(UInt#(TLog#(TAdd#(TAdd#(ai,af),ci))))
   i        <- mkReg(0);      //cycle counter
Reg#(Bit#(1))
   signs    <- mkReg(?);

rule cycle (i<fromInteger(maxnum) && (available
   ==False));
        if(r>=d)
        action
                q[fromInteger(maxnum)-i]<=1;
                if(i<fromInteger(maxa))r<={(r-d)
                    [fromInteger(maxb)-2:0],n[
                    fromInteger(maxa)-1-i]};
                else r<=(r-d)<<1;
```

```
            endaction
            else
            action
                    q[fromInteger(maxnum)-i]<=0;
                    if(i<fromInteger(maxa))r<={r[
                        fromInteger(maxb)-2:0],n[
                        fromInteger(maxa)-1-i]};
                    else r<=r<<1;
            endaction
            i <= i+1;
    endrule: cycle

    method Action start(FixedPoint#(ai, af) a,
       FixedPoint#(bi, bf) b) if (available);
            n <= pack(abs(a));
            d <= signExtend(pack(abs(b)));
            r<=zeroExtend(n[fromInteger(maxa)-1]);
            i<=1;
            q<=0;
            signs<=(msb(pack(a)))^(msb(pack(b)));
            available <= False;
    endmethod

    method FixedPoint#(ci, cf) result() if (i >=
       fromInteger(maxnum) );
            FixedPoint#(ci, cf) res;

            if(shift<0)
                    res=(unpack(q[fromInteger(maxc)
                        -1:0])<<-shift);
            else
                    res=(unpack(q[fromInteger(maxc)
                        -1:0])>>shift);

            if(signs==1)
                    return -res;
            else
```

```
                        return res;
        endmethod

        method Action acknowledge() if ((i >=
           fromInteger(maxnum)) && !available);
                available <= True;
        endmethod
endmodule : mkDivFix

endpackage : DivFix
```

The division delay using this method is equal to the number of bits of the numerator plus the number of bits in the fractional part of the result.

In the final version of the FPGA implementation, the division module has actually not been used although originally motivated by requirements of the ESR's physical. Instead, the code was reused to develop and inversion function, which is similar to a division, only with a numerator always equal to one. In the end, this division module is now available for applications that may require it.

3.2.5.2 Inversion

The inversion module was created based on the division module. The motivation behind the creation and the use of this module is the fact that the inversion is a division that has a numerator equal to one. As a consequence, the number of steps required to complete an inversion is less than that of a division given the same data types. Because the fixed part of the numerator is just equal to one, there is no need to cycle over all the bit of the integer

part, hence the sparing of a number of steps equal to the number of bits in the integer part of the fixed point data type. Therefore there are $2f$ steps instead of $2f + i$, since only the fractional part has to be covered.

Naturally, divisions can be decomposed into an inversion and a multiplication:

$$\frac{N}{D} = N \times \frac{1}{D} \tag{3.2}$$

where N is the numerator and D is the denominator.

3.2.5.3 Inversion Farm

When an inversion operation is requested, an inversion module takes $2f$ cycles to complete the inversion. If only one inversion module is present in the FPGA, a new inversion can only start once the previous inversion has been completed, that is after $2f$ cycles. This would create a serious bottleneck in the flow. Multiple instances of these modules can be created and grouped into a farm. This allows for handling several inversions (or other operations) at the same time. If there are as many modules in the farm as it takes cycles to complete one operation, the farm can handle a new request at each cycle. This technique was used in the FPGA implementation to achieve full pipelining. The inversion farm is illustrated in figure 3.6.

The following is a description of the farm and its behavior:

- The farm contains 2f inverters.
- Initially, all the inverters are available.

- When an inversion is requested, the inversion is assigned to the next available inverter.
- A flag associated with the inverter is set as busy.
- At the end of the inversion, the inversion result is queued and data in the inverter is released and the inverters flag is set to available.

Since the inversion algorithm used has a constant number of steps, the order of the results in the queue will naturally be in the same order as the request order. There is no need for order management. If fast division was used, order management will be required, since the number of cycles required to complete the division may not be constant.

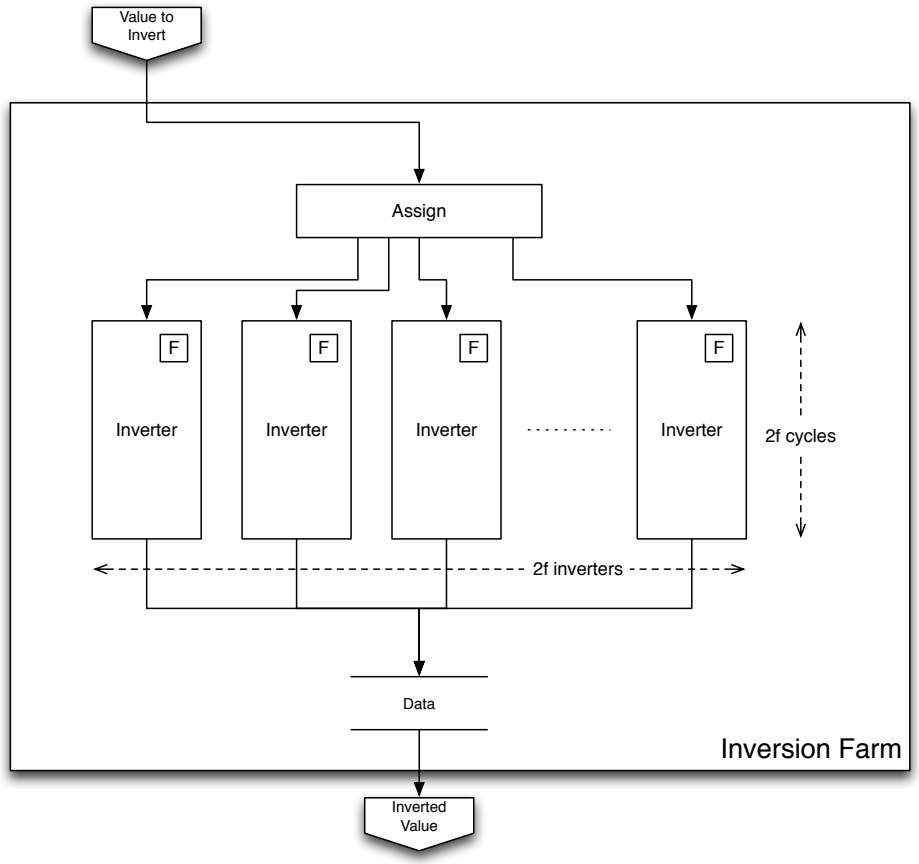


Figure 3.6: Inversion Farm Module

3.2.5.4 Exponential

A detailed architecture with performance results can be found in [4].

Two options have been identified concerning the exponential function. The first one is to store values that the exponential function would return in memory. The other option is to use the Taylor series expansion, which was

selected.

$$e^x = \sum_{n=o}^{\inf} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots \tag{3.3}$$

To compute the exponential with sufficient precision for the ESR estimator, the order should be 15 given the range of the input $[-2.6686; 2.1834]$. Whereas, the factorial numbers can be set as constants, the powers need to be computed. In order to do so in a pipelined fashion, the powers of x can be computed through successive multiplications. The computation of all the powers of x to the 16^{th} power can be done in 4 stages as in figure (3.7).

To complete the exponential computation, the powers of x are multiplied by the inverse of the factorials at each stage. These inverse factorials are pre-computed during compile-time and stored as constants. Multiplying by the stored inverses avoids long divisions, and can be done in one cycle.

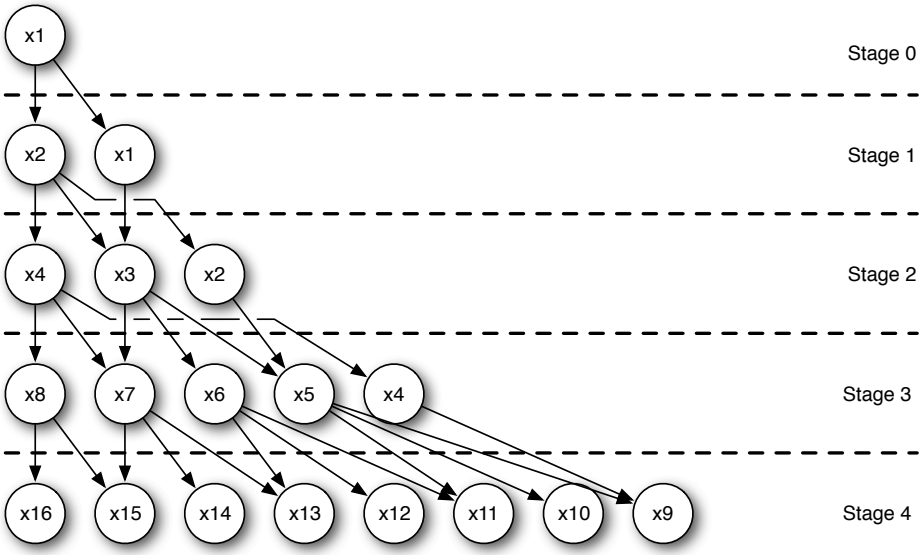


Figure 3.7: Computing the powers of x for the Taylor series expansion of the exponential function

3.2.6 FPGA Implementation Toolchain

In the course of designing and implementing the FPGA based design, only two software suites are required. The first suite is Bluespec System Verilog which includes the tools for defining and testing the functional architecture of the design and the tools to generate hardware description language files, namely Verilog files. The second is Xilinx ISE which can synthesis and place and route the design for a specific FPGA target.

Bluespec include a compiler, bsc. BSC can generate simulations files that are executable within Linux and can display the value of registers or

variables during runtime of the simulations, which behave in the same fashion as the printf() function is C. The same compiler can be used to generate Verilog files. The compiler provides the option of generating Verilog 95 compliant code, which excludes the display functions used for the simulation.

Xilinx ISE provides the rest of the tool chain to produce netlists and FPGA programation files. The first step consists in creating a project. The project has to be configured with the FPGA device number and the desired/required speed grade as well as design goal. The Verilog file that contains the top module is then imported into the project as well as optional additional Verilog files containing sub modules and the Bluespec provided libraries used in the design. From there, the design needs to be synthesized, placed and routed with optional constraint files.

3.2.6.1 Software Versions

- Bluespec System Verilog v.2008.11.C
- Xilinx ISE v.10.1

Chapter 4

Implementation Results and Analysis

4.1 Implementation Results

In this section the speeds, in terms of particles per second, of the CPU and the FPGA implementation are compared. For the CPU implementation, the results are obtained by timing the computation of a given number of particles. For the FPGA implementation, the speed results are obtained by dividing the maximum FPGA frequency given by the design tools after place/route by the number of cycles that the computation of a given number of particles computation would require, obtained through functional simulation.

4.1.1 CPU Results

For the given models, the approximate average speed of computation on the given CPU-based configuration was 2.67M particles per second. Since the CPU frequency is 2.6GHz, it can be deduced that each particle computation requires around 1000 cycles.

4.1.2 FPGA Results

The FPGA that was used to evaluate the performance of the implementation is XC5VSX240T Virtex 5 from Xilinx. The XC5VSX240T has the

following specifications found in table 4.1:

Table 4.1: FPGA Specifications

XC5VTXT240T	
Resource	Amount
Slices	37,440
Logic Cells	239,616
CLB Flip-Flops	149,760
Maximum Distributed RAM (Kbits)	4,200
Block RAM/FIFO w/ECC (36Kbits each)	516
Total Block RAM (Kbits)	18,576
DSP48E Slices	1,056

Under the FPGA design tool, Xilinx ISE 10.1, the FPGA has been configured with the settings found in table 4.2.

Table 4.2: Xilinx ISE Project Configuration

Project Configuration	
Parameter	Value
Family	Virtex 5
Device	XC5VTXT240T
Package	FF1738
Speed	-2
Design Goal	Balanced

The utilization results after synthesis and place and route are found in table 4.3.

Table 4.3: FPGA Resource Utilization

FPGA Resource Utilization			
Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	3,144	149,760	2%
Number used as Flip Flops	3,144		
Number of Slice LUTs	8,223	149,760	5%
Number used as logic	7,508	149,760	5%
Number used as Memory	689	67,200	1%
Number used as Dual Port RAM	689		
Number used as exclusive route-thru	26		
Number of route-thrus	316	299,520	1%
Slice Logic Distribution	Used	Available	Utilization
Number of occupied Slices	2,626	37,440	7%
Number of LUT Flip Flop pairs used	8,591		
Number with an unused Flip Flop	5,447	8,591	63%
Number with an unused LUT	368	8,591	4%
Number of fully used LUT-FF pairs	2,776	8,591	32%
Number of unique control sets	176		
Number of slice register sites lost to control set restrictions	282	149,760	1%
Other	Used	Available	Utilization
Number of DSP48Es	473	1,056	44%

Minimum period achieved with speed grade -2: 24.529ns (Maximum Frequency: 40.767MHz.)

The maximum speed is achieved with a full pipeline, that is when a particle is produced at each clock cycle. Since the maximum frequency obtained under the given configuration is 40.767MHz, the achieved speed was $40.767 Mparticle.s^{-1}$.

The 44% utilization of the DSP48Es (which provide for the fixed point multiplications) is the limiting factor as is it the resource that is the most utilized. The FPGA still has the potential to hold a second copy of the modules, which would double the simulation speed while using 88% of the DSP48E resources. Unfortunately, ISE runs out of memory while trying to place and route such a design, probably due to the complexity of finding a solution with so few resources left on the FPGA.

4.2 Analysis

4.2.1 Speed and Number of Devices

Based on these results and the knowledge of the architectures it is possible to define models to figure out the speed of particle computation and the number of device required to achieved a certain performance level. This performance level is defined by a number of particles, P to compute within a certain deadline t_D.

4.2.1.1 CPU Speed

For the purpose of real-time computing, that is to guaranty that all the tasks of the system are computed before a certain deadline (which is important when controlling a physical process), it is best to assume that the CPU will compute the code in a worst case scenario. In that scenario, the code doesn't benefit from CPU optimizations such as branch prediction and is executed in a constant worst time that is proportional to the CPU's frequency. A CPU may contain a set of cores, which are capable of computing one operation at a time. One operation may take several clock cycles depending on the architecture. The speed of computation therefore depends on the number of CPUs, the number of cores per CPU, the number of cycle to compute one particle and the CPU frequency with the following relationship:

$$R_{CPU} = N_{CPU} \times N_{cores} \times \frac{1}{N_{cycles}} \times F_{CPU} \tag{4.1}$$

where:

- R_{CPU} is the particle computation rate on a CPU-based computation system;
- N_{CPU} is the number of CPUs in the computation system;
- N_{cores} is the number of cores per CPU;
- N_{cycles} is the number of cycles per particle computation; and

- F_{CPU} is the frequency of the CPUs (assuming that the CPUs each run at the same frequency).

4.2.1.2 Number of CPUs Required for a Given Performance Level

As mentioned in the introduction, there is a relationship between the number of particles, the required deadline to compute that number of particles and the minimum number of devices required to achieve this performance. In the case of CPUs this relationship is simple since the computing speed is independent from the number of particles. From the rate equation (4.1), the following device requirement equation is inferred:

$$N_{CPU} \geq \left\lceil \frac{P}{N_{cores} \times R_{CPU} \times t_d} \right\rceil \tag{4.2}$$

4.2.1.3 FPGA Speed

Since the FPGA implementation uses pipelining, the FPGA speed depends both on the number of particles per FPGA, which determines the pipeline's usage, and the FPGA frequency. The speed is the time it takes to compute a number of particles. The number of cycles it takes to compute P particles is $P + L$, where P is the number of particles and L is the latency, that is the number of cycles it takes for the first particle computation to be output by the system. The speed of computation for one FPGA-based simulator is therefore equal to:

$$R_{simulator} = \frac{P}{L + P} \times F_{FPGA} \tag{4.3}$$

where:

- $R_{simulator}$ is the particles computation rate for one FPGA-based simulator in $particles.s^{-1}$;
- P is the number of particles;
- L is the latency in $cycles$; and
- F_{FPGA} is the FPGA frequency in $cycles.s^{-1}$.

Further, an FPGA-based system can be comprised of multiple FPGA devices, each containing multiple simulators operating in parallel. This results in the following model:

$$R_{FPGA} = N_{FPGA} \times N_{simulators} \times \frac{P}{P+L} \times F_{FPGA} \tag{4.4}$$

where:

- R_{FPGA} is the particles computation rate for one FPGA-based simulator in$particles.s^{-1}$;
- N_{FPGA} is the number of FPGAs in the FPGA based system;
- $N_{simulators}$ is the number of simulators per FPGA;
- P is the number of particles;
- L is the latency in cycles; and
- F_{FPGA} is the FPGA frequency in $cycles.s^{-1}$.

4.2.1.4 Number of FPGAs Required for a Given Performance Level

As in the CPU case, a minimum number of FPGAs can be determined. Posing that for each FPGA device $\left\lceil \frac{P}{N_{FPGA}} \right\rceil$ is the number of particles per device, we have:

$$\frac{L + \left\lceil \frac{\frac{P}{N_{FPGA}}}{N_{simulators}} \right\rceil}{F_{FPGA}} \leq t_d \tag{4.5}$$

$$\implies L + \left\lceil \frac{\frac{P}{N_{FPGA}}}{N_{simulators}} \right\rceil \leq t_d F \tag{4.6}$$

$$\implies \left\lceil \frac{P}{N_{simulators} N_{FPGA}} \right\rceil \leq t_d F - L \tag{4.7}$$

$$\implies N_{FPGA} \geq \left\lceil \left\lceil \frac{P}{N_{simulators}} \right\rceil \frac{1}{t_d F - L} \right\rceil \tag{4.8}$$

4.2.2 FPGA v. CPU performance outlook

Using the models presented in the previous sections, the performance of CPUs and FPGAs can be forecasted and compared. Currently, we were able to have only one simulator per FPGA and we only used one CPU core on one CPU. In the near future, we can hope to increase the FPGA frequency and hold multiple simulators per FPGA. On the CPU side, the use of 4 cores is currently feasible. To get an idea of what lies ahead, we can plot the performance levels given by the models.

4.2.2.1 Speeds

The following plots show the speed performance for the CPU and FPGA implementations. The speed is measured in $particles.s^{-1}$. For CPUs, the

speed remains constant for a given number of CPUs. The reason for this is that the CPU implementation is serial and therefore its speed is independent from the number of particles. In contrast, the FPGA computation speed depends on the number of particles. The speed increases as there are more particles to fill the pipeline. The maximum theoretical speed for one FPGA configuration is equal to the FPGA's clock frequency since the pipelining has been done in such a way that one particle computation result is produced on each cycle after the first result is obtained. The maximum speed is achieved when the input to output latency (i.e. the number of clock cycles it takes for the result to be obtained once the input has been applied) is negligible compared to the number of particles. In this implementation, this appears to happen around 15000 particles. From the performance plot it can be seen that the FPGA based simulations are substantially faster on FPGAs if the number of particles is greater than 10 in all cases. Since many applications are susceptible to require many more particles, FPGAs appear to be the solution of choice for future applications in terms of speed.

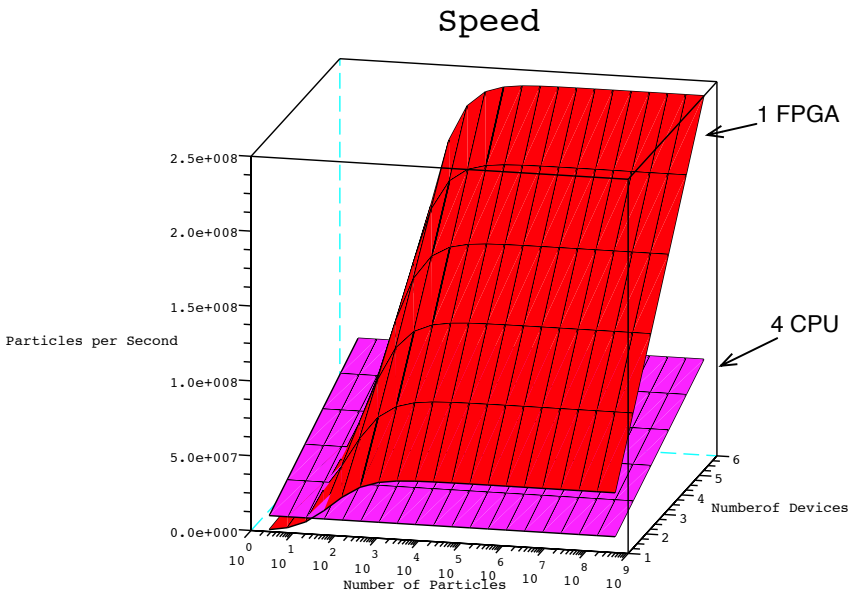


Figure 4.1: Particle Computation Rates for 4 CPU Cores and 1 FPGA based simulator

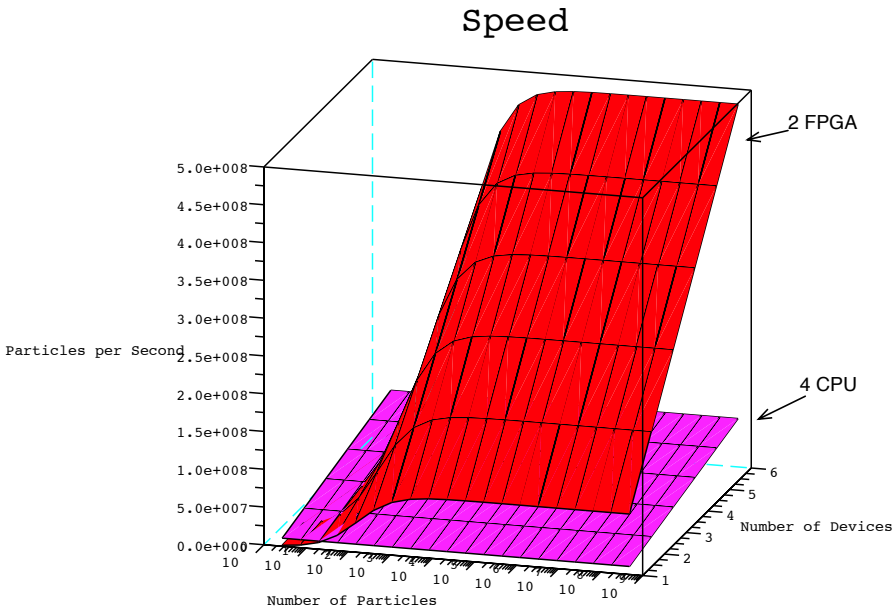


Figure 4.2: Particle Computation Rates for 4 CPU Cores and 2 FPGA based simulator

4.2.2.2 Number of Devices

The following plots show the minimum numbers of CPUs or FPGAs that are or would be necessary to compute a certain number of particles under a given deadline. To better view the influence of deadline and particle the required minimum number of devices has been plotted in 3D. In these 3D plots it can be seen that the area where two CPUs are required is far away from the area of operation of the ESR estimator. However, for other applications such as the Gas Metal Arc Welding process which is a much quicker process,

there may be a need for computing more particles under a shorter deadline. If this is the case, then it appears again that FPGAs will be the device of choice. The graphs show that many more CPUs will be needed as the number of particles increases and the deadline to compute this number of particles shortens compared to the number of FPGAs that will be necessary.

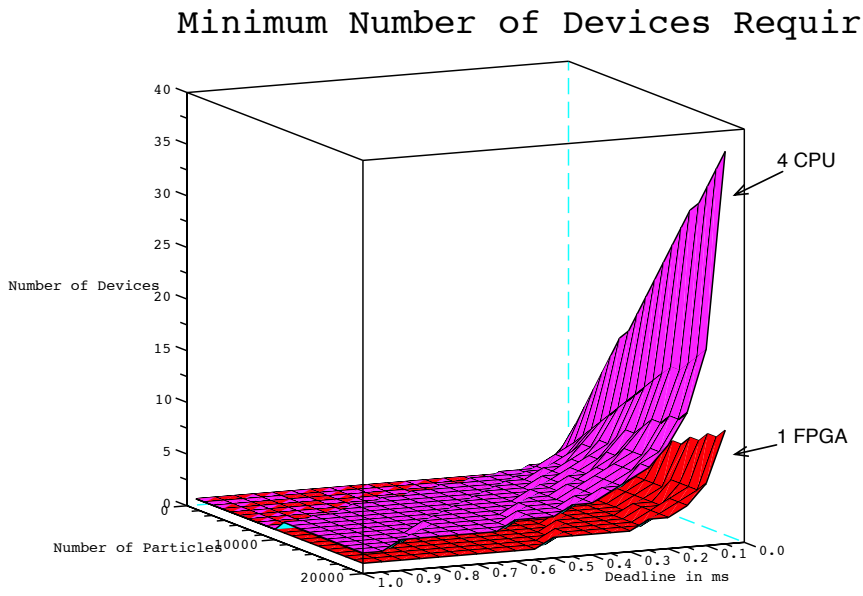


Figure 4.3: Number of Devices Required; 4 Cores per CPU v. 1 Simulator per FPGA

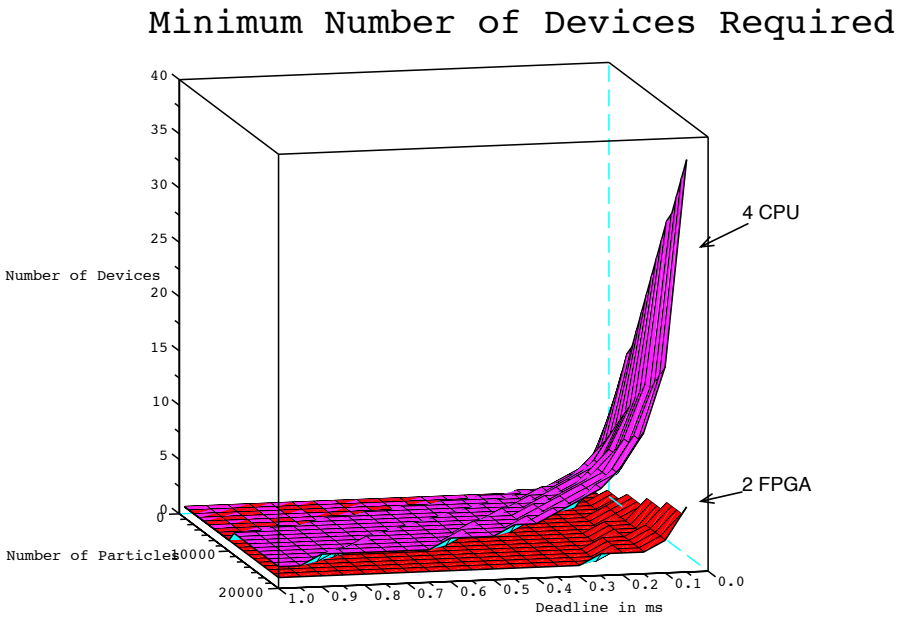


Figure 4.4: Number of Devices Required; 4 Cores per CPU v. 2 Simulator per FPGA

4.2.2.3 Performance Analysis

The major speed advantage that the FPGA implementation features is the use of pipelining. Pipelining the particle computation allows for intense parallelization, which leads to a substantial speed increase. In terms of speed and number of devices required, FPGAs provide the best performance at this point in time as well as in the years to come. Whereas the CPU implementation can only be improve through the use of multiple cores (the current working

implementation uses only one single core), the FPGA implementation can still receive more enhancements. In the current implementation, many additions and multiplications are grouped together. Since the FPGA still has available resources, these steps of the pipeline can be cut into smaller substeps in which only one operation is performed. This will lead to two improvements: a shorter maximum delay (faster clock) and more parallelization. Currently, the FPGA implementation is limited by the number of DSP48 blocks used, which is 44%. The design could be modified to use fewer multipliers by adding some steps for instance.

Chapter 5

Conclusion and Future Work

5.1 Analysis of Results

The project started with the objective to compute at least 100 particles within 133ms for the ESR process given the Ahn's models. As it turns out, both technologies are capable of achieving this performance level with a large margin. This opens the door to using particle filtering estimation online for the Electro-Slag Remelting process with even further precision than before.

During the course of this performance study, we were able to determine macro level performance models for CPUs and FPGAs that give an idea of the capabilities of these two devices. These macro level performance models are a real asset for the solution space exploration of the implementation problem. For the process at hand, these models show that FPGA devices are capable of achieving higher performance than CPUs. Furthermore, these models also show that fewer FPGA devices would be required to achieve the same performance level as CPU for this application. As the rate of required particles per second increases, the number of CPUs required increases significantly faster in contrast to the the number of FPGA to complete the same tasks. FPGAs provide further performance that scales with complexity. In terms of costs,

implementation complexity and predictability, FPGAs provide the best solution for systems with large state variables that require many particles to be computed under a short period of time.

5.2 Conclusion

The experimental results show that an FPGA is capable of out-performing a CPU in terms of speed for the specific application of estimating the state of a particular process using Particle Filtering. The FPGA implementation exploits the fact that the estimation technique uses processes that can be executed in parallel, in this case the process of drawing of particles. The technique further exploits the fact that the particles are run through physical models that are also parallel due to the fact that processes occur in parallel in the physical world.

On the down side, FPGAs are devices that are more complex to program and may fail to provide fast performance in cases where the computing requires further non trivial mathematical operations such as square roots or logarithms. These operations are further burdened by the requirements of data representation. For instance, if the application requires double precision floating point data representation, the number of simultaneous operations that can be done will be significantly lower than the number of operations that would been done with only fixed point data representation. This is due to the fact that floating point operations require more resources than their fixed point equivalents.

The FPGA is exploiting a more efficient structure and parallelism to get the same amount of work done than its CPU counterpart. Specifically the FPGA runs at a much lower frequency than the CPU for the same amount of work. From a broad perspective, lower frequency usually implies lower power consumption. Although the manufacturing process requires substantially more power than its controller (the controller's power consumption is not an issue for the application at hand), for other applications power may be an issue such as in mobile applications. One example is mobile robotics, where for instance particle filtering is used to determine a robot's position based on physical models and noisy observations. Other applications may include transportation systems, portable medical devices or other systems in which there is also a critical interaction with the physical world.

Lastly, these results show that more complex models can now be developed to even further reduce the root mean squared error of the estimate, potentially yielding even better control signals, improving the quality of the resulting material produced.

5.3 Future Work

The future of this work is to develop a tool chain to provide assistance in the choice of a target device. This tool chain would use implementation performance results and the sort of performance models that were put together in this research to evaluate the speed and precision offered by each potential target device.

One additional future task would consist in evaluating the other tools ESL that are commercially available such as: The Mathworks' Filter Design HDL Coder, AccelDSP from Xilinx, National Instrument's Labview FPGA, DK Design Suite from Agility DS, Cynthesizer from Forte.

Finally other computing devices such as GPUs and DSPs should be studied for suitability. The GPU suitability research is in the process of evaluation within the research group.

Appendix

Appendix 1

Bluespec Implementation Code

1.1 Description

This appendix is the complete set of code that runs the physical and measurement models. The code is decomposed in several Bluespec .bsv files that contain modules. The code is given in order of dependency, that is that later files require earlier files. The order is therefore: datatypes, parameters, inversion and exponentiation, F and H separetely, F and FH together, multiple F and H in parallel (the latter is provided eventhough the FPGA wasn't able to support more than one piped simulator) and finally some test benches.

1.2 Source Code

1.2.1 Types.bsv

```
/**
* File : Types.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
        This file contains the datatypes that are used
           for the simulations.
        All datatypes are of the fixedpoint type from
           the Bluespec Library.
```

```
        Finally, it was decided that all the data will
           be of type
        FixedPoint#(20,28). This is because of the range
            of some data and the
        size of the DSP48 multipliers (48bits)

        If required, different size could be defined for
            each variables. A lot
        of the simulation models have been designed to
           accept different sizes.

        Due to the possibility of different data types
           for the variables within
        a same category (state, command, observation,
           noise...) the data is
        grouped in structures.
*/


package Types;

        import FixedPoint :: * ;

        typedef 20 Isize;
        typedef 28 Fsize;
        typedef 48 DTsize;

        typedef FixedPoint#(Isize,Fsize) Datatype; //
           standard datatype for this project

        //State Vector data types
        typedef Datatype X1type;
        typedef Datatype X2type;
        typedef Datatype X3type;
        typedef Datatype X4type;
        typedef Datatype X5type;
```

```
        //U data types
        typedef Datatype U1type;
        typedef Datatype U2type;

        //Measurment noise data types
        typedef Datatype MN1type;
        typedef Datatype MN2type;
        typedef Datatype MN3type;
        typedef Datatype MN4type;
        typedef Datatype MN5type;

        //Process noise data types
        typedef Datatype PN1type;
        typedef Datatype PN2type;

        //Observation vector datatypes
        typedef Datatype Y1type;
        typedef Datatype Y2type;
        typedef Datatype Y3type;
        typedef Datatype Y4type;
        typedef Datatype Y5type;



typedef struct {
        X1type delta;
        X2type ts;
        X3type d;
        X4type xram;
        X5type me;
} Xtype deriving (Bits);

typedef struct {
        Y1type d;
        Y2type xram;
        Y3type ir;
```

```
        Y4type lc;
        Y5type volt;
} Ytype deriving (Bits);

typedef struct {
        U1type ic;
        U2type vramc;
} Utype deriving (Bits);

typedef struct {
        MN1type d;
        MN2type xram;
        MN3type ir;
        MN4type lc;
        MN5type volt;
} MeasNoisetype deriving (Bits);

typedef struct {
        PN1type ic;
        PN2type vramc;
} ProcessNoisetype deriving (Bits);


endpackage: Types
```

1.2.2 Params.bsv

```
/**
* File : Params.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
        Contains the precomputed parameters for the
           physical and measurment
        models.
```

```
        For the details about the values of these
           parameters, consult Ahn's
        dissertation.
*/

package Params;
import FixedPoint :: * ;
import Types :: * ;
typedef Datatype Param;
//Param p_R1 = 6.0000000000e-03;
//Param p_R1e = 6.5000000000e-03;
//Param p_k1 = 1.3000000000e+03;
Param p_De = 2.0320000000e+01;
Param p_re = 1.0160000000e+01;
Param p_Di = 2.5400000000e+01;
Param p_ri = 1.2700000000e+01;
Param p_Aelect = 1.2130000000e-03;
Param p_m0 = 1.4285714286e-02;
Param p_m1 = 7.6923076923e-04;
Param p_Tsstar = 2.2000000000e+03;
Param p_Tr = 3.0000000000e+02;
Param p_rhor = 7.8300000000e+00;
Param p_Cr = 4.3400000000e-01;
Param p_Kroom = 6.3900000000e-01;
Param p_alphar = 1.8803962074e-01;
Param p_Tm = 1.7830000000e+03;
Param p_rhom = 7.4000000000e+00;
Param p_Cm = 1.1680000000e+00;
Param p_Km = 3.1300000000e-01;
Param p_alpham = 3.6213439467e-02;
Param p_hm = 8.9287129300e+03;
Param p_Tsup = 1.8830000000e+03;
Param p_L = 2.7196000000e+02;
Param p_hsup = 1.1543287930e+04;
Param p_Cs0 = 1.4700000000e+00;
Param p_rhos = 2.5500000000e+00;
Param p_Ks = 4.1800000000e-02;
```

```
Param p_Tss = 1.7730000000e+03;
Param p_Ms = 5.6750000000e+04;
Param p_Vs = 2.2254901961e+04;
Param p_Ae = 3.2429278662e+02;
Param p_Ai = 5.0670747910e+02;
Param p_betam = -8.0741590882e-01;
Param p_lambda = 3.4149767859e+00;
Param p_a0 = 3.6000000000e-01;
Param p_den = 2.1244930358e+01;
Param p_Cdd = 1.0746632253e+01;
Param p_Cdp = 5.1437804365e+00;
Param p_Csd0 = 2.0781252910e+00;
Param p_Csp = 1.7681745250e+00;
Param p_mu = 5.5000000000e-01;
Param p_mdot0 = 5.0000000000e+01;
Param p_Pm0 = 7.7995188716e+04;
Param p_P0 = 1.4180943403e+05;
Param p_pm0 = 2.4050855256e+02;
Param p_Ts0 = 2.2000000000e+03;
Param p_d0 = 0.0000000000e+00;
Param p_hs0 = 4.3920610764e+01;
Param p_He = 5.7675911885e-01;
Param p_Hs = 4.2642023223e-02;
Param p_R0 = 5.0000000000e-03;
Param p_I0 = 5.3255879305e+03;
Param p_V0 = 2.6627939653e+01;
Param p_delta0 = 1.4584705609e+01;
Param p_Sdot0 = 2.0835359390e-02;
Param p_Vram0 = 7.5007293803e-03;
Param p_mur0 = 0.0000000000e+00;
Param p_sigmadV = 5.0000000000e-02;
Param p_sigmaPos = 3.0000000000e-01;
Param p_sigmaImeas = 2.0000000000e+02;
Param p_sigmaLC = 5.0000000000e+02;
Param p_sigmaV = 1.0000000000e-01;
Param p_sigmaI = 1.8420000000e+02;
Param p_sigmaVram = 1.0000000000e-02;
```

```
Param p_sigmamur = 5.0000000000e-04;
Param p_sigmaa = 3.6000000000e-03;
Param p_sigmaIb = 2.6627939653e+00;
Param p_sigmaVramb = 7.5007293803e-04;
Param p_sigmaVoltb = 1.3313969826e-02;
Param p_k0 = 7.0000000000e+01;
Param p_k1 = 1.3000000000e+03;
Param p_R1 = 6.0000000000e-03;
Param p_R1e = 6.5000000000e-03;
Param p_dInflection = 0.0000000000e+00;
Param p_SlagTemperatureTimeConstant = 1.0000000000e+01;
Param p_KTs = 1.0000000000e-01;
Param p_DepthControlTimeConstant = 1.0000000000e+00;
Param p_Kd = 1.0000000000e+00;

Param p_Pi = 3.14159265358979323846;
endpackage
```

1.2.3 inverse.bsv

```
/**
* File : inverse.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
*       Computes the inverse value of a fixed point
   value in a pipelined fashion.
*
*       The algorithm used is the slow division.
*       The number of steps it takes is 2*fsize, where
   fsize is the size
*       of the fractional part.
*
*       Note: Since this algorithm was adapted from the
   division algorithm
```

```
*       most of the variable names refer to names that
   would be used for divisions
*/

package inverse;

import FixedPoint :: * ;

/**
abstract interface Div_IFC
        parameters:
                i: number of bits for the interger part
                f: number of bits for the fractional
                   part
        start: feed value to be inverted into the
           pipeline
        result: returns the inverted value
        acknowledge: removes the value from the pipeline
           .
*/
interface Div_IFC#(numeric type i,numeric type f);
        method Action            start (FixedPoint#(i,f)
           den);
        method Maybe#(FixedPoint#(i,f)) result();
        method Action            acknowledge();
endinterface

module mkInverse(Div_IFC#(i,f))
provisos(
                Bits#(FixedPoint::FixedPoint#(i, f),
                   TAdd#(i, f)),
                Add#(TAdd#(i, f), f, TAdd#(TAdd#(i, f),
                   f)),
                Add#(1, a__, TAdd#(i, f))
        );

        Integer isize=valueOf(i);
```

```
Integer fsize=valueOf(f);
Integer fpsize=isize+fsize;

Reg#(Bit#(TAdd#(i,f)))                  d
   <- mkReg(0);
Reg#(Bit#(TAdd#(i,f)))                  r
   <- mkReg(0);
Reg#(Bit#(TAdd#(i,f)))                  q
   <- mkReg(0);

Reg#(Bit#(TAdd#(TLog#(TAdd#(f,f)),1)))  count
          <- mkReg(0);
Reg#(Bool)                              available
   <- mkReg(True);
Reg#(Bit#(1))                           sign
   <- mkReg(?);
Reg#(Bool)                              outOfRange
   <- mkReg(False);

rule cycle((!available) && (count<=fromInteger
   (2*fsize)));

        if(r<d)
                action
                        //$display("r<d\n");
                        r<=r<<1;
                        q<=q<<1;
                endaction
        else
                action
                        //$display("r>=d\n");
                        r<={(r-d)<<1};
                        q<=(q<<1)+1;
                endaction
        if(q[fpsize-2]==1)outOfRange<=True; //-2
            because msb is 1 after shift need to
            add: && count<fromInteger(2*fsize)
```

```
            count<=count+1;
            /*
            $display("cycle==========\n");
            $display("cycle: d = %b :",d);
            $display("cycle: r = %b :",r);
            $display("cycle: q = %b :",q);
            */

    endrule

    method Action start (FixedPoint#(i,f) den) if
        (available);
            FixedPoint#(i,f) one=1;
            d <= {pack(abs(den)),'0};
            r <=1;
            q <=0;
            count <= 0;
            available <=False;
            sign<=msb(pack(den));
            outOfRange <= False;
            /*
             $display("count = %d \n",count);
             $display("isize = %d \n",isize);
             $display("fsize = %d \n",fsize);
             $display("fpsize = %d \n",fpsize);
             */
              $display("INV START = %d \n",count);
    endmethod

    method Maybe#(FixedPoint#(i,f)) result() if (
       count>fromInteger(2*fsize) && !available);
            if(outOfRange==False)
                    if(sign==1)
                            return Valid(-unpack(q))
                                ;
                    else
```

```
                                    return  Valid(unpack(q))
                                       ;
                    else
                                return Invalid;
            endmethod

            method Action   acknowledge() if (count>=
               fromInteger(2*fsize) && !available);
                    available <=True;
            endmethod

endmodule : mkInverse

endpackage : inverse
```

1.2.4 invFarm.bsv

```
/**
* File : invFarm.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
*       The inverse farm is a set of inverters that can
   accomodate a new
        inversion request a each cycle, while continuing
            the inversions that
        are already under way.
        To do this, the number of inverters in the set
           must be equal to the
        number of cycles it takes to compute an
           inversion (2*f).
        At each new request, an inversion is assigned to
            an available inverter
        and this inverter is flagged as busy until the
           result has been removed
```

```
        from it.

        Since each inversion requires the same number of
            cycles, the results
        are provided in the same order as the requests.
*/

package invFarm;

import FixedPoint :: *;
import inverse :: *;
import FIFO    :: *;


/**
interface InvFarm_IFC

Parameters:
        i: number of bits of the integer part
        f: number of bits of the fractional part
        nbinv: number of inverters (suggested 2f)

Methods:
        submit: submit a value to be inverted
        result: return the next available result in the
           other the it was requested
        acknowledge: remove the current result from the
           inverter and sets the
                next inverter from which the next result
                    should be read from.
*/
interface InvFarm_IFC#(numeric type i,numeric type f,
   numeric type nbinv);
        method Action                    submit (
           FixedPoint#(i,f) inval);
        method Maybe#(FixedPoint#(i,f)) result();
        method Action                    acknowledge();
```

```
endinterface

module mkInvFarm(InvFarm_IFC#(i,f,nbinv))
provisos(
        Bits#(FixedPoint::FixedPoint#(i, f), TAdd#(i, f)
           ),
        Add#(TAdd#(i, f), f, TAdd#(TAdd#(i, f), f)),
        Add#(1, a__, TAdd#(i, f))
);
        Integer isize=valueOf(i);
        Integer fsize=valueOf(f);
        Integer fpsize=isize+fsize;
        Integer n=valueOf(nbinv);

        //Reg#(Bool)     busy[n];

        Reg#(UInt#(TAdd#(TLog#(nbinv),1)))
           nextInverter    <-mkReg(0); //Index of the
           next available inverter
        Reg#(UInt#(TAdd#(TLog#(nbinv),1)))
           nextResult      <-mkReg(0); //Index of the
           next result

        FIFO#(FixedPoint#(i,f)) in <- mkLFIFO;
        FIFO#(Maybe#(FixedPoint#(i,f))) out <- mkLFIFO;

        //create n inverters
        Div_IFC#(i,f) inverter[n];
        for(Integer i=0;i<n;i=i+1)
                inverter[i] <- mkInverse;

        for(Integer i=0;i<n;i=i+1)
        begin
                Reg#(Bool) busy_i<- mkReg(False);
                rule start_i( (!busy_i) && (nextInverter
                   ==fromInteger(i)) ); //start
                    inversion on the next available
```

```
                    inverter.
                        inverter[i].start(in.first());
                        in.deq();
                        busy_i<=True;
                        if(i>=fromInteger(n-1))
                                nextInverter <= 0;
                        else
                                nextInverter<=
                                    fromInteger(i)+1;
                        //$display("START %d",i);
                endrule

                rule end_i(busy_i && (nextResult==
                    fromInteger(i)) ); //Once the
                    inverter for which a result is
                    expected is done, get the result and
                    free the inverter.
                        out.enq(inverter[i].result());
                        inverter[i].acknowledge;
                        //$display("END %d",i);
                        busy_i<=False;
                        if(i>=fromInteger(n-1))
                                nextResult <= 0;
                        else
                                nextResult<=fromInteger(
                                    i)+1;
                endrule
        end

/*
        rule display_values;
                //$display("nextInverter= %d",
                    nextInverter);
                //$display("nextResult= %d",nextResult);
                //$display("busy[nextInverter]= %d",busy
                    [nextInverter]);
        endrule
```

```
*/
        method Action                           submit (
           FixedPoint#(i,f) inval);
                //$display("SUBMIT");
                in.enq(inval);
        endmethod

        method Maybe#(FixedPoint#(i,f)) result();
                return out.first();
        endmethod

        method Action                           acknowledge();
                out.deq();
        endmethod


endmodule

endpackage : invFarm
```

1.2.5 ExponFix.bsv

```
/**
* File : ExponFix.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
*       Computes the exponential for the ESR by using a
   Taylor
*       series expanssion up to the 15th order.
*       The implementation is a 4 stage pipeline.
*       The interface is abstract. Since the module is
   intended
*       for fixed point values, the integer and
   fractional sizes
```

```
*       must be provided.
*
*       Stage0: compute x^2,
*       Stage1: compute x^3 and x^4
*       Stage2: compute x^5, x^6, x^7 and x^8
*       Stage3: compute x^9, x^10, x^11, x^12, x^13, x
   ^14, x^15, x^16
*/

package ExponFix;

import FIFO::*;
import FixedPoint :: * ;

function Integer factorial (Integer n) = (n<=1 ? 1 : n *
    factorial(n-1));

/**
Interface Exp_IFC
Description:
        values:
                ai:     size of the integer part
                af:     size of the factional part
                a:      data to be exponentiated
        methods:
                feed:           feeds value into the
                   pipeline
                fetch:          returns exponential (
                   will not be called if
                                there is no result in
                                   the pipeline)
                removeresult:   removes the data in the
                   last stage

*/
interface Exp_IFC#(numeric type ai,numeric type af);
        method Action   feed(FixedPoint#(ai,af) a);
```

```
        method FixedPoint#(ai,af)       fetch();
        method Action    removeresult();
endinterface

module mkExponFix(Exp_IFC#(ai,af))
provisos(
        //Bitwise#(FixedPoint::FixedPoint#(ai, af)),
        RealLiteral#(FixedPoint::FixedPoint#(ai, af)),
        Arith#(FixedPoint::FixedPoint#(ai, af)),
        Add#(ai, af, TAdd#(ai, af)),
        Add#(1, a__, ai)
        );
        FIFO#(FixedPoint#(ai,af)) a1    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a1p   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a2    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a2p   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a3    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a3p   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a4    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a5    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a6    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a7    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a8    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a9    <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a10   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a11   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a12   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a13   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a14   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a15   <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) a16   <- mkLFIFO();

        FIFO#(FixedPoint#(ai,af)) expval0  <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) expval1  <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) expval2  <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) expval3  <- mkLFIFO();
        FIFO#(FixedPoint#(ai,af)) expval4  <- mkLFIFO();
```

```
rule stage0;
        a2.enq(a1.first()*a1.first());
        a1p.enq(a1.first());
        expval1.enq(expval0.first()+a1.first());
        a1.deq();
        expval0.deq();
        $write( "stage 0 expval0 is " ) ;
           fxptWrite( 10, expval0.first() );
endrule

rule stage1;
        a2p.enq(a2.first());
        a3.enq(a2.first()*a1p.first());
        a4.enq(a2.first()*a2.first());
        expval2.enq(expval1.first()+a2.first()*
           fromRational(1,factorial(2)));
        a1p.deq();
        a2.deq();
        expval1.deq();
        $write( "stage 1 expval1 is " ) ;
           fxptWrite( 10, expval1.first() );
endrule

rule stage2;
        a3p.enq(a3.first());
        a5.enq(a2p.first()*a3.first());
        a6.enq(a3.first()*a3.first());
        a7.enq(a3.first()*a4.first());
        a8.enq(a4.first()*a4.first());
        expval3.enq(expval2.first()+a3.first()*
           fromRational(1,factorial(3))+a4.first
           ()*fromRational(1,factorial(4)));
        a2p.deq();
        a3.deq();
        a4.deq();
        expval2.deq();
```

```
        $write( "stage 2 expval0 is " ) ;
           fxptWrite( 10, expval2.first() );
endrule

rule stage3;
        a9.enq(a6.first()*a3p.first());
        a10.enq(a5.first()*a5.first());
        a11.enq(a5.first()*a6.first());
        a12.enq(a6.first()*a6.first());
        a13.enq(a6.first()*a7.first());
        a14.enq(a7.first()*a7.first());
        a15.enq(a8.first()*a7.first());
        a16.enq(a8.first()*a8.first());
        expval4.enq(expval3.first()+a5.first()*
           fromRational(1,factorial(5))+a6.first
           ()*fromRational(1,factorial(6))+a7.
           first()*fromRational(1,factorial(7))
           + a8.first()*fromRational(1,factorial
           (8)));
        a3p.deq();
        a5.deq();
        a6.deq();
        a7.deq();
        a8.deq();
        expval3.deq();
        $write( "stage 3 expval3 is " ) ;
           fxptWrite( 10, expval3.first() );
endrule

method Action feed(FixedPoint#(ai,af) a);
        a1.enq(a);
        expval0.enq(1);
endmethod

method FixedPoint#(ai,af) fetch();
        return expval4.first()+a9.first()*
           fromRational(1,factorial(9))+a10.
```

```
                    first()*fromRational(1,factorial(10))
                    +a11.first()*fromRational(1,factorial
                    (11))+a12.first()*fromRational(1,
                    factorial(12))+a13.first()*
                    fromRational(1,factorial(13))+a14.
                    first()*fromRational(1,factorial(14))
                    +a15.first()*fromRational(1,factorial
                    (15))+a16.first()*fromRational(1,
                    factorial(16));
        endmethod

        method Action   removeresult();
                a9.deq();
                a10.deq();
                a11.deq();
                a12.deq();
                a13.deq();
                a14.deq();
                a15.deq();
                a16.deq();
                expval4.deq();
        endmethod

endmodule : mkExponFix

endpackage : ExponFix
```

1.2.6 HBRAM.bsv

```
/**
* File : HBRAM.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
```

```
*       Computes the ESR's measurment model in a
   pipelined fashion
*
*/

package HBRAM;
import Types::*;
import FixedPoint :: * ;
import ExponFix::*;
import FIFO::*;
import BRAMFIFO :: * ;
import Params::*;
//initParameters();


/**
interface H_IFC
Description:
        interface for the measurement module

        measure: feed the state, the command vector and
           noise into the pipeline
        fetchMeasurement: return the value of the
           simulated measurement
        removeMeasurement: remove the measurement value
           from the pipeline
*/
interface H_IFC;
        method Action   measure(Xtype x ,Utype u ,
           MeasNoisetype n);
        method Ytype    fetchMeasurement();
        method Action   removeMeasurement();
endinterface


module mkH(H_IFC);
```

```
//NOTE: The minimum size for the these first
   FIFOS has not been determined
//They were set to 20 for now.
FIFO#(Xtype) x          <- mkSizedBRAMFIFO(20);
FIFO#(Utype) u          <- mkSizedBRAMFIFO(20);
FIFO#(MeasNoisetype) n  <- mkSizedBRAMFIFO(20);

FIFO#(Ytype) y          <- mkSizedBRAMFIFO(20);


FIFO#(Datatype) r       <- mkSizedBRAMFIFO(20);
FIFO#(Datatype) ubvolt  <- mkSizedBRAMFIFO(20);
   //unbiased voltage in stage 7
FIFO#(Datatype) volt    <- mkSizedBRAMFIFO(20);


//stage1
FIFO#(Datatype) d4noise          <-
   mkSizedBRAMFIFO(11);
FIFO#(Datatype) xram4noise              <-
   mkSizedBRAMFIFO(11);
FIFO#(Datatype) dStage3                 <-
   mkSizedBRAMFIFO(3);
FIFO#(Datatype) meStage2         <- mkLFIFO();
FIFO#(Datatype) meStage3                <-
   mkSizedBRAMFIFO(3);
FIFO#(Datatype) ir               <-
   mkSizedBRAMFIFO(9);
FIFO#(Datatype) ir4noise                <-
   mkSizedBRAMFIFO(11);
FIFO#(Datatype) m0Xd             <- mkLFIFO();
FIFO#(Datatype) m1Xd                    <-
   mkLFIFO();
FIFO#(Datatype) aeXrhosXd        <- mkLFIFO();
FIFO#(Datatype) tsMtsstar               <-
   mkLFIFO();
```

```
//stage2
FIFO#(Datatype) negDepthR       <-mkLFIFO();
FIFO#(Datatype) posDepthR       <-mkLFIFO();
FIFO#(Datatype) exponent        <-mkLFIFO();
FIFO#(Datatype) posDepthMass    <-mkLFIFO();

//stage3
FIFO#(Datatype) rd      <-mkSizedBRAMFIFO(6);
FIFO#(Datatype) lc              <-
   mkSizedBRAMFIFO(9);

Exp_IFC#(Isize,Fsize) exponentiator <-
   mkExponFix();

Datatype ib=0;
Datatype voltb=0;

rule stage1;
        //Utype uu;
        //uu=u.first();

        m0Xd.enq(x.first().d*p_m0);
        m1Xd.enq(x.first().d*p_m1);
        aeXrhosXd.enq(x.first().d*p_Ae*p_rhos);
        xram4noise.enq(x.first().xram);
        tsMtsstar.enq(x.first().ts-p_Tsstar);
        ir.enq(ib+u.first().ic);
        ir4noise.enq(ib+u.first().ic);
        d4noise.enq(x.first().d);
        meStage2.enq(x.first().me);
        meStage3.enq(x.first().me);
        dStage3.enq(x.first().d);

        x.deq();
        u.deq();
        $display("H: Stage1\n") ;
endrule
```

```
rule stage2;
        negDepthR.enq(p_R1-m0Xd.first());
        posDepthR.enq(p_R1-m1Xd.first());
        exponent.enq(tsMtsstar.first()*-p_Aelect
            );
        posDepthMass.enq(meStage2.first()-
            aeXrhosXd.first());

        m0Xd.deq();
        m1Xd.deq();
        tsMtsstar.deq();
        meStage2.deq();
        aeXrhosXd.deq();
        $display("H: Stage2\n") ;
endrule

rule stage3;
        if(dStage3.first()>0)
        action
                rd.enq(posDepthR.first());
                lc.enq(posDepthMass.first());
        endaction
        else
        action
                rd.enq(negDepthR.first());
                lc.enq(meStage3.first());
        endaction
        exponentiator.feed(unpack(pack(exponent.
            first()))));

        dStage3.deq();
        posDepthR.deq();
        posDepthMass.deq();
        negDepthR.deq();
        meStage3.deq();
        exponent.deq();
```

```
        $display("H: Stage3\n") ;
endrule

rule computeR;
        r.enq(rd.first()*exponentiator.fetch());

        rd.deq();
        exponentiator.removeresult();
        $display("H: computeR\n") ;
endrule

rule computeUbVolt;
        ubvolt.enq(r.first()*ir.first());

        r.deq();
        ir.deq();
        $display("H: computeUbVolt\n") ;
endrule

rule computeVolt;
        volt.enq(ubvolt.first()+voltb);

        ubvolt.deq();
        $display("H: computeVolt\n") ;
endrule

rule addnoise;
        Ytype measurement;
        MeasNoisetype noise;

        noise=n.first();

        measurement.d=d4noise.first()+noise.d;
        measurement.xram=xram4noise.first()+
            noise.xram;
        measurement.ir=ir4noise.first()+noise.ir
            ;
```

```
                measurement.lc=lc.first()+noise.lc;
                measurement.volt=volt.first()+noise.volt
                    ;
                y.enq(measurement);

                d4noise.deq();
                xram4noise.deq();
                ir4noise.deq();
                lc.deq();
                volt.deq();
                n.deq();
                $display("H: addnoise\n") ;
        endrule
        /*
        rule displaystuff;
        $write( "result is " ) ; fxptWrite( 10, ir.first
           () ) ; $display("") ;
        endrule
        */

        method Action    measure(Xtype xin ,Utype uin ,
           MeasNoisetype nin);
                x.enq(xin);
                u.enq(uin);
                n.enq(nin);
                //$display("method measure") ;
        endmethod

        method Ytype     fetchMeasurement();
                return y.first();
        endmethod

        method Action    removeMeasurement();
                y.deq();
        endmethod

endmodule
```

```
endpackage: HBRAM
```

1.2.7 Fpiped.bsv

```
/**
* File : Fpiped.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
*       Computes the ESR's physical model in a pipelined
    fashion
*
*       The inverses of deltas are computed through an
   inversion farm
*       Noise is added to the inputs
*       The dynamics are computed
*       The dynamics are integrated in the state vectors
    (=particles)
*/

package Fpiped;

import FixedPoint :: * ;
import ExponFix::*;
import FIFO::*;
import BRAMFIFO :: * ;
import Params::*;
import Vector :: * ;
import Types::*;
import invFarm::*;

/**
abstract interface F_IFC
Description:
```

```
        nbparticles is the number of particles that need
            to be generated
        (also the number of states to fetch/update)

        init:           set the initial state of the
           system (only done once)
        evolve:         ask to compute all the new
           states for a new command vector,
                        applied during ts seconds.
        fetchNewState:  Get the value of a new state
        removeNewStateL:remove the new state value from
           the pipeline to make room
                        for another value.
*/
interface F_IFC#(numeric type nbparticles);
        method Action   init(Xtype xvect);
        method Action   evolve(Utype u, Datatype ts);
        method Xtype    fetchNewState();
        method Action   removeNewState();
endinterface

/**
interface Dyn_IFC
Description:
        has two seperate input methods so that values
           can be preloaded
        into the pipeline before the inverses of Delta
           are fed, since their
        computation takes many cycles. This saves a few
           cycles.
        feedRest should be used until the pipeline is
           full. It block automatically.
        feedInvDelta should be used to feed the inverses
            of Delta into the pipeline.

        feedInvDelta:   feed a new inverted delta
        feedRest:       feed all the other inputs
```

```
        fetchRates:     return the computed rates
        acknowledge:    remove the computed rates from
           the pipeline to make room
                        for the next rates.
*/
interface Dyn_IFC;
        method Action   feedInvDelta(X1type invDelta);
        method Action   feedRest(X2type x_Ts, X3type x_D
            ,Utype u ,ProcessNoisetype nvect);
        method Xtype    fetchRates();
        method Action   acknowledge();
endinterface

/**
function integrateOverTime
Description:
        Updates the state by adding the chamges for this
            step
*/
function Xtype integrateOverTime(Xtype xvect ,Datatype
   ts, Xtype rates);
        Xtype newX;

        newX.delta=xvect.delta+rates.delta*ts;
        newX.ts=xvect.ts+rates.ts*ts;
        newX.d=xvect.d+rates.d*ts;
        newX.xram=xvect.xram+rates.xram*ts;
        newX.me=xvect.me+rates.me*ts;

        return newX;
endfunction: integrateOverTime


/**
module mkDyn
Description:
```

```
        Computes the rates of change of the state
           variables in a pipelined fashion
*/
module mkDyn(Dyn_IFC);
        //Precomputed constants
        Datatype c1=0.003083633189672;
        Datatype c2=1.980324083561251e-04;
        Datatype c3=5.760942788541822e-04;
        Datatype c4=1.198717372411520e-05;
        Datatype c5=2.777777777777778;

        //bias
        Integer vramb=0;
        Integer voltb=0;
        Integer ib=0;

        Reg#(Bool)      available       <- mkReg(True);
        Reg#(Utype)     uReg            <- mkReg(?);

        //FIFOS
        FIFO#(ProcessNoisetype) fifo_Noise
           <- mkSizedBRAMFIFO(9);
        FIFO#(X1type) fifo_InvDelta
           <- mkSizedBRAMFIFO(9);
        FIFO#(X2type) fifo_Ts4computeQMandQS
           <- mkSizedBRAMFIFO(9);
        FIFO#(X2type) fifo_Ts4computeExponent
           <- mkSizedBRAMFIFO(9);
        FIFO#(X3type) fifo_D
           <- mkSizedBRAMFIFO(9);
        FIFO#(Datatype) fifo_Qm
           <- mkLFIFO();
        FIFO#(Datatype) fifo_Qs
           <- mkLFIFO();
        FIFO#(Datatype) fifo_Rd
           <- mkSizedBRAMFIFO(7);
```

```
FIFO#(Datatype) fifo_Exponent
   <- mkLFIFO();
FIFO#(Datatype) fifo_QmQs
   <- mkSizedBRAMFIFO(3);
FIFO#(Datatype) fifo_Pm
   <- mkSizedBRAMFIFO(9);
FIFO#(Datatype) fifo_R
   <- mkLFIFO();
FIFO#(Datatype) fifo_I4Volt
   <- mkSizedBRAMFIFO(5);
FIFO#(Datatype) fifo_I4P
   <- mkSizedBRAMFIFO(5);
FIFO#(Datatype) fifo_Vram
   <- mkSizedBRAMFIFO(5);
FIFO#(Datatype) fifo_Volt
   <- mkLFIFO();
FIFO#(Datatype) fifo_P
   <- mkLFIFO();
FIFO#(Datatype) fifo_Sdot
   <- mkLFIFO();
FIFO#(Datatype) fifo_ddot
   <- mkLFIFO();
FIFO#(Datatype) fifo_Medot
   <- mkLFIFO();
FIFO#(Datatype) fifo_Xramdot
   <- mkSizedBRAMFIFO(5);//mkSizedFIFO(valueOf(
   nbparticles)+3);
FIFO#(Datatype) fifo_Tsdot
   <- mkLFIFO();
FIFO#(Datatype) fifo_Deltadot
   <- mkSizedBRAMFIFO(3);

//Instanciate modules
Exp_IFC#(Isize,Fsize) exponentiator <-
   mkExponFix();

rule addnoise;
```

```
        ProcessNoisetype noise=fifo_Noise.first;

        Datatype i=uReg.ic+noise.ic+fromInteger(
            ib);
        Datatype vram=uReg.vramc+noise.vramc+
            fromInteger(vramb);

        fifo_Vram.enq(vram);
        fifo_Xramdot.enq(vram);
        fifo_I4Volt.enq(i);
        fifo_I4P.enq(i);
        fifo_Noise.deq;

        $display("addnoise\n") ;
endrule

rule computeQMandQS;
        X2type x_ts=fifo_Ts4computeQMandQS.first
            ;

        fifo_Qm.enq((x_ts-p_Tm)*p_Ae*p_He);
        fifo_Qs.enq(p_Hs*2*p_Pi*p_ri*p_hs0*(x_ts
            -p_Tss));
        fifo_Ts4computeQMandQS.deq();

        $display("computeQMandQS\n") ;
endrule

rule computeRd;
        X3type x_D=fifo_D.first;
        if(x_D<p_dInflection)
                fifo_Rd.enq(p_R1-p_m0*x_D);
        else
                fifo_Rd.enq(p_R1-p_m1*x_D);
        fifo_D.deq();
        $display("computeRd\n") ;
endrule
```

```
rule computeExponent;
        X2type x_ts=fifo_Ts4computeExponent.
            first;

        fifo_Exponent.enq((x_ts-p_Tsstar)*-
            p_Aelect);
        fifo_Ts4computeExponent.deq();
        $display("computeExponent\n") ;
endrule

rule loadexponential;
        exponentiator.feed(fifo_Exponent.first()
            );
        fifo_Exponent.deq();
        $display("loadexponential\n") ;
endrule

rule computePmAndQmplusQs;
        fifo_Pm.enq(fifo_Qm.first()*c1); //mur
            =0;
        fifo_QmQs.enq(fifo_Qm.first()+fifo_Qs.
            first());

        fifo_Qm.deq();
        fifo_Qs.deq();
        $display("computePmAndQmplusQs\n") ;
endrule

rule computeR;
        fifo_R.enq(fifo_Rd.first()*exponentiator
            .fetch());

        fifo_Rd.deq();
        exponentiator.removeresult();
        $display("computeR\n") ;
endrule
```

```
rule computeVolt;
        fifo_Volt.enq(fifo_R.first()*fifo_I4Volt
            .first()+fromInteger(voltb));

        fifo_R.deq();
        fifo_I4Volt.deq();
        $display("computeVolt\n") ;
endrule

rule computeP;
        fifo_P.enq(fifo_Volt.first()*fifo_I4P.
            first());

        fifo_Volt.deq();
        fifo_I4P.deq();
        $display("computeP\n") ;
endrule

rule computeTsdot;
        fifo_Tsdot.enq((fifo_P.first()-fifo_QmQs
            .first())*c4);

        fifo_P.deq();
        fifo_QmQs.deq();
        $display("computeTsdot\n") ;
endrule

rule computeSdotandDeltadot;
        X1type invDelta=fifo_InvDelta.first();
        fifo_Sdot.enq(fifo_Pm.first()*c2+
            fxptTruncate(invDelta)*p_alphar*
            p_Csd0);
        fifo_Deltadot.enq(fifo_Pm.first()*c3+
            fxptTruncate(invDelta)*p_alphar*p_Cdd
            );
```

```
        fifo_Pm.deq();
        fifo_InvDelta.deq();
        $display("computeSdotandDeltadot\n") ;
endrule

rule computeMedotandddot;
        fifo_Medot.enq(fifo_Sdot.first()*-p_rhom
            *p_Ae);
        fifo_ddot.enq(fifo_Vram.first()*c5-
            fifo_Sdot.first());

        fifo_Sdot.deq();
        fifo_Vram.deq();
        $display("computeMedotandddot\n") ;
endrule

method Action   feedInvDelta(X1type invDelta);
        fifo_InvDelta.enq(invDelta);
endmethod

method Action   feedRest(X2type x_Ts, X3type x_D
    ,Utype u ,ProcessNoisetype nvect);
        fifo_Ts4computeQMandQS.enq(x_Ts);
        fifo_Ts4computeExponent.enq(x_Ts);
        fifo_D.enq(x_D);
        uReg    <=u;
        fifo_Noise.enq(nvect);
endmethod

method Xtype    fetchRates();
        Xtype rates= Xtype {
                delta : fifo_Deltadot.first ,
                ts :    fifo_Tsdot.first ,
                d :     fifo_ddot.first ,
                xram :  fifo_Xramdot.first ,
                me :    fifo_Medot.first
        };
```

```
                return rates;
        endmethod

        method Action   acknowledge();
                        fifo_Deltadot.deq();
                        fifo_Tsdot.deq();
                        fifo_ddot.deq();
                        fifo_Xramdot.deq();
                        fifo_Medot.deq();
        endmethod

endmodule

/**
module mkFpiped

Description:
        links to gether the parts of the physical model:
                The inverter farm
                The dynamics module
                The integration module
*/
module mkFpiped(F_IFC#(nbparticles));

        Reg#(Bool)      available       <-mkReg(True);
        Xtype xdefault = Xtype
        {
                delta : p_delta0,
                ts :    p_Ts0,
                d :     p_d0,
                xram :  0,
                me :    100000
        };

        ProcessNoisetype pn = ProcessNoisetype
        {
                ic : 0,
```

```
        vramc : 0
};

Reg#(Vector#(nbparticles,Xtype))
    xvectReg          <- mkReg(replicate(xdefault))
    ;
Reg#(Vector#(nbparticles,ProcessNoisetype))
    pnvectReg         <- mkReg(replicate(pn));

Reg#(Utype)       uReg
   <- mkReg(?);
Reg#(Datatype)    tsReg
   <- mkReg(0);

Reg#(UInt#(TAdd#(TLog#(nbparticles),1)))
    invCount          <-mkReg(0);
Reg#(UInt#(TAdd#(TLog#(nbparticles),1)))
    dynCount          <-mkReg(0);
Reg#(UInt#(TAdd#(TLog#(nbparticles),1)))
    intCount          <-mkReg(0);
//Reg#(UInt#(TLog#(nbparticles)))
    noiseCount        <-mkReg(0);

//SUBMODULES
InvFarm_IFC#(Isize,Fsize,TAdd#(TMul#(Fsize,2),4)
    ) invFarm <- mkInvFarm;
Dyn_IFC dynamics <- mkDyn;


FIFO#(Xtype)      fifo_newX
   <- mkLFIFO();

rule feedInvFarm(invCount<fromInteger(valueOf(
    nbparticles))&&!available);
        invFarm.submit(xvectReg[invCount].delta)
            ;
        invCount<=invCount+1;
```

```
        //$display("mkFpiped: feedInvFarm");
endrule

rule feedDyn( !available && (dynCount<
   fromInteger( valueOf(nbparticles) )) );
        Xtype xtemp=xvectReg[dynCount];

        dynamics.feedRest(xtemp.ts,xtemp.d,uReg,
           pnvectReg[dynCount]);
        dynCount<=dynCount+1;
        //$display("mkFpiped: feedDyn");
endrule

rule feedInvDeltaIntoDyn(!available);
        Maybe#(X1type)rawres=invFarm.result();
        X1type res=fromMaybe(unpack('1),rawres);

        invFarm.acknowledge();
        if(isValid(rawres))
        action
                dynamics.feedInvDelta(-res);
                //$display("     Result = %b",
                   res);
                //$write( "mkFpiped: Result is "
                    ); fxptWrite( 7, res );
                   $display("" );
        endaction
        else
                //$display("mkFpiped: INVALID
                   DIVISION");
endrule

rule integrate( !available && (intCount<
   fromInteger( valueOf(nbparticles) )) );
        Xtype rates=dynamics.fetchRates();
        fifo_newX.enq(integrateOverTime(xvectReg
           [intCount],tsReg,rates));
```

```
            intCount<=intCount+1;
            dynamics.acknowledge();
            //$display("mkFpiped: integrate");

            //$write( "mkFpiped_integrate: xvect.
               delta is " ) ; fxptWrite( 7, rates.
               delta ) ; $display("" ) ;
            //$write( "mkFpiped_integrate: xvect.ts
               is " ) ; fxptWrite( 7, rates.ts ) ;
               $display("" ) ;
            //$write( "mkFpiped_integrate: xvect.d
               is " ) ; fxptWrite( 7, rates.d ) ;
               $display("" ) ;
            //$write( "mkFpiped_integrate: xvect.
               xram is " ) ; fxptWrite( 7, rates.
               xram ) ; $display("" ) ;
            //$write( "mkFpiped_integrate: xvect.me
               is " ) ; fxptWrite( 7, rates.me ) ;
               $display("" ) ;
    endrule

    rule makeAvailable( !available && (intCount>=
       fromInteger( valueOf(nbparticles) )) );
            available<=True;
    endrule

    method Action    init(Xtype xvect);
            xvectReg <= replicate(xvect);
    endmethod

    method Action    evolve(Utype u, Datatype ts) if(
       available);
            available        <=False;
            uReg     <=u;
            tsReg    <=ts;

            invCount<=0;
```

```
                dynCount<=0;
                intCount<=0;
        endmethod

        method Xtype     fetchNewState();
                return fifo_newX.first();
        endmethod

        method Action    removeNewState();
                fifo_newX.deq();
        endmethod
endmodule



endpackage: Fpiped
```

1.2.8 FHpiped.bsv

```
/**
* File : FHpiped.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
*       Computes the ESR's physical (F) and measurement
   (H) models in a
        pipelined fashion. F and H are connected through
            a rule that
        states that when a result is available form F,
           this result
        is taken and fed into H.

        Upon instanciation of the FHpiped module, a
           number of copies
```

```
        is provided. This will create parallel copies of
            FH computation
        hardware and rules to connect F and H in each
           copy
*/

package FHpiped;
import FixedPoint :: * ;
import Vector :: * ;
import Types :: * ;
import HBRAM :: * ;
import Fpiped :: * ;
import Params :: *;

/**
interface FHpiped_IFC
Parameters:
        nc: number of FH copies (will instanciate the
           hardware nc times)
        np: number of particles per copies
Methods:
        init:           set the initial state of the ESR
            (should only be used once)
        startNewStep:   ask to compute all the particles
            with a new command
                        vector and for this step of
                           length ts.
        fetchMeasurements:      returns a vector
           containing all the simulation results
                                in the pipeline for each
                                    copy
        removeMeasurement:      removes the current
           results from the pipeline of each copy
                                (only executed if all
                                   the output FIFOs are
                                   not empty)
*/
```

```
interface FHpiped_IFC#(numeric type nc, numeric type np)
   ;
        method Action            init(Xtype xinit);
        method Action            startNewStep(Utype uin,
           Datatype ts);
        method Vector#(nc,Ytype)
           fetchMeasurements();
        method Action            removeMeasurement();
endinterface

module mkFHpiped(FHpiped_IFC#(nc,np));
        Integer nb_copies=valueOf(nc);
        Integer nb_particles=valueOf(np);
        //initial state
        Xtype xdefault = Xtype
        {
                delta : p_delta0,
                ts :    p_Ts0,
                d :     p_d0,
                xram :  0,
                me :    100000
        };

        //process noise
        ProcessNoisetype pn = ProcessNoisetype
        {
                ic : 0,
                vramc : 0
        };

        //measurement noise
        MeasNoisetype mn = MeasNoisetype
        {
                d : 0,
                xram : 0,
                ir : 0,
```

```
        lc : 0,
        volt : 0
};


Reg#(Utype) uReg                        <- mkReg
    (Utype{ic:0,vramc:0});
Reg#(MeasNoisetype) mnvectReg           <- mkReg
    (mn);


F_IFC#(np)      f[nb_copies];
H_IFC           h[nb_copies];

for(Integer i=0;i<nb_copies;i=i+1)
begin
        f[i] <- mkFpiped;
        h[i] <- mkH;
end

for(Integer i=0;i<nb_copies;i=i+1)
begin
        rule feedNewStateIntoHandRemoveFromF_i;
                f[i].removeNewState();
                h[i].measure(f[i].fetchNewState
                    () ,uReg ,mnvectReg);
                $display("
                    feedNewStateIntoHandRemoveFromF
                    \n") ;

        $write( "
            feedNewStateIntoHandRemoveFromF_i: f[
            i].fetchNewState().delta is " ) ;
            fxptWrite( 7, f[i].fetchNewState().
            delta ) ; $display("" ) ;
        $write( "
            feedNewStateIntoHandRemoveFromF_i: f[
```

```
            i].fetchNewState().ts is " ) ;
            fxptWrite( 7, f[i].fetchNewState().ts
             ) ; $display("" ) ;
        $write( "
            feedNewStateIntoHandRemoveFromF_i: f[
            i].fetchNewState().d is " ) ;
            fxptWrite( 7, f[i].fetchNewState().d
            ) ; $display("" ) ;
        $write( "
            feedNewStateIntoHandRemoveFromF_i: f[
            i].fetchNewState().xram is " ) ;
            fxptWrite( 7, f[i].fetchNewState().
            xram ) ; $display("" ) ;
        $write( "
            feedNewStateIntoHandRemoveFromF_i: f[
            i].fetchNewState().me is " ) ;
            fxptWrite( 7, f[i].fetchNewState().me
             ) ; $display("" ) ;
        endrule
end
/*
rule getMeasurement; //whenever a measurement is
    avaible
        h.removeMeasurement();
        $write( "xram result is " ) ; fxptWrite(
            10, h.fetchMeasurement().xram ) ;
            $display("\n");

        $display("FHTest: getMeasurement\n") ;
endrule
*/

method Action    init(Xtype xinit);
        for(Integer i=0;i<nb_copies;i=i+1)
        begin
                f[i].init(xinit);
        end
```

```
                /* xvectReg     <= replicate(xinit);
                xvectReg2       <= replicate(xinit); */
                //initialized   <=True;
        endmethod

        method Action   startNewStep(Utype uin, Datatype
            ts);// if (initialized);
                for(Integer i=0;i<nb_copies;i=i+1)
                begin
                        f[i].evolve(uin, ts);
                end
                uReg<=uin;

        endmethod

        method Vector#(nc,Ytype)
           fetchMeasurements(); //When both measurements
            are available
                Vector#(nc,Ytype) vect;
                for(Integer i=0;i<nb_copies;i=i+1)
                begin
                        vect[i]=h[i].fetchMeasurement();
                end
                return vect;
        endmethod

        method Action   removeMeasurement();
                for(Integer i=0;i<nb_copies;i=i+1)
                begin
                        h[i].removeMeasurement();
                end
        endmethod
endmodule: mkFHpiped

endpackage: FHpiped
```

1.2.9 FHpipedTb.bsv

```
package FHpipedTb;

import FixedPoint :: * ;
import Vector :: * ;
import Types::*;
import FHpiped::*;
import Params :: *;


        typedef 50 NbParticles;
        typedef 1 NbCopies;


module mkFHpipedTb(Empty);
        Utype u;
        Reg#(Vector#(NbParticles,Vector#(NbCopies,Ytype)
           )) yvectReg <- mkReg(?);

        Integer np=valueof(NbParticles);
        Integer nc=valueof(NbCopies);

        Integer nbSteps=3;

        Datatype ts=fromRational(2,15); //sampling time
        //command vector
        u.ic=200;
        u.vramc=20;

        //init state
        Xtype xinit = Xtype
        {
                delta : p_delta0,
                ts :    p_Ts0,
                d :     p_d0,
                xram :  0,
                me :    100000
```

```
};

FHpiped_IFC#(NbCopies,NbParticles) fh <-
    mkFHpiped();

Reg#(Utype) uReg <- mkReg(u);

Reg#(UInt#(50)) simCount <- mkReg(0);
Reg#(UInt#(50)) fetchCount <- mkReg(0);
Reg#(UInt#(20)) count <- mkReg(0);

rule init(count==0);
        fh.init(xinit);
endrule

rule feedNewU(simCount<fromInteger(nbSteps));
        fh.startNewStep(uReg,ts);
        simCount<=simCount+1;
        $display("mkFHpipedTb: feedNewU");
endrule

rule fetchAndDisplaySimulations;//(simCount<
    fromInteger(np));
        Vector#(NbCopies,Ytype) measurements=fh.
            fetchMeasurements();

        yvectReg[fetchCount]<=measurements;
        fh.removeMeasurement();
        $display("mkFHpipedTb: FETCH");
        fetchCount<=fetchCount+1;

        for(Integer j=0;j<fromInteger(nc);j=j+1)
        begin
                        $write("yvectReg[step:
                            %d][copy: %d].d= ",
                            fetchCount,j);
                            fxptWrite(10,
```

```
                                        measurements[j].d ) ;
                                         $display("\n") ;
                                    $write( "yvectReg[step:
                                       %d][copy: %d].xram=
                                       ",fetchCount,j ) ;
                                       fxptWrite( 10,
                                       measurements[j].xram
                                       ) ; $display("\n") ;
                                    $write( "yvectReg[step:
                                       %d][copy: %d].ir= ",
                                       fetchCount,j ) ;
                                       fxptWrite( 10,
                                       measurements[j].ir )
                                       ; $display("\n") ;
                                    $write( "yvectReg[step:
                                       %d][copy: %d].lc= ",
                                       fetchCount,j ) ;
                                       fxptWrite( 10,
                                       measurements[j].lc )
                                       ; $display("\n") ;
                                    $write( "yvectReg[step:
                                       %d][copy: %d].volt=
                                       ",fetchCount,j ) ;
                                       fxptWrite( 10,
                                       measurements[j].volt
                                       ) ; $display("\n") ;
                end
        endrule

/*      rule displaySimulations(fetchCount>=fromInteger(
   nc));

                for(Integer j=0;j<fromInteger(nc);j=j+1)
                begin
                        for(Integer i=0;i<fromInteger(np
                           );i=i+1)
                        action
```

```
                                        $write( "yvectReg[%d].d=
                                            ",i ) ; fxptWrite(
                                           10, yvectReg[i][j].d
                                           ) ; $display("\n") ;
                                        $write( "yvectReg[%d].
                                           xram= ",i ) ;
                                           fxptWrite( 10,
                                           yvectReg[i][j].xram )
                                            ; $display("\n") ;
                                        $write( "yvectReg[%d].ir
                                           = ",i ) ; fxptWrite(
                                           10, yvectReg[i][j].ir
                                            ) ; $display("\n") ;
                                        $write( "yvectReg[%d].lc
                                           = ",i ) ; fxptWrite(
                                           10, yvectReg[i][j].lc
                                            ) ; $display("\n") ;
                                        $write( "yvectReg[%d].
                                           volt= ",i ) ;
                                           fxptWrite( 10,
                                           yvectReg[i][j].volt )
                                            ; $display("\n") ;
                                endaction
                        end
                endrule */

                rule cycleCount;
                        count <= count+1;
                        $display("mkFHpipedTb: ----------------
                           Count = %d ----------\n",count );
                endrule

                rule stop(count>350);
                        $finish(0);
                endrule

endmodule: mkFHpipedTb
```

```
endpackage: FHpipedTb
```

1.2.10 MultipleFH.bsv

```
/**
* File : MultipleFH.bsv
* Date created: not sure
* Last update: June 09
* Author: Thomas Lauzon
* Description:
*       This is the Top module.
        All it does is instanciating the FHpiped module
           with a given number of parallel copies
        and a nuber of particles and provides an
           interface to start the computations and fetch
        the results.
*/
package MultipleFH;

import FixedPoint :: * ;
import Vector :: * ;
import Types::*;
import FHpiped::*;
import Params :: *;

        typedef 150 NbParticles;
        typedef 2 NbCopies;

/**
interface MultFHpiped_IFC
methods:
        init: set an initial state (should only be used
           once, at the beginning
        startNewStep: request a simulation for a given
           command (uin) applied for ts seconds
```

```
        fetchMeasurments: returns all the observations
           in the output FIFOs of all simulators
        removeMeasurement: removes all the observations
           in the output FIFOs of all simulators


*/
interface MultFHpiped_IFC;
        method Action             init(Xtype xinit);
        method Action             startNewStep(Utype uin,
           Datatype ts);
        method Vector#(NbCopies,Ytype)
           fetchMeasurements();
        method Action             removeMeasurement();
endinterface

(* synthesize *)
module mkMultipleFH(MultFHpiped_IFC);

        FHpiped_IFC#(NbCopies,NbParticles) fh <-
           mkFHpiped();

        method Action    init(Xtype xinit);
                         fh.init(xinit);
        endmethod

        method Action    startNewStep(Utype uin, Datatype
            ts);
                         fh.startNewStep(uin, ts);
        endmethod

        method Vector#(NbCopies,Ytype)
           fetchMeasurements(); //When both measurements
            are available
                         return fh.fetchMeasurements();
        endmethod
```

```
        method Action    removeMeasurement();
                         fh.removeMeasurement();
        endmethod
endmodule
endpackage: MultipleFH
```

1.2.11 MultipleFHTb.bsv

```
package MultipleFHTb;

import FixedPoint :: *;
import Vector :: *;
import Types::*;
import MultipleFH::*;
import Params :: *;


        //typedef 3 NbParticles;
        //typedef 3 NbCopies;


module mkMultipleFHTb(Empty);
        Utype u;
        //Reg#(Vector#(NbParticles,Vector#(NbCopies,
            Ytype))) yvectReg <- mkReg(?);
        //Reg#(Vector#(NbCopies,Ytype)) yvectReg <-
            mkReg(?);

        Integer np=valueof(NbParticles);
        Integer nc=valueof(NbCopies);

        Integer nbSteps=3;

        Datatype ts=fromRational(2,15); //sampling time
        //command vector
        u.ic=200;
```

```
u.vramc=20;

//init state
Xtype xinit = Xtype
{
        delta : p_delta0,
        ts :    p_Ts0,
        d :     p_d0,
        xram :  0,
        me :    100000
};

MultFHpiped_IFC fh <-mkMultipleFH();

Reg#(Utype) uReg <- mkReg(u);

Reg#(UInt#(50)) simCount <- mkReg(0);
Reg#(UInt#(50)) fetchCount <- mkReg(0);
Reg#(UInt#(20)) count <- mkReg(0);

rule init(count==0);
        fh.init(xinit);
endrule

rule feedNewU(simCount<fromInteger(nbSteps));
        fh.startNewStep(uReg,ts);
        simCount<=simCount+1;
        $display("mkMultipleFHTb: feedNewU" );
endrule

rule removeAndDisplaySimulationResults;//(
   simCount<fromInteger(np));
        Vector#(NbCopies,Ytype) measurements=fh.
           fetchMeasurements();

        //yvectReg[fetchCount]<=measurements;
        //yvectReg<=measurements;
```

```
fh.removeMeasurement();
$display("mkMultipleFHTb: FETCH");
fetchCount<=fetchCount+1;

for(Integer j=0;j<fromInteger(nc);j=j+1)
begin
                $write("yvectReg[
                    fetchCount: %d][copy:
                     %d].d= ",fetchCount,
                    j); fxptWrite(10,
                    measurements[j].d);
                     $display("\n");
                $write("yvectReg[
                    fetchCount: %d][copy:
                     %d].xram= ",
                    fetchCount,j);
                    fxptWrite(10,
                    measurements[j].xram
                    ); $display("\n");
                $write("yvectReg[
                    fetchCount: %d][copy:
                     %d].ir= ",fetchCount
                    ,j); fxptWrite(10,
                     measurements[j].ir)
                     ; $display("\n");
                $write("yvectReg[
                    fetchCount: %d][copy:
                     %d].lc= ",fetchCount
                    ,j); fxptWrite(10,
                     measurements[j].lc)
                     ; $display("\n");
                $write("yvectReg[
                    fetchCount: %d][copy:
                     %d].volt= ",
                    fetchCount,j);
                    fxptWrite(10,
                    measurements[j].volt
```

```
                                        ) ; $display("\n") ;
                end
        endrule

/*      rule displaySimulations(fetchCount>=fromInteger(
    nc));

                for(Integer j=0;j<fromInteger(nc);j=j+1)
                begin
                        for(Integer i=0;i<fromInteger(np
                            );i=i+1)
                        action
                                $write( "yvectReg[%d].d=
                                    ",i ) ; fxptWrite(
                                    10, yvectReg[i][j].d
                                    ) ; $display("\n") ;
                                $write( "yvectReg[%d].
                                    xram= ",i ) ;
                                    fxptWrite( 10,
                                    yvectReg[i][j].xram )
                                     ; $display("\n") ;
                                $write( "yvectReg[%d].ir
                                    = ",i ) ; fxptWrite(
                                    10, yvectReg[i][j].ir
                                     ) ; $display("\n") ;
                                $write( "yvectReg[%d].lc
                                    = ",i ) ; fxptWrite(
                                    10, yvectReg[i][j].lc
                                     ) ; $display("\n") ;
                                $write( "yvectReg[%d].
                                    volt= ",i ) ;
                                    fxptWrite( 10,
                                    yvectReg[i][j].volt )
                                     ; $display("\n") ;
                        endaction
                end
        endrule */
```

```
        rule cycleCount;
                count <= count+1;
                $display("mkMultipleFHTb: ----------------
                    Count = %d ----------\n",count );
        endrule

        rule stop(count>600);
                $finish(0);
        endrule

endmodule: mkMultipleFHTb

endpackage: MultipleFHTb
```

Bibliography

[1] S. Ahn. *Modeling, estimation, and control of electroslag remelting process*. PhD thesis, The University of Texas at Austin, 2005.

[2] M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *Signal Processing, IEEE Transactions on*, 50(2):174–188, Feb 2002.

[3] I. Bluespec. *Bluespec SystemVerilog Reference Guide, 2008*.

[4] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168.

[5] A. Doucet and N. De Freitas. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[6] A. Gothandaraman, G.D. Peterson, G. Lee Warren, R.J. Hinde, and R.J. Harrison. FPGA acceleration of a quantum Monte Carlo application. *Parallel Computing*, 2008.

[7] E.A. Lee. Cyber-physical systems-are computing foundations adequate. 2006.

[8] E.A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. *University of California at Berkeley, Technical Report No. UCB/EECS-2007-72, May*, 2007.

[9] E.E. Swartzlander. *Computer Arithmetic*. CRC Press, 1993.

[10] R. Yates. Fixed-point arithmetic: An introduction. *Digital Audio Signal Processing*.

Vita

Thomas Lauzon was born in Paris, France on 23 August 1982. He is the son of Véronique Schumpp and Charles Lauzon. After completing high school at the École Active Bilingue Jeannine Manuel (EABJM), a bilingual school in Paris, he entered the École d'Ingénieurs en Électronique et Électrotechnique (ESIEE-Paris) in Noisy-le-Grand, France. During this time he focused on embedded systems and interned at an automobile manufacturing plant, an IT firm, a Brazilian university and an aeronautics company. Upon completion in 2005, he obtained a Diplôme d'Ingénieur, which is a European Master's degree. In 2006, Mr. Lauzon worked for a technical consulting company that placed him on avionics projects. In August 2006 he started graduate studies at the University of Texas at Austin.

Permanent address: 27 rue Letellier
75015 Paris, France

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.