ECE1373 VLSI Systems Design project report
MonteCarlo-based Channel Estimator (MCCE)

Mahdi Shabany
Hassan Shojania
Jing Zhang

June 18, 2004

Contents

1 Introduction

Channel estimation is an important building block in modern digital communication systems and has vast applications in wireless communication systems. Advanced signal processing technologies, such as 2D channel estimation and Kalman filtering [1][2], have been applied widely in this topic, and some implementation attempts have resulted in digital functioning chips. The purpose of this project is to implement a channel estimator based on the so called Sequential Monte Carlo Method, which is a promising advanced solution for various estimation problems in communication systems.

This project covers the design and verification activities from the algorithm evaluation down to the running FPGA demo, involving the design activities in the algorithm level, architecture level and circuit level. The implementation has been carried out following the guidelines presented in the course.

In the following sections, first the algorithm is explained. The methodology of the design process is described in great detail in section 3. Section 4 presents the macro architecture of the design. Design characteristics are summarized, and some relevant issues such as design problem and design time distributions are also discussed in section 5. Moreover, specifications of the DEMO system in terms of the system hardware/software is explained in section 6. In section 7 some problems encountered during the course of project are briefly discussed. Finally the conclusion section summarizes the design feature, design procedure and design experience. Appendixes provide our design's architecture and register specifications, PAR report, floorplan diagram and the hierarchy of the code checked into the course CVS area.

2 Algorithm Explanation

In wireless communication systems, the transmitted signal is corrupted by the fading characteristics of the channel along with the Additive White Gaussian Noise (AWGN). To recover the information in the transmitted signal correctly, it is important to know the Channel Impulse Response

(CIR) which is achieved by channel estimation.

The particular channel estimation method implemented in our design is based on two theories. One is the framework for dynamic state estimation problem (of which channel estimation is a particular class of problem) based on the Sequential Monte Carlo Method and the other one is a channel model presented in [3] for a general wireless channel with given maximum Doppler frequency shift.

The Sequential Monte Carlo approach relys on constructing the probability density function of the desired state based on all observed/known information. On the other hand, for the channel model presented in [3], the fading characteristic of the channel can be modeled by a Gaussian noise, namely the process noise $w(t)$ with a known variance of Q, fed into a typical low-pass filter, like the Batterworth Filter. The frequency response and impulse response of the channel can be respectively written as

$$H(Z) = \frac{D}{AZ^{-1} + BZ^{-2} + CZ^{-3} + 1} W(Z) \tag{1}$$

and

$$h(t) = -Ah(t-1) - Bh(t-2) - Ch(t-3) + Dw(t) \tag{2}$$

where the coefficients A, B, C and D are determined based on the maximum Doppler frequency shift in the system along with the sampling frequency of the channel.

Using this channel model, the received signal is the convolution of $h(t)$ with transmitted data, $p(t)$, plus the AWGN noise of the channel, which is called the measurement noise, $v(t)$. Therefore, the received signal can be expressed as:

$$y(t) = p(t) * h(t) + v(t) \tag{3}$$

Assuming a known transmitted signal, e.g. the pilot signal, channel estimation is carried out based on the observed signal, $y(t)$, using the following idea of Bayesian method. For the first iteration, draw the initial values of $h(t-1), h(t-2)$ and $h(t-3)$ from a Gaussian distribution with known variances and calculate N *a priori* estimations of $h(t)$ based on equation (2)

where $w(t)$ drawn from a Gaussian distribution with known variance, and N is the number of sample space. Using above initial vectors, a predicted vector of $h(t)$ with N entries can be determined. This prediction is compared with the real observation, $y(t)$, and a weight is given to each element of the predicted vector, $\mathbf{H}_{predicted}(t)$. Based on the assigned weight of each element, the *a priori* estimation is either discarded or it survives and repeated several times where the number of repetition is determined by the assigned weight. The higher the weight, it will be repeated more. Meanwhile, one final value of the channel estimation, i.e. the *a posteriori* estimation, is generated as the mean value of all the a posterior estimations using weighted sum. After the first iteration, the *a posterior* prediction vector $\mathbf{H}_{predicted}(t)$ will be used as the *a priori* estimations of $h(t-1)$ for the next iteration. The previous $h(t-1), h(t-2)$ are now shifted as $h(t-2)$ and $h(t-3)$, respectively.

From the viewpoint of the calculation, the process could be described as:

1. Calculate estimation vectors $h_{estimated}(t)$ for the initial time step based on random numbers drawn from a Gaussian distribution with variance of $InitVar$.

   For each time step $t = \{4, 5, ...\}$:

2. Pick a vector of random numbers W from a Gaussian distribution with variance Q (variance of process noise).

3. Calculate prediction vector from

$$\mathbf{H}_{predicted}(t) = -A.\mathbf{H}_{estimated}(t-1) - B.\mathbf{H}_{estimated}(t-2) - C.\mathbf{H}_{estimated}(t-3) + D.W(t). \tag{4}$$

4. Calculate the vector of the estimated received signal based on estimated state (calculated in the previous step) as

$$\mathbf{Y}_{estimated} = PilotData(t).\mathbf{H}_{predicted}(t). \tag{5}$$

   Note that $PilotData(t)$ is a scalar value (either -1 or 1 as is the case in BPSK modulation).

5. Calculate *a posteriori* distribution of the predicted vector using

$$p_j = e^{\frac{-1}{2R}(y_{received}(t)-y_j)^2}, \quad y_j \in \mathbf{Y}_{estimated}. \tag{6}$$

where $y_{received}(t)$ is a scalar value representing the received signal at time t, R is the variance of measurement noise (in a Gaussian distribution), and y_j is one element of $\mathbf{Y}_{estimated}$.

6. Calculate the *importance weight* vector using

$$\mathbf{Q} = \{q_j\}, \qquad q_j = \frac{p_j}{\sum p_i}. \tag{7}$$

Note that sum of the members of $\mathbf{Q}$ equals to unity.

7. Update the state predicted vector $\mathbf{H}_{predicted}(t)$. *Importance weights* vector ($\mathbf{Q}$) is used to filter/resample members of estimation vector. The members with weight less than $\frac{1}{N}$ are eliminated while others repeated in proportion to their weights to fill the whole vector.

8. The final estimated value $h(t)$ is the mean of updated predicted vector $\mathbf{H}_{predicted}(t)$.

9. Go back to step 2 to repeat steps 3 to 9 for the next time steps.

3 Methodology

3.1 Design Flow and Principle

Fig. 1 shows the design flow adopted in this project. From this diagram, it is clear that the following principles are used in the design methodology:

Top-Down:

The overall design flow traverses (abstract) algorithm level, macro architecture level, logic level and physical level design activities.

Simulation Based:

The algorithm level simulation guarantees to solve the right problem and the refinement based on it, the bit-true algorithm model, provides golden reference model for Verilog model simulation, while the Verilog model simulation strives to present correct circuit. In addition, post-PAR simulation makes sure the function of the original RTL code has not been altered.

Iteration Based:

Fig. 1. shows that some iterations exist in every step, and it emphasizes the iterations from chip-level simulation to unit-level implementation. In fact these iterations could go cross other activities too (e.g. from physical level to logic level).

3.2 Major Activities

Some of the major activities in Fig. 1. are elaborated as follows.

3.2.1 Algorithm Exploration

This is the phase to elaborate the algorithm by Matlab simulation, refining and tuning the algorithm such that it has physical and engineering significance and that it is (intuitively) implementable. Meanwhile, this is also the good chance for every team-member to closely "observe" and appreciate the detail of the algorithm.

When this phase was finished, a function specification document was written to record the work as a mile-stone. More importantly, the Matlab model, which involves all the calculations in double-precision arithmetic operation, is ready to be used as the initial golden reference for future use.

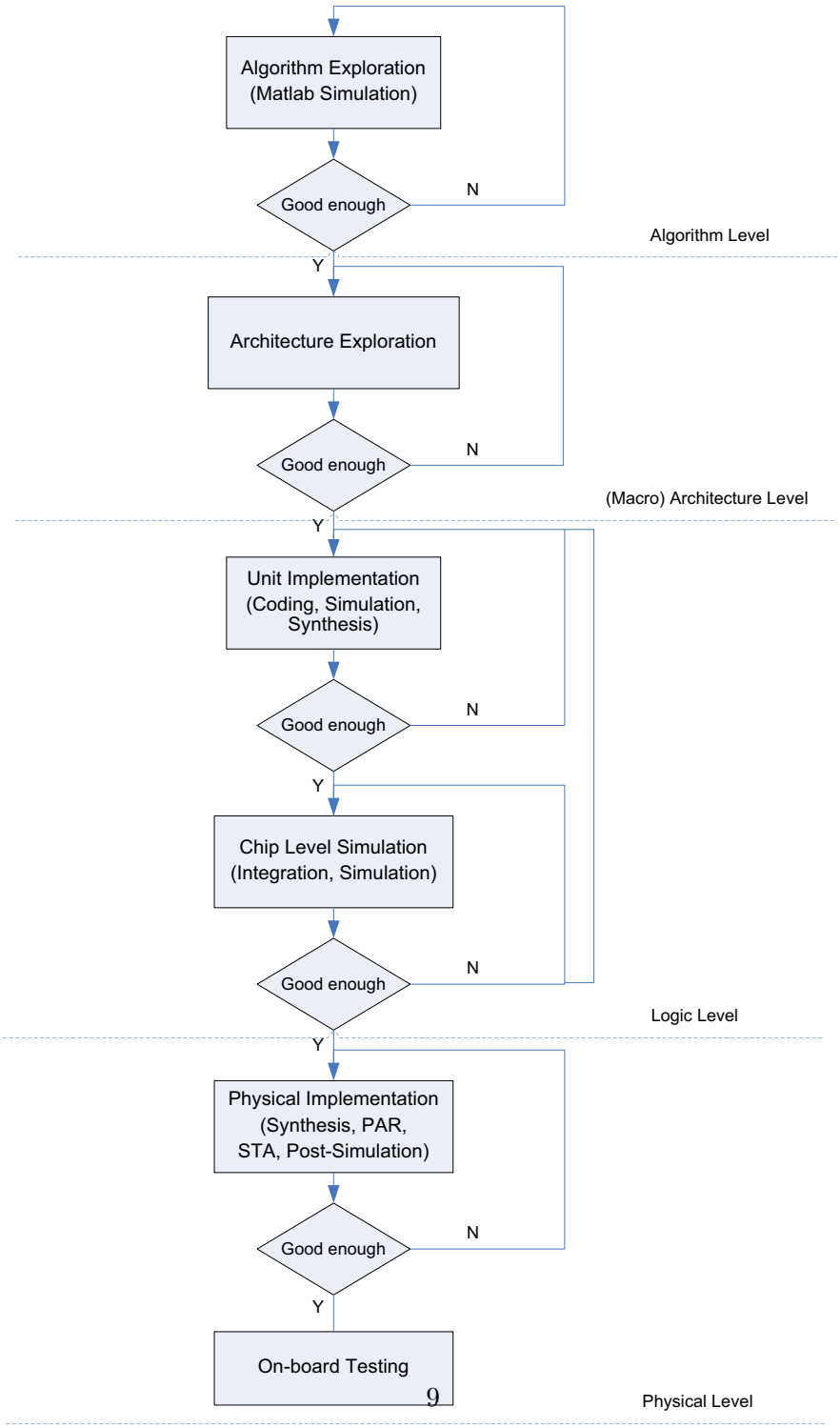

Figure 1: Design Flow of MCCE

3.2.2 Architecture Exploration

This is the phase to study the intrinsic parallelism of the algorithm, propose suitable (macro) architecture alternatives, and make architecture decisions. This phase consists of the following major activities:

Data dependence analysis:

Due to the nature of the algorithm, the implementation will be a piece of calculation intensive (instead of control intensive) hardware. To explore the parallelism of this kind of algorithm, data dependence graph is a powerful tool. Based on this approach, several architecture alternatives were proposed and analyzed. See Appendix C for details.

Word-length-effect evaluation:

For calculation intensive hardware implementation, it is very important to evaluate the finite-word-length effect. Bit-true simulations are necessary in the early design phase to evaluate the effect of the word-length and make appropriate design decision. According to [4][5], the strategy of bit-true simulation is a two-layered problem, namely the operational level and the decision level. In the operational level, we need to implement a simulation mechanism for bit-true simulation. For instance, we need to emulate the behavior of overflow or saturation for given word-length. In the decision level, we need to follow a suitable methodology to efficiently find the suitable word-length.

In operation level, a set of functions in Matlab was developed to emulate the bit-true behavior of the hardware; while in decision level, analysis method and the comparisons between the bit-true model and the double-precision model are used to facilitate the word length selection.

Physical block definition:

Based on the decision from data dependence analysis and word-length-effect evaluation, the sketch of the architecture is clear. Next the function and interface of each block is defined and documented, as in the Appendix C.

When reaching this point, the overall architecture is defined. An architecture spec document has been made available for future use, and the original Matlab "ideal" functional model has also been transformed into a bit-true model which could facilitate the verification in later phases. The original Matlab code along with the codes for bit-true simulation can be found in *Matlab* sub-directory of checked-in code.

3.2.3 Simulation and Verification

Roughly speaking, there are five kinds of simulations in our design procedure:

Algorithm Simulation:
This is the Matlab Simulation of the algorithm with double-precision floating point representation of the data.

Bit-true Algorithm Simulation:
This is the Matlab Simulation of the algorithm with finite-word-length representation of the data. Originally, this simulation was carried out in relatively high granularity, i.e. the major variables of the Matlab code are bit-true, but the arithmetic operations (e.g. addition, multiplication) between the major variables are not always bit-true. This kind of "bit-true" algorithm simulation is good enough to evaluate the finite-word-length effect. However, later in Verilog model simulation, this model is not precise enough to be the golden reference model. Normally, a third model, the "executable specification" needs to be developed, which might be written in a language that supports parallelism better than Matlab and so looks more like "hardware modeling language", e.g. SystemC. Fortunately, in our case, due to the function of the design, the bit-true algorithm model could be modified by simply lowering the granularity of the bit-true behavior of the model, and be good enough as the golden reference for HDL verification.

Unit-level Functional Simulation:
This is the functional Verilog model simulation at the sub-block level. In

this project, the Device Under Verification (DUV) of unit verification could be a file, or several files that are closely related. In many projects, unit level simulation is carried out in an ad-hoc manner. This means that the stimuli is generated manually, and visual inspection is widely used to check the response of the DUV. In our project, however, the following ideas have been introduced to facilitate systematic and efficient unit-level simulation:

- Self-checking verification
- Transaction level verification [6]

Two kinds of scenarios, namely the manually-crafted simulation and the golden reference model simulation, are generated following the above principle, as stated below.

- **Manual**:

  If the stimuli is to be generated manually, which is necessary for white box verification, then we use some language construct of Verilog (i.e. task) to implement the signal transaction level behavior and let the testcase focus on the content/value of the stimuli. This is possible because all the internal interfaces have been defined in a standard and simple way so the internal interface timing will not be a big deal. Meanwhile, checking the response of the DUV has followed similar idea: a *data valid*[1] signal has been implemented for each group of data, and the testbench will only check the response of the DUV when triggered by this data valid signal, against the expected response. Thus the testcase could focus on the expected transaction content instead of timing detail or cycle based content of the output. For the detailed implementation of the idea, please refer to *checker* module in the Verilog code.

[1] In most cases, this *data valid* signal is required by other part of the chip, so no additional effort is needed to implement this signal just for verification.

- **Golden reference model**:

  In addition to manually generated testcase, the above mentioned golden reference model in Matlab has also been used in unit-level verification. The input and output (i.e. content of each transaction) of the sub-block in the Matlab code are exported as text files, and then the files are imported into the Verilog simulation and are used as the stimuli and expected response respectively. Please refer to the module *TB_TOP* module for details.

- **Chip-level Functional Simulation**:

  This is the functional Verilog model simulation at the chip level. To be precise, it is carried out along the process of integration, or in other words, an incremental simulation procedure. Most – if not all – testcases are using golden reference model and follow the same approach as in unit-level simulation, except that now the intermediate results, which are the output of individual sub-block are also checked if necessary.

- **Post-PAR Simulation**:

  This is the Verilog simulation after place and route, using the Verilog netlist and the SDF timing annotation file generated by Xilinx PAR tool. Before it is carried out, STA is run to guarantee there is no timing violation. The purpose of this simulation is to check if there is any difference between the functional model and the physical implementation (assuming timing is OK and all the asynchronous interfaces have been appropriately designed). This simulation turns out to find a huge bug in the Xilinx tools.

After passing all the above mentioned verification, hardware testing goes smoothly, without any functional problem.

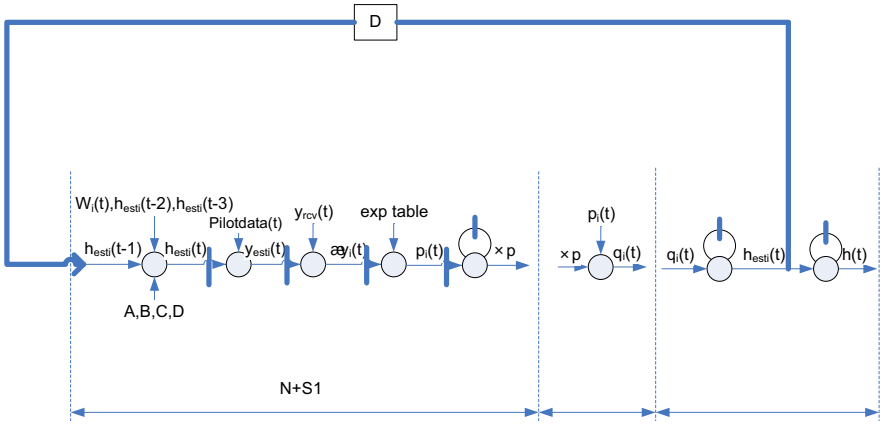

Figure 2: Architecture of MCCE calculation core

4 Architecture Specification

The implementation of MCCE consists of two major parts, i.e. the Calculation Core and the CPU Bus Interface, which will be described in the following sections respectively.

4.1 Calculation Core

As specified in early sections, several alternative architectures were analyzed and compared in the architecture exploration phase. Finally the architecture with three pipeline sections (Fig. 2) was chosen. This section will introduce the detail of this final choice.

4.1.1 Pipeline Section Description

As seen in Figure 2, the calculation core of MCCE consists of three pipeline sections (not pipeline stages!) which operate in sequence to implement the calculation of each iteration.

The functions of each pipeline section are:

Pipeline section one:

This pipeline section calculates the conditional probability of each sample. First calculates $\mathbf{H}_{predicted}$ according to equation (4), then calculate $\mathbf{Y}_{estimated}$ according to equation (5). After that, the difference between $\mathbf{Y}_{estimated}$ and $y_{received}$ is calculated and used as an index to look up in the exponential table, generating the equivalent result of the conditional probability as in equation (6). There are 500 samples, and the calculations for all samples are pipelined, so that the overall performance is optimal. At the end of the calculation, the sum of all probabilities is also generated and passed to the later section.

Pipeline section two:

This pipeline section calculates the base weight that is needed for the final calculation of the weight of each sample, based on the sum of all probabilities provided by pipeline section one. That is, instead of direct calculation of q_j following equation (7), an intermediate result $N/\sum p_i$ is generated as the base weight.

Pipeline section three:

This pipeline section determines the weight of each sample, updates the samples for next iteration based on the weights, and calculates the final estimation of the channel impulse response. For each sample, $q_j \times baseweight$ is calculated as the weight, which is the number of times the corresponding sample needs to be repeated in the next iteration as $h_{estimated}(t-1)$. So if the weight is smaller than one, then the sample is discarded, otherwise it is passed through at least one time.

4.1.2 Block Partitioning

As shown in Figure 3, MCCE consists of two major parts. The Calculation Core and the CPU Interface. The Calculation Core is further divided into the calculation pipelines (CP1, CP2, CP3), table server (TS), interface FIFO (YF, HF), global control logic (GC), and instantiated block memory, while the CPU Interface provides an interface between the OPB-BUS and internal

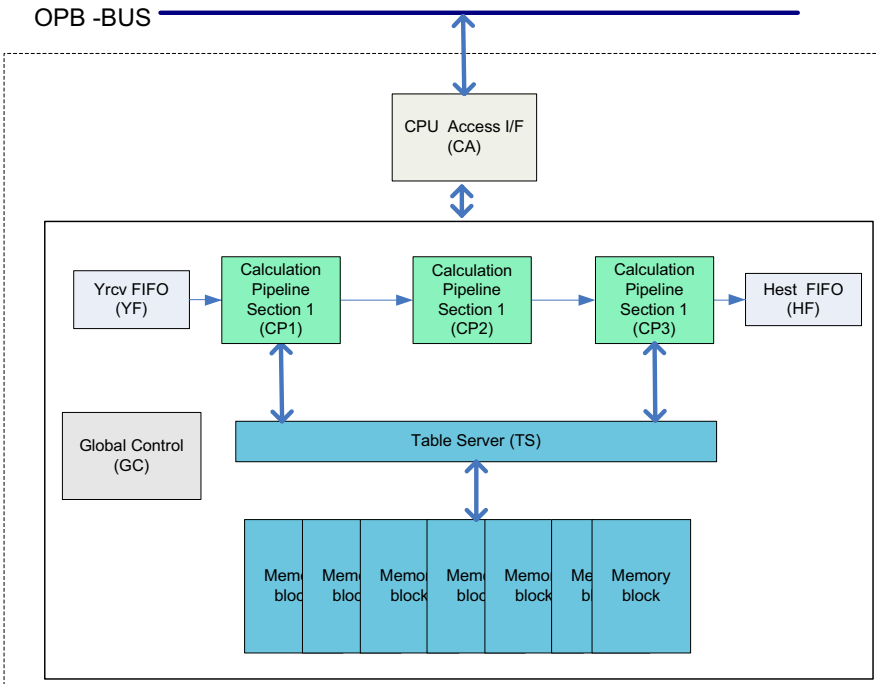

Figure 3: Partition of MCCE

accessible registers and memories.

The partitioning of the design was based on the following factors:

- Function of the Design
- Available FPGA Resources
- Human Resources

Regarding the block diagram in Figure 3, the following table gives brief information about each individual block.

The corresponding codes could be found in */pc/r/r2/vlsicourse/vlsi2004/mcce*. Please see Appendix H for hierarchy of code checked into this area.

Block name	Verilog Module	Function
GC	gc	FSM to generate global control/timing signal
Cp1	cp1	Calculates the conditional probability of each sample
Cp2	cp2, divider	Calculates the base weight for all samples
Cp3	cp3, cp3_ts	Calculates weight for each sample, and updates samples
TS	table_server	Implements all tables, coordinating all table accesses
CA	mcce, mcce_inc, core, mcce_clk, synch_reg	Implements bus interface (includes input/output FIFOs)

4.2 CPU Bus Interface

4.2.1 Specification

Since Xilinx *Virtex-II Multimedia Development Board* (with XC2V2000 FF896 FPGA) was our implementation platform, IBM OPB (On-Chip Peripheral Bus) [7] was chosen for interfacing MCCE core to the CPU. MCCE is implemented as a user core added to the embedded system centering on a MicroBlaze soft-processor.

MCCE is an OPB slave device with a 32-bit interface to the main bus and uses only byte-enable architecture of OPB v2.0 specification as recommended by [8], so no conversion cycle or data mirroring is required from the master side. Since MCCE is a 16-bit device from the size of data bus transaction point of view (i.e. all registers and memory-mapped accesses to the tables are 16-bit aligned), depending on the addressed data the MCCE 16-bit data is routed to/from either upper or lower half of the OPB main bus as is shown in Fig. 4. The data access is initiated through Microblaze half-world instructions.

To simplify the process of attaching a user core to a CoreConnect bus, which super-sets the OPB interface, Xilinx provides a portable pre-designed

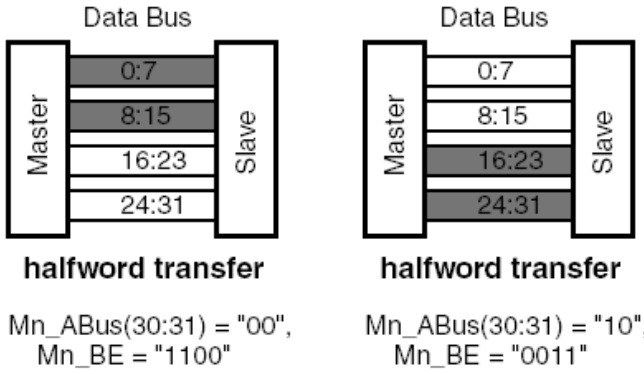

Figure 4: Byte lane usage for MCCE transfers (figure from [8])

bus interface (called the IP interface, IPIF) that takes care of the bus interface signals, bus protocol, and other interface issues by presenting a simpler interface called IPIC (IP InterConnect) to the user core [9]. Although use of IPIC makes a design portable, we chose not to use it to experience all low-level signal interface and timing issues instead. Further, our byte-enabled slave device didn't have the complexities of a full-scale OPB v 2.0 master device (which needs to support both legacy and byte-enabled access, and also handle bus arbitration, possible data mirroring, ...).

Fig. 5 shows the interconnection of the register data path from the bus interface to the calculation core.

4.2.2 Design Issues

Now we briefly discuss some topics related to design of CPU interface.

Multiple clock domains:

Since the calculation core is running at a different clock from the CPU interface block, the registers accessed by both CPU interface and calculation cores need to be properly synchronized. So a handshaking mechanism to pass data between clock domains based on [10] was implemented.

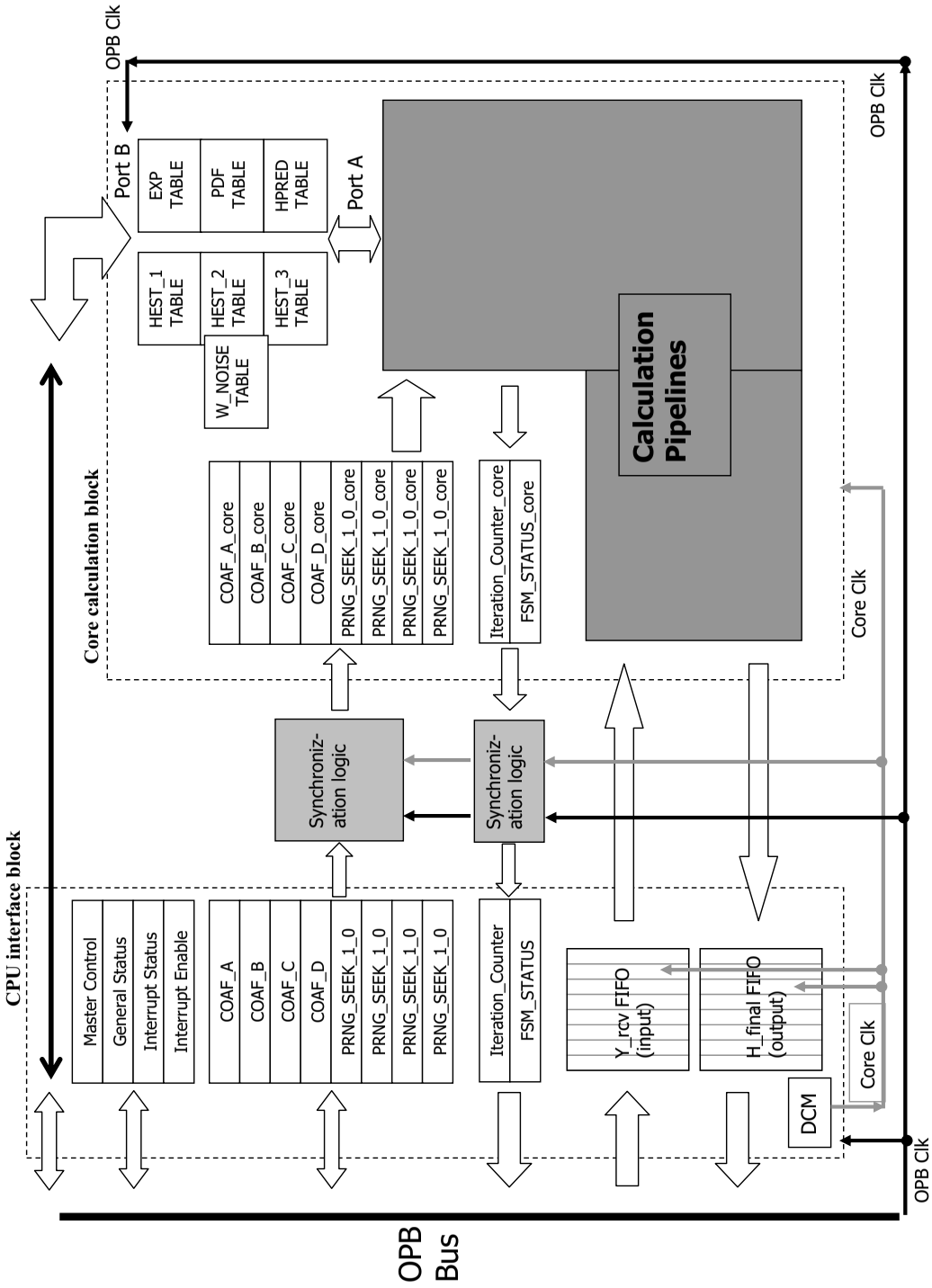

Figure 5: MCCE CPU/Core data path

Some registers are read only from calculation core point of view (e.g. $COAF\_A$) while the others are write only (e.g. FSM_STATE). So each side, bus interface and calculation core, keeps its own copy of such registers clocked by OPB clock and core clock, respectively. Extra handshake signals manage the synchronization protocol for each register where every synchronization signal itself is synchronized through a two-flip-flop synchronizer. Data input and output streams to MCCE are managed through asynchronous FIFOs which is implemented from distributed memory to save on-chip Block RAM. Since the content of internal tables are exposed to the CPU for debugging purposes, all tables are instantiated as dual port memory so each clock domain has its own exclusive port to the memory.

Clocking:

The clock for calculation core is built from the OPB clock using a Digital Clock Manager (DCM). The core block is currently set to five times of the OPB clock (i.e. 5 * 27 = 135 MHz) considering current bottlenecks. With further improvement and deeper pipelining, it's expected to be able to increase the core clock close to the maximum speed of hard multipliers ($\sim$180 MHz on speed grade of 4).

The clock provided to the calculation core is the frequency synthesized output of DCM passed through a $BUFGCE$ buffer to be able to gate the clock. This allows the calculation core to be put in low-power mode through a programmable bit of $MASTER\_CONTROL$ register. The $LOCKED$ output of DCM is used to prevent handling of CPU requests till DCM clock goes to stable state.

Reset:

All registers and pipeline stages of calculation core use asynchronous reset. The CPU interface block provides reset for calculation core block based on OPB_RESET or soft-reset. This reset is synchronized with the clock to make sure enough time is given for register state change before arrival of the first clock after the reset.

Verification using OPB Bus Toolkit:

The OPB Bus Functional Toolkit developed by IBM provides unit/system level simulation of OPB bus activity to accelerate the design cycle time by identifying problems at earlier stage [11]. This toolkit really helped us to speed up the test of CPU interface and its interaction with calculation core (instead of developing our own bus activity signaling through Verilog or TestBuilder).

Our Bus Functional Language(BFL) test scripts covered read/write access to most registers, some table locations, interrupt generation and acknowledge, and also data accumulation in input FIFO and data consumption from output FIFO.

CPU access wait cycles:

Originally no wait cycle inserted as the bus clock speed used in the system was low ($27MHz$). The falling edge of OPB clock was used to trigger read and write so the register access cycle ended by the rising edge of the next clock (based on [7]). When we started to verify the post-synthesis model of our MCCE core using OPB Bus Toolkit, we figured out that the toolkit failed all CPU accesses because of bad timing. This was originated from the fact that all bus signals (input and outputs) were routed to the pads because of simulating the IP core alone not the whole system. This was causing extra 4.5 ns routing delay from input pads (e.g. address signals) to the core and similar delay from the core to output pads (e.g. data bus) shifting the timing around 9 ns.

Although this wasn't a real issue for the final embedded system, we inserted a one cycle delay to resolve this issue and used this one cycle wait even for the final implementation rather than using the delayed mode only for post-synthesis simulation and original no-wait cycle for final implementation.

Debugging features:

We originally planned for a single-step like debugging capability of calculation core. This was planned to be done through enabling the debug mode

first, e.g. a bit of master control register, and then further writes to another bit of master control register advancing the calculation core's state machine one cycle for every trigger. FSM_STATUS and $ITERATION\_COUNTER$ registers would expose enough information to learn about the exact state of calculation core's FSM. Then content of intermediate tables could be investigated to find possible bugs.

The natural way to implement this mode was to gate enable signal of every register and pipeline stage with the trigger signal if debug mode is active. Another method we explored a bit was to completely gate the clock of calculation core when single-step was active and then let through a single pulse whenever the single-step trigger was activated. Though this method was simpler to implement (without introducing extra gate/delay in pipeline stage), we had to drop it as synchronization logic to pass register data between clock domains depend on an active calculation core clock to handle the handshake protocol.

We didn't have to use this debugging mode in practice as the only problem we found when moving to the real hardware was resolved through post-synthesis simulation (refer to section 7 for CoreGenerator issue with multipliers when inputs are registered).

Interrupt:

To reduce the software programming overhead of data streaming to MCCE and more efficient use of CPU resources instead of polling, MCCE was made capable of generating interrupts based on different FIFO conditions (e.g. input FIFO empty/almost empty to initiate refill of input FIFO or output FIFO full/almost full to initiate consumption of produced output data). The interrupt signal is level sensitive and is managed through interrupt enable and status (also handles acknowledgement) registers. For a polling scenario, the status provided through $GENERAL\_STATUS$ register can be used instead.

5 Design Characteristics

5.1 Design Environment

Due to the available resource and the maturity of EDK, the project was carried out mostly in PC, and Xilinx ISE version 6.2 was used as the major environment. The project navigator was used as the GUI to manage the project. Because it could show the layered module instantiation relationship, manage multiple top modules and hide the physical location information of the files, no complicated and explicit source file directory structure was used. At the same time, no version control tool was available for the PC environment, and the version control was achieved by introducing version information into the naming of directories. Fortunately, this has worked well up to now.

For scripting, most of the operations were under GUI mode, although for the simulation, macro command files were used as the script to manage the testbench and the testcase. Because of the complexity and the extent of the design, the GUI could provide a quicker and convenient way of working around.

The tools used in this project are:

- Design entry: Xilinx ISE built-in editor
- Simulation: Modelsim 5.7g Starter
- Synthesis: XST (ISE built-in synthesizer)
- PAR: ISE 6.2i
- CPU core design environment: EDK 6.2

If we implemented the project in Unix environment, explicit file directory structure would be used including the source code, simulation and synthesis directories. The source code directory would be further divided into sub sections according to the partitioning of the design. At the same time, version control would also be incorporated. This strategy should have been

constructed in the Unix system and mounted into the PC environment as virtual disks.

5.2 Chip Characteristics

Please see Appendixes E and F for the PAR reports.

Since we picked architecture 3 (see Appendix C), each iteration takes $2N + S1 + S2 + S3 = 2N + S_{pipeline\_overhead}$ (N is number of samples and $S1$, $S2$ and $S3$ are pipeline overhead/latency for each section). The total overhead is 39 cycles so each iteration takes 1039 cycles (for $N = 500$). Since calculation core is running at 135 MHz, ideally *129.93 K Iterations/seconds* can be achieved. Software overhead shouldn't prevent us from reaching this rate as input/output FIFO mechanism and interrupt-based notification to CPU should minimize/eliminate software overhead.

This high iteration rate allows employing this project for applications with higher estimation rate requirements.

5.3 Computing System

Our main development and deployment environment was a P4 2.4 GHz system with 512 MB RAM running WindowsXP. This system was good enough for almost all stages of our design. Sometimes we used Solaris servers for lengthy MATLAB and ModelSim simulations (e.g. for 1500 iterations case) not necessarily because of speed but because it let our PC free for other tasks. Our PC version of ModelSim was *5.7g Starter* edition, so it was running pretty slow making it impossible to simulate large number of iterations.
Below are some data regarding our computing resource usage:

- MATLAB bit-true simulation takes:
  - 6' 35" for 47 iterations on PC.
  - 9' 45" for 47 iterations on Solaris; around 5 hours for 1500 iterations.

- Functional simulation with ModelSim (with memory footprint of around 16 MB in both cases):
  - 20 iterations/hour on *5.7g Starter* PC version; taking 2 hr 24' for 47 iterations.
  - 1081 iterations/hour on *5.7f* Solaris version; taking 1 hr 25' for 1500 iterations.
- Post-PAR simulation with ModelSim on PC takes 44 minutes for 3 iterations (nearly 5 times slower than functional simulation).
- MCCE core alone takes
  - 1' 15" to synthesize.
  - 9' 30" to Place and Route.
- Whole embedded system takes around 12.5 minutes to synthesize, place and route, and finally compile the system software.
- Peak memory usage in different stages of system compilation:
  - platgen: 222 MB
  - XST: 95 MB
  - map: 111 MB
  - PAR: 205 MB
- Disk usage
  - Whole embedded system EDK project: 101 MB
  - Demo PC software: 29 MB

5.4 Where the Time Went

Fig. 6 shows the summary of the time consumed for the project, where:

- Week No. 1 starts from Feb 9, 2004, the week when the team decided to implement the present project.

Week No.	Understanding the Problem	Algorithmic Simulation	Learning Time	Tool Debugging	Coding	RTL Simulation	Gate-Level Simulation	Synthesis & PAR	Writing Document	**Total**
1	30	30	30							90
2	40	30	20							90
3	31	40								71
4	23	50								73
5	10		28						20	58
6		45							30	75
7			20		40	45				105
8			12	20	30	40		10	10	122
9			55		10	45		20		130
10		20		30	10	30				90
11			30	20	40	50		10		150
12			20	20	30	60		20		150
13			30		30	30	40	30	20	180
14		10	10		20	30	10	5	30	115
15			10		60	10	10	5	20	115
16					60				50	110
Total	134	225	265	90	330	340	60	100	180	1724
Percentag	8%	13%	15%	5%	19%	21%	3%	6%	10%	1

Figure 6: Time Consumption Summary (in hours)

- "Learning Time" refers to "explicit learning time". Actually learning exists in every activity.
- "Coding" refers to RTL coding , C coding (both system software and demo PC software) and TestBuilder coding, but not Matlab coding which is already included in "Algorithmic Simulation".
- "Synthesis & PAR" includes the time to "make the design run faster".
- All numbers are approximate, mostly derived from the weekly report.

It is interesting to learn from the figure that:

- *RTL simulation* consumes almost a quarter of the total time, which is quite reasonable. If only considering the "implementation phase", and if we did not need to spend so much time learning new stuff, the percentage would be much higher.
- *Algorithm simulation* holds around 13 % of all the time, which is a little bit lower than our expectation. We worked on some simulated Matlab codes and modified them to fit our need; this may have saved

us some time. It is also interesting to notice that during the later phase of the project, we still need to go back to the algorithm model, in order to have a better reference model, or to get rid of the discrepancy.

- *Gate-level Simulation*(post-PAR simulation) takes about 3%. If not for the bugs of Xilinx ipCore tools, the figure would be even less. For a FPGA design, the physical level design activities – synthesis, PAR, STA, Post-Sim – should only be a small porion of the project.

6 Specification of the DEMO System

6.1 Hardware

The components comprising the final system are shown in Figure 6. Since enough on-chip Block RAM was available, external memory was not used. Out of 56 available blocks of 18 KBits BRAMs, 7 blocks were used by MCCE core to hold different tables. Note that only 500 words out of 1024 available words of each table is currently used and the rest is kept for further extension.

To have more flexibility with interrupt properties (e.g. edge/level and polarity), MCCE's interrupt line was connected to interrupt controller rather than direct connection to MicroBlaze. A UART interface was used as input/output interface from PC to the embedded system.

6.2 System Software:

In the first revision of the software, PC was launching only a terminal session to serve as user input/output where all input command parsing and output generation (e.g. terminal echo back and command results) were generated by MicroBlaze code. When using standard library for input command parsing (e.g. *scanf* and string functions), we ran out of on-chip memory. Since there was no reduced form of *scanf* function (like *xil_print* for *printf*) with smaller library footprint, we developed our own string and *scanf* functions. The type of commands defined with this revision were register read and write functions with different widths (full-word, half-word and byte) and a

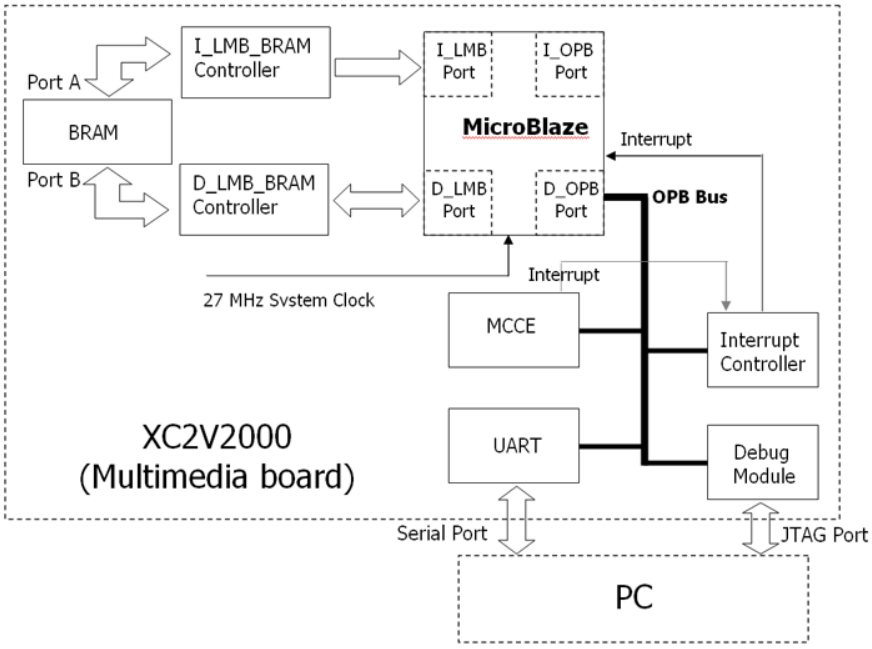

Figure 7: Embedded system components

testbench command (see section 7 for why we couldn't use XMD register access commands). This testbench command was feeding the input FIFO of MCCE with one data, receiving the corresponding output data and comparing it with the expected output and repeating this for the next data, very similar to our Verilog testbench without employing interrupt.

Since all input and expected output data were defined as online arrays rather than through file access, test of larger dataset was not feasible through this method. Also, we had to re-initialize the BRAM after first trial of the testbench since we were not capable of re-initializing the content of our tables through reading the table initialization files.

In the next phase of the software intended for final demo, we delegated command parsing, file access and user input/output to the PC software where MiroBlaze software was processing stream of simple command packets received over serial interface from the PC software. These command packets were either single register read/write access (fixed size packet), mul-

tiple table memory read/write access (variable length packet) or FIFO input/output access (variable length packet).

Two modes implemented for FIFO input/output access. One in a synchronous fashion without employing interrupt using polling to check MCCE's output FIFO for available data and the other in asynchronous fashion relying on MCCE's interrupt to initiate receive of output FIFO data. In either way, MicroBlaze uses some large input and output data streaming buffer (each 1KByte) implemented as a C++ object in a circular buffer fashion to:

- accumulate input data received from PC software.
- consume from it to fill MCCE's input FIFO.
- wait till MCCE's output FIFO has ready data for picking-up through polling or interrupt.
- consume from MCCE's output FIFO and store it in MiroBlaze output streaming buffer.
- generate a response packet for PC software when number of available data in output streaming buffer has passed a threshold. The threshold is a command parameter passed to system software through PC software.

Since streaming input/output circular buffers are accessed by both main thread and interrupt handler, proper synchronization at critical sections implemented through disabling interrupts.

The general form of command processing by system software is as below chart:

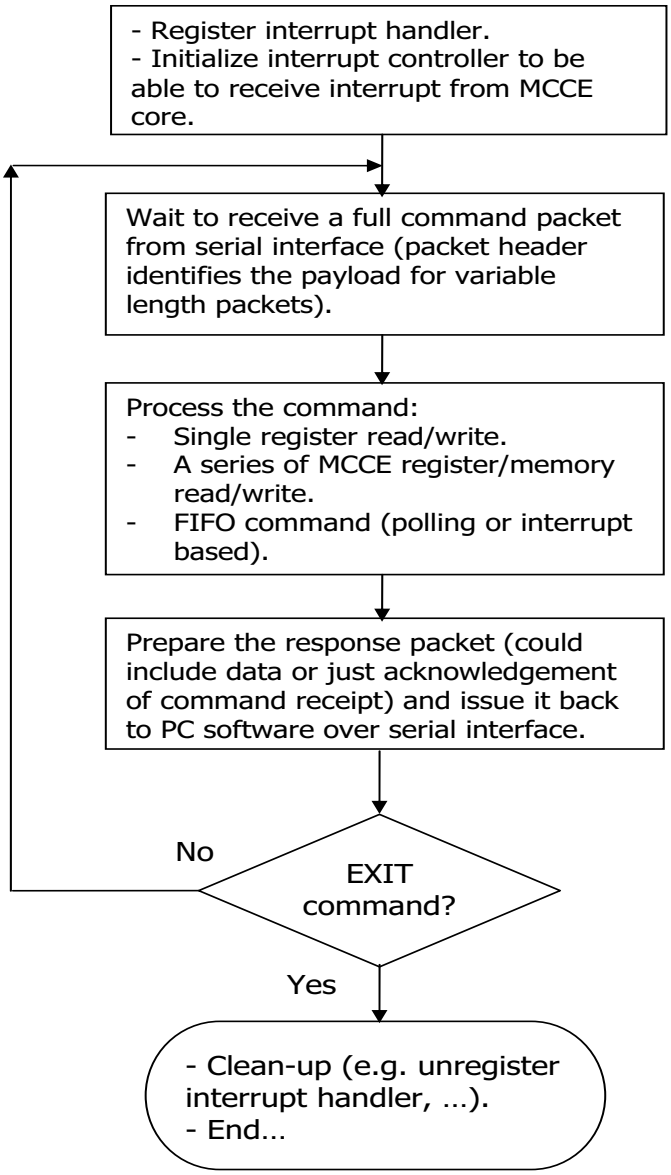

Figure 8: System software flow

6.3 PC Software

PC software is mainly responsible for:

- Command parsing
- Command packet creation
- Communication to embedded system over serial interface
- File access to load input data to be provided to MCCE and its corresponding expected data (to be used by testbench commands for success/failure determination)
- Data representation in a graph form
- Initialization of MCCE tables (with proper random values read from memory initialization files)

Beside the low-level commands discussed already in the previous section, it is also capable of running script files of known commands. It is developed using Microsoft Visual C++ .NET to have full control over serial access. A mix of MATLAB and C code was an option especially for live simulation of algorithm in MATLAB and using a C code interface to handle serial communication. However, it was ruled out because of low speed of MATLAB simulation. The best option was to port the algorithm completely to C for live calculation which wasn't feasible in the limited time available for demo preparation. So we had to use offline data already prepared from MATLAB simulation.

For serial communication, *MSCOMM32* (a Microsoft ActiveX control encapsulating serial interface as an object) was used in the beginning. This made serial access very easy but had the drawback that if the embedded system was sending binary messages (instead of text messages) through the serial interface, *MSCOMM32* methods were not able to retrieve the message properly as all input/output parameters expected to be null-terminated strings rather than binary data. So we had to use lower-level APIs which

treats serial interface like a file access but with extra APIs to set/retrieve serial interface properties [12].

For graph representation of the received data, Microsoft Chart Control 6.0 was used. Though this ActiveX control is not intended for dynamic update (causing flashes with each redrawing), it speeded-up our demo development.

Fig. 9 shows a snapshot of the PC software.

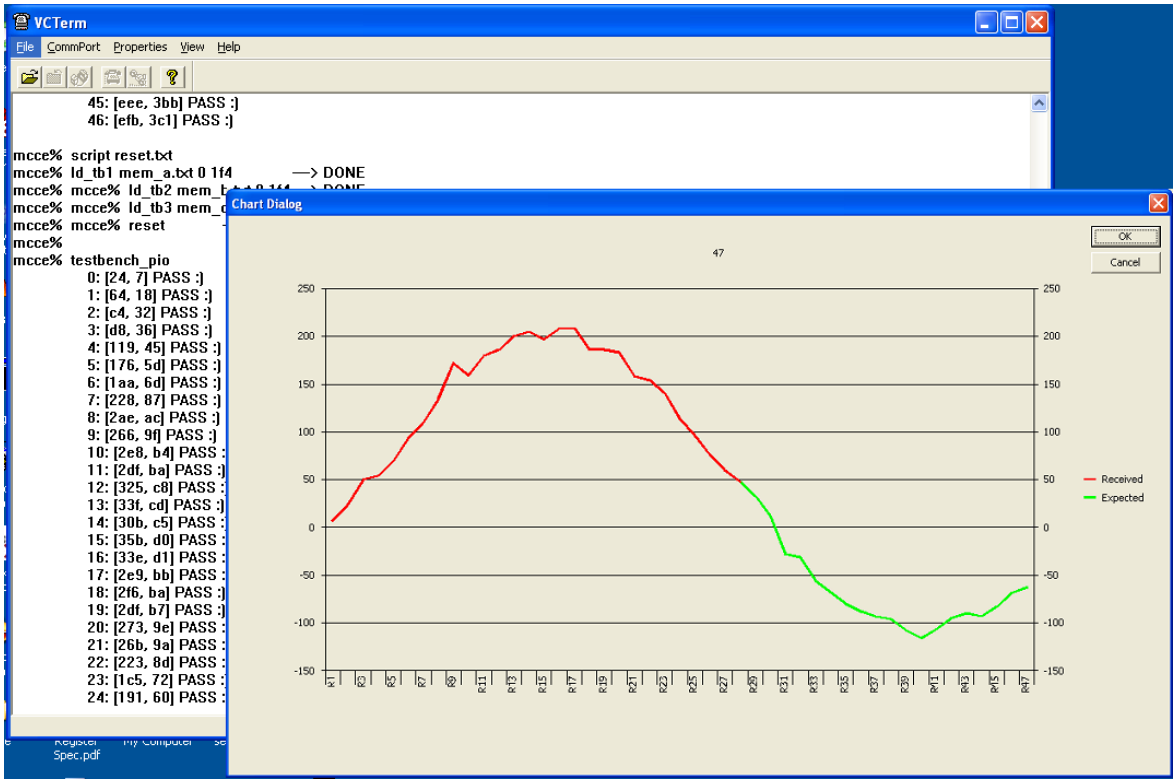

Figure 9: Demo snapshot

7 Problems

Signed-Number Multiplier Inference:

Verilog language construct does not support signed number multiplier

directly, so if we want to infer a signed number multiplier, the behavior of it must be described based on un-signed number multiplication. The synthesis result, however, could not figure out it is a signed-number multiplier, so that it could not use the multiplier block in Virtex-II as signed-number multiplier. Instead, unsigned-number multipliers are inferred with addition logic to implement the target, and this introduce a very long timing path in the design.

Solution: Instantiate the signed-number multiplier generated by Core-generator.

Probe the internal signal for post-PAR simulation:

In order to compare the internal signals of the functional model and the post-PAR netlist, we need a quick way to probe the internal signal and map the signal to top layer of the design as output port. Verilog language provides the hierarchy name as a quick way that is supposed to support this kind of needs. However it turns out that XST does not support this kind of language construct, although the simulator (Modelsim) supports it very well.

Solution: Implement the probing signals as ports in each layer and connect them bottom-up.

Difference between Golden model and Verilog Model:

As mentioned before, the initially developed golden model did not agree with the Verilog Model. There were two reasons for the problem: one was the coding error in either model; the other was the granularity of the Matlab golden model.

Solution: Correct the coding errors; lower the granularity of the Matlab golden model.

Difference between functional model and post-PAR netlist:

Due to the bug of the CoreGenerator of ISE, when the multiplier is configured such that the input signal is registered, the functional model does not agree with the post-PAR netlist and it turns out that the timing

of the functional model is incorrect.

Solution: Use the un-registered input in the multiplier.

XST fatal error when synthesizing the system:

XST was generating a fatal error synthesizing our embedded system. This turned out to be a known EDK/ISE 6.1 issue with mixed language cores (Answer Record #17491).

Solution: Upgrading to EDK/ISE 6.2 solved the problem.

ModelSim post-PAR simulation failure in Solaris:

Post-PAR simulation runs much slower than functional simulation so we tried to use the full version of ModelSim available on Solaris. We were not able to run post-PAR simulation on Solaris receiving "Error: (vsim-SDF-3262) mcce_timesim.sdf(36): Failed to find matching specify timing constraint." error on every "(PERIOD (posedge CLK) (1530:1530:1530))" line in generated Standard Delay Format (SDF) model file. We couldn't figure out how to resolve this.

Solution: We had to do post-PAR simulation only on PC.

XMD bug with half-word data access:

Xilinx Microprocessor Debugger (XMD) always issue a full-word memory access in response to *mrd* and *mwr* commands no matter what type is requested. This causes request failure with our MCCE core as all register and memory accesses must be half-word requests. Further, even if the core accepted full-world requests, *mrd* command would return wrong mix of bytes for half-world accesses. We had to use *ChipScope Pro 6.2i* to debug bus transaction and figure out it wasn't a core issue.

Solution: Issuing MicroBlaze half-word instructions directly works fine.

XST failure to load include file:

When synthesizing the system, XST was failing because of not being able to find a file included in a Verilog source file (*'include "mcce_inc.v"* used in *mcce.v*) though both files were in the same path. This was caused

because EDK make file runs from a different path so when XST was called by EDK, the current path was not the HDL source path. Probably this can be solved if core's *Peripheral Analyze Order (.pao)* accepts a new directive about include file search path.

Solution: Specifying the included file with full path solved the problem. The drawback was that ModelSim didn't like the full path, so had to use both forms using conditional compilation (see *mcce.v*).

Issues with CPP source files:

Apparently either there is no C start-up code (in the C runtime library) to call constructor of objects defined in global scope or it doesn't work properly. We had to initialize the object by explicit function call (i.e. calling an *Init* function rather relying on automatic constructor call). Also, EDK documentations don't mention the required change in linker script to be able to compile CPP files (e.g. new linker section necessary for exception handling frame, ...).

8 Conclusions, Suggestions, Comments

In this project, the Sequential Monte Carlo channel estimation algorithm has been implemented in FPGA, involving the design and verification activities in algorithm level, macro algorithm level, logic level and physical level. During the process, many of the topics covered in the course have been practiced and different tools have been used for relevant tasks. This was a great experience for every member of the team.

During the project period, many problems were encountered. Some were related to the understanding of the problem and finding a proper application for the target system while others were regarding to the methodology and tools, and the rest were management and communication issues. Although most of the problems were addressed by satisfactory solutions, in retrospect, there are still lessons to learn and a few suggestions for future projects.

For instance, in our case, many working hours devoted to the understanding and refinement of the algorithm. Due to the nature of this course,

the algorithm itself should not be the major focus. In fact, even after so much effort put into the algorithm, our interpretation and application of the algorithm is still away from engineering reality. So one suggestion is to provide a list of well-defined alternative projects, or to give a strict deadline on presenting the function specification of the project and the spec should be taken "as is" for future development, no matter how incomplete it may look like. [2]

Another lesson for MCCE is the design environment. Although due to the complexity of the design and the size of the team, our somehow ad-hoc design environment on WindowsXP did not bring much trouble, we missed a good chance of experiencing and learning well-organized design environment (e.g. running synthesis, simulation and PAR through command line with proper environment setup). So the suggestion is to "force" certain design environment from the very beginning.

9 Contributions

Mahdi:

- *Algorithm level*: Developing
  - Ideal functional model
  - Bit-true Simulation
  - Executable model simulating hardware (Golden reference for future use)
- *Logic level*:
  - Design, implementation and verification of first pipeline (cp1) and table server; update of cp3
  - Integration of all pipeline stages resulting in the core

[2] In real engineering project, the iteration of function spec – in fact, every piece of work – is almost unavoidable, and we need to live with that. For course project, to make life easier, and to focus on the theme of the course, a ridiculous function spec is better than a non-frozen spec.

  - Integration of the core and Bus-interface
  - Verification of the whole core
  - Verification of the integrated system (BUS & Core)

Hassan:

- MATLAB data type for bit-true simulation (later dropped for simpler approach)
- Initial design and implementation of first pipe (cp1) and table server.
- Design, implementation and verification (with OPB bus toolkit) of bus interface unit
- Integration of bus interface and calculation core.
- Performance improvement of calculation core (increasing calculation core clock from 3x of OPB clock to 5x)
- Embedded system design and bring-up (including MicroBlaze software)
- Development of system and PC software for demo

Jing Zhang:

- *Algorithm level*: strategy and implementation of bit-true simulation
- *Architecture level*: alternative analysis, re-sampling implementation, documentation of architecture spec and register spec
- *Logic level*: sub-block cp2, cp3, gc coding, verification and synthesis; Verification strategy design, utility template development
- *Physical level*: sub-block timing optimization

References

[1] S. K. K. Fazel, *Multi-Carrier and Spread Spectrum Systems.* John Wiley, 2003.

[2] Y. Zheng, "A novel channel estimation and tracking method for wireless ofdm systems based on pilots and kalman filtering," *Transactions on Consumer Electronics*, vol. 49, no. 2, May 2003.

[3] S. P. M.J. Omidi, P.G. Gulak, "Parallel structures for joint channel estimation and data detection over fading channels," *Journal on Selected Areas in Communications*, vol. 16, no. 9, December 1998.

[4] M. C. H. M. H. Keding, M. Willems, "Fridge: A fixed-point design and simulation environment," *Proceedings, Design, Automation and Test in Europe*, pp. 429–435, 23-26 February 1998.

[5] K. S. Kim and W. Sung, "Fixed-point optimization utility for c and c++ based digital signal processing programs," *Transactions on Circuits and Systems-II: Analog and Digital Signal Processing*, vol. 45, no. 11, November 1998.

[6] *TestBuilder User Guide*, Product version 1.3-s8 ed., Cadence Design Systems, Inc., August 2003.

[7] *On-Chip Peripheral Bus Architecture Specification*, version 2.1 ed., IBM Corp., August 2003.

[8] *Processor IP User Guide*, EDK 6.2 documentation, Xilinx, Inc., 2004.

[9] *User Core Templates Reference Guide*, EDK 6.2 documentation, Xilinx, Inc., 2004.

[10] C. E. Cummings, "Synthesis and scripting techniques for designing multi-asynchronous clock designs," in *SNUG-2001*, San Jose, 2001.

[11] *OPB Bus Functional Model Toolkit User's Manual*, version 3.5 ed., IBM Corp., June 2003.

[12] *Communications Functions in Platform SDK*, version 3.5 ed., Microsoft Developer Network documentation, Microsoft Corp., July 2002.

A Functional Specifications

A.1 Performance Target

Basically there are two performance targets for the FPGA implementation:

1. Estimation frequency: 1K estimations/sec
2. Sample space N: 500

These targets are the tentative targets for the FPGA implementation. Final implementation result might be (generally) better than the above figures.

A.2 Finite-Word-length For FPGA Implementation

After analysis and Bit-true simulation using Matlab, we have finalized the word-length for all the coefficients and variables used in the calculation.

Object Name	Meaning	Total Length	Integer Length
$h(t)$	Channel impulse response	10	6
$y(t)$	Received signal	10	6
A,B,C,D	Channel coefficient	10	3
Mul Operation	Multiplication used to calculate $h_{estimated}(t)$	16	10
Add Operation	Addition used to calculate $h_{estimated}(t)$	16	10
$h_{estimated}(t)$	Estimation of channel impulse response	10	6
W	Process noise	10	6
$y_{estimated}(t)$	Estimated receive signal	10	6
$y_{received}(t)$-$y_{estimated}(t)$	Difference between received and estimated signal	10	6
p	Posteriori distribution of estimation vector	4	2
sum p	Sum of p for one time step	16	10
q	Importance weight	10	1

B Verification Strategy Summary

Verification is the most time-consuming part of MCCE and it directly determines the quality of the project. In the following sections, the general strategy will be presented, followed by some testcase example.

Usage of golden reference model:

As described before, the Matlab code had been originally implemented for algorithm evaluation and finite-word-length effect evaluation. But later it was modified to be the behavior model of the design and used as the golden reference for HDL simulation.

Incremental verification:

For HDL verification, unit simulation is carried out at a reasonable granularity first; then adjacent units are integrated and simulated gradually till the fully functioning chip.

Automatic checking:

As described in the report, some utilities were developed to fulfill automatic checking so that manual checking was kept to minimum.

Transactional level verification:

The signal level behavior of the testbench is encoded into some utilities and the testcase will focus on the content of the stimuli and responses. This also helps using the text file generated by the Matlab code as stimuli and responses, i.e. the mechanism of using golden reference model.

Exhaustive simulation:

In the demo, some 1500 iterations were used to demonstrate the correctness of the implementation.

Some testcases examples are as follows.

Feature under simulation	Stimulus	Expected response
Manipulate mix of positive and negative samples	Initialize the memory and part of the samples are positive while others are negative	Updated estimation and final estimation as in golden model.
Manipulate overflow and "divide by zero" condition	Initialize the memory such that the sum of probabilities would case overall, or be zero	Clap to the maximum possible value, as in golden model.
Operate on FIFO correctly	Force the FIFO to be either full or empty	Pause operation after the present iteration; while the FIFO is OK, resume operation.
Operate on consecutive negative or positive receive value for long time (> 50 iterations)	In Matlab code, manually construct the channel impulse response to satisfy the condition, and then generate the stimuli and responses text file	Behavior as golden model.

Table 1: Sample test cases

C Architecture Specifications

Introduction

The MCCE (Monte Carlo based Channel Estimation) is the FPGA implementation of the algorithm as presented in the function spec [1]. The core function of the implementation could be expressed as in Figure 1.

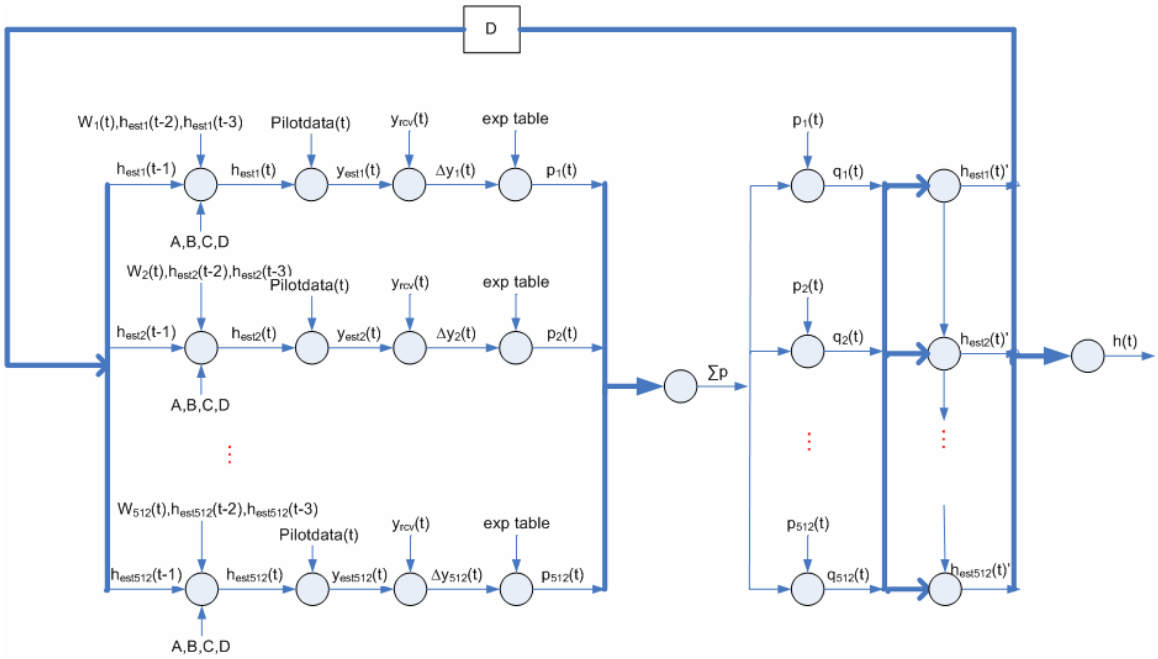

Figure1. Dependence Graph

Architecture Exploration

In order to find a good architecture, we have studied the intrinsic parallelism of the algorithm, and identified the following Key factors affecting the exploration of parallelism:

- Data dependence
- I/O constraint (Implementation of the table)

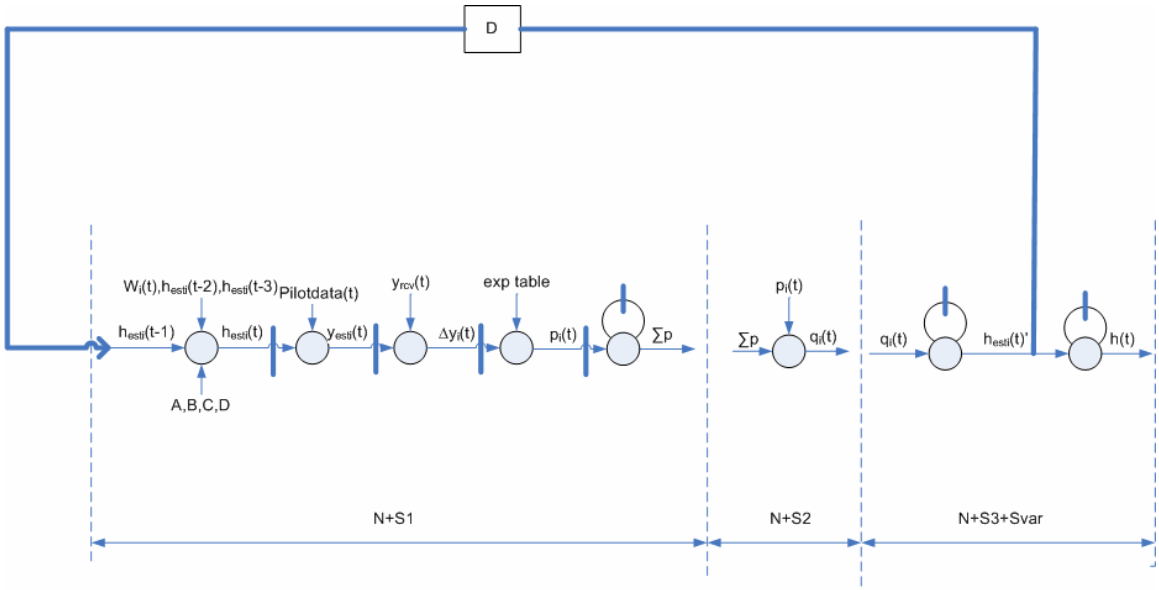

Figure 2. Architecture 1: Three cascaded pipelines

Based on the study, we have considered the following four possible architectures.

Architecture 1:

As in Fig. 2, the architecture consists of three cascaded pipeline sections. The relationship between pipeline section 2 and 3 is "producer-consumer", and we can use a table to glue 2 and 3 (i.e., start 3 only after 2 is finished).

The overall iteration period (the time required to generate an effective $h(t)$) is

$$\left(N + S_1 + N + S_2 + N + S_3 + S_{\text{var}}\right) * T_{ps} = \left(3N + S_1 + S_2 + S_3 + S_{\text{var}}\right) * T_{ps}$$

where N is the sample space (i.e. 512 in our case), S_1, S_2, S_3 are the pipeline stage numbers for the three pipeline sections respectively, S_{var} is the variable processing time of pipeline section 3, and T_{ps} is the calculation time (e.g. clock numbers) for each pipeline stage.

The advantage and disadvantage of this architecture are:

Pros: simple.

Cons:

- Calculation times for pipeline section 1 and 2 are fixed, but the calculation time for pipeline section 3 is variable.
- Have not made the best possible use of intrinsic parallelism. i.e. due to the usage of table between pipeline section 2 and 3, the start of the calculation in pipeline section 3 has been delayed till all entries of the table are available.

In order to make the best possible use of the intrinsic parallelism, we come up with architecture 2.

Architecture 2:

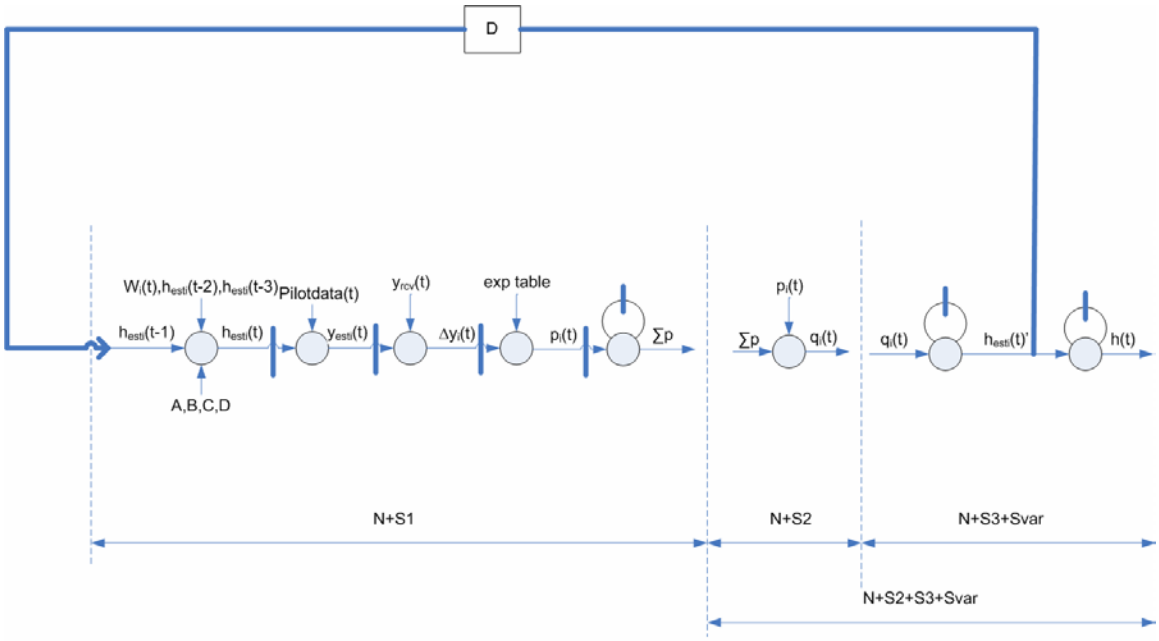

Figure 3. Architecture 2: Three pipeline sections, with section 2 and 3 partially overlapped using elastic buffer

In this architecture, due to the contribution of the elastic buffer, once section 3 has enough information, it can start to work

The overall iteration period is

$$\left(N + S_1 + N + S_2 + S_3 + S_{\text{var}}\right) * T_{ps} = \left(2N + S_1 + S_2 + S_3 + S_{\text{var}}\right) * T_{ps}$$

The advantage and disadvantage of this architecture are:

Pros: simple, and quicker than architecture 2.

Cons: Calculation times for pipeline section 1 and 2 are fixed, but the calculation time for 3 is variable.

At this time, we have considered the following question: Why not try to make the calculation time for section 3 fixed? Then we come up with architecture 3.

Architecture 3:

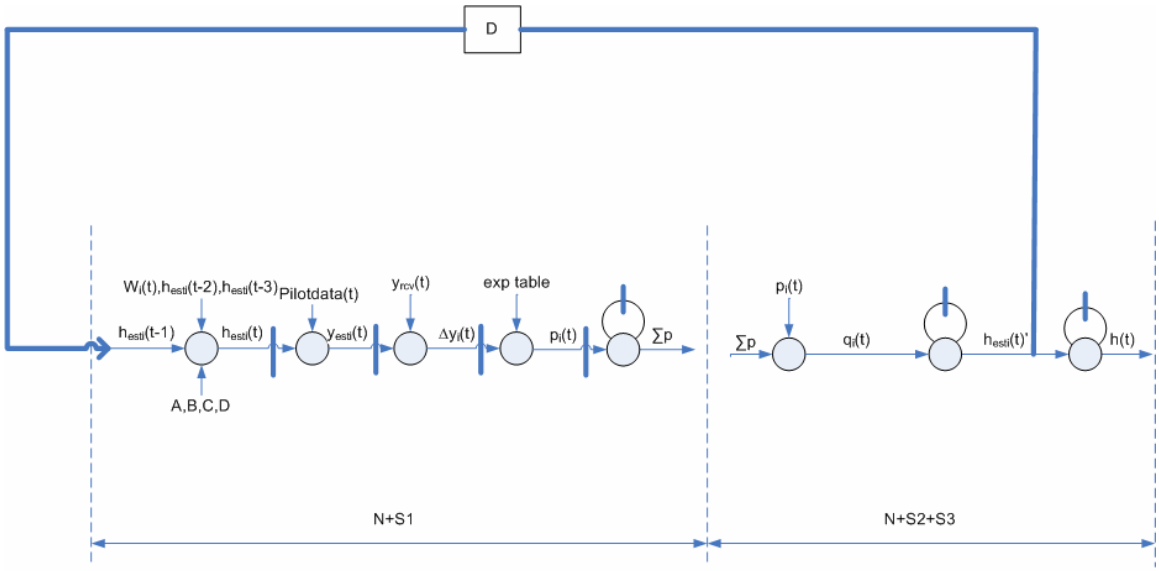

Figure 4. Architecture 3: Improved data structure for storing h(t)

The reason that the calculation time for section 3 is variable is that we need to write the records one by one. We can improve the data structure of $h(t)$ such that for multiple entries with the same value of $h(t)$, only one write to the memory is needed. This is achieved by adding one bit to each record of $h(t)$: New_record. When it is "1", that record is a new record and will be used, otherwise that record will be ignored and the last valid record will be used instead. This architecture is actually a cascade of two pipeline sections.

The overall iteration period is

$$(N + S_1 + N + S_2 + S_3) * T_{ps} = (2N + S_1 + S_2 + S_3) * T_{ps}$$

The advantage and disadvantage of this architecture are:

Pros: simple, fixed calculation time.

Cons: because one pipeline is used per section, high estimation frequency requirement may not be reached.

In order to solve the possible performance limitation, we have considered architecture 4.

Architecture 4:

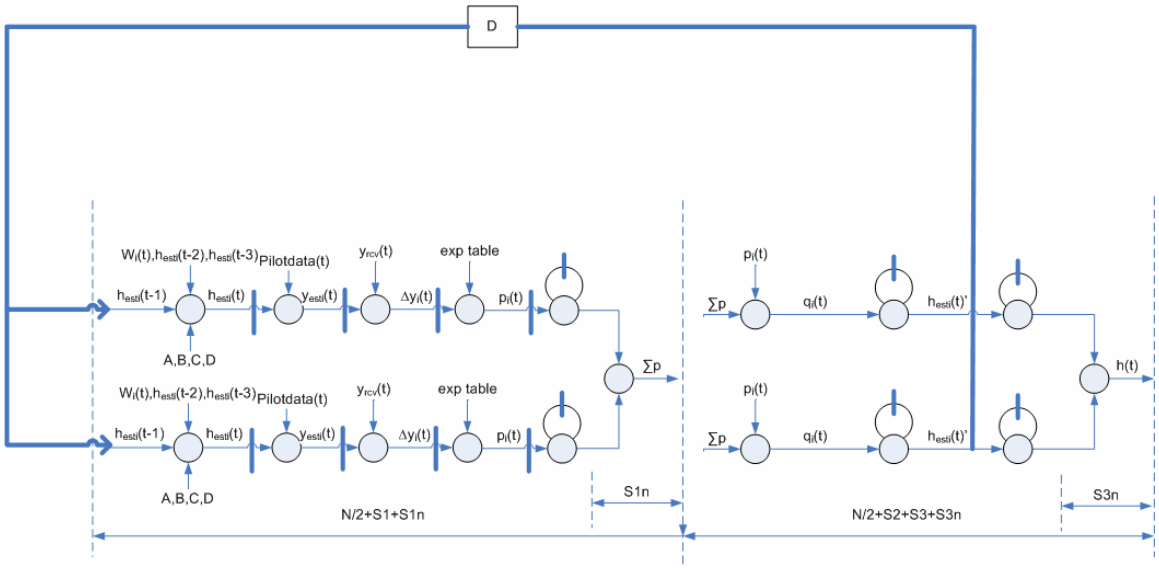

Figure 5. Architecture 4: Dual pipelines for each section
(improvement based architecture 3)

For section 1, multiple pipelines are straightforward: we only need to be careful of separate table for each pipeline. For section 2, due to the dependency, it seems impossible to implement multiple parallel pipelines for the calculation. But we find that if we do the update from both the head and the tail of the table simultaneously, we could have two pipelines. So this leads to dual pipelines for each section.

The overall iteration period is

$$\left(\frac{N}{2}+S_1+S_{1n}+\frac{N}{2}+S_2+S_3+S_{3n}\right)*T_{ps}=\left(N+S_1+S_{1n}+S_2+S_3+S_{3n}\right)*T_{ps}$$

The advantage and disadvantage of this architecture are:

Pros: quick.

Cons: complicated.

For our implementation target, architecture 3 is good enough. So after comparison, we decide to choose architecture 3.

Meanwhile we should point out that the above architectures are time(speed)-oriented architecture. We have not considered the possibility of reusing some calculation elements for different calculation, which might result in area efficient implementation(s).

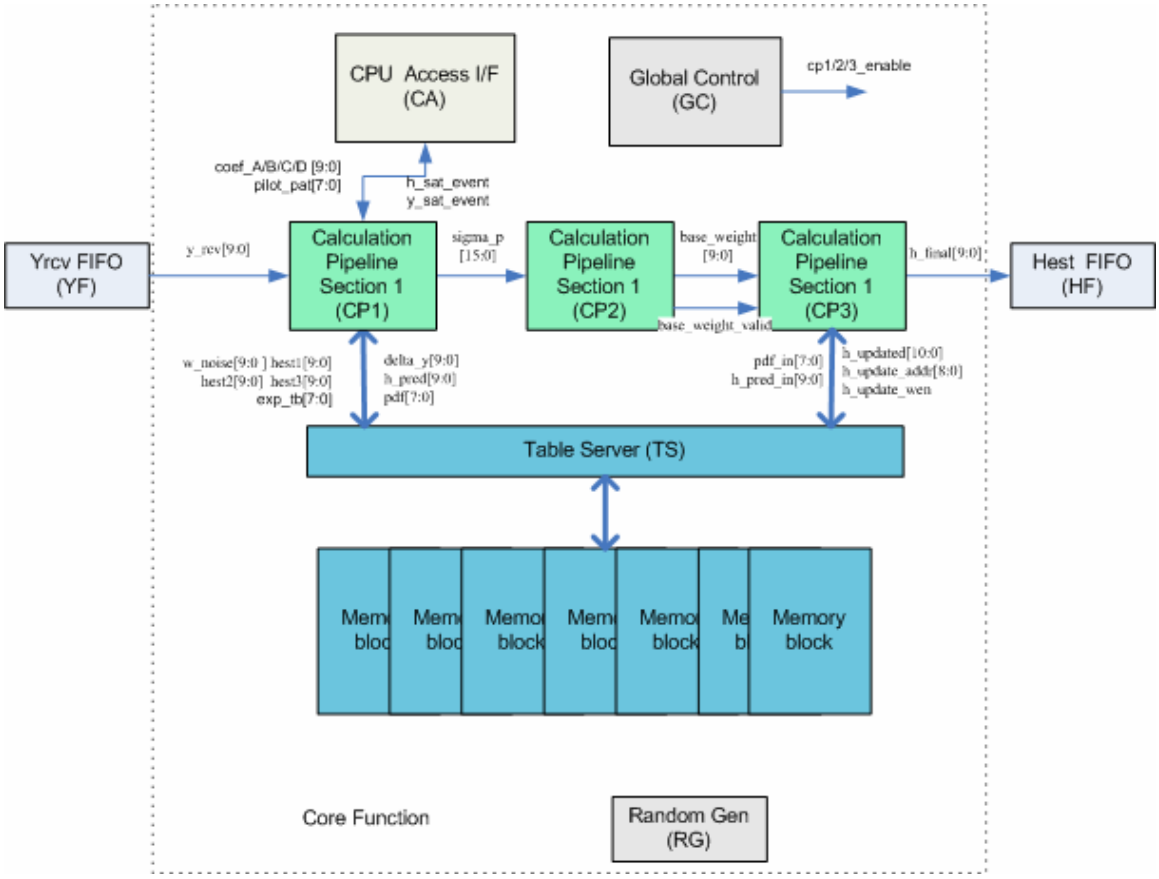

Figure 6. Physical Block Diagram

Fig. 6 is the physical block diagram of the project, which corresponds to the Verilog code of the design. The blocks in the “core function block” are the blocks to implement the

algorithm itself, while two more FIFO and some related registers, control logic are used for validation purpose. The blocks in the diagram are:

Global Control: FSM to generate global control/timing signal.

CPU Access Interface: The module implementing all registers, tables are listed in later section.

Random Gen: Random Number Generator (RNG) with Gaussian distribution.

Calculation Pipeline Section 1/2/3: The modules carrying out the calculation as discussed in previous sections.

Table Server and Memory Block: The module to implement all tables, coordinating all table access. There are 5 tables for $h_{estimated}(t\text{-}1)$, $h_{estimated}(t\text{-}2)$, $h_{estimated}(t\text{-}3)$, $h_{predicted}(t\text{-}1)$, $p(t)$ respectively, each of which consumes a block RAM. The physical memory for the four $h(t)$ tables are "rotated" after each iteration. There is also one exponential calculation table, which is pre-calculated, $2^{10} * 8$ bits.

Block Interface Definition

CA to CP1

Wire Name	Description
coef_A[9:0]	A
coef_**B**[9:0]	B
coef_C[9:0]	C
coef_**D**[9:0]	D
pilot_pat[7:0]	Pilot pattern

CP1 to CA

Wire Name	Description
h_sat_event	A one clock cycle pulse to indicate that saturation of the calculation of hpred happens.
y_sat_event	A one clock cycle pulse to indicate that saturation of the calculation of Yrcv happens.

YF to CP1

Wire Name	Description
y_rcv[9:0]	Received y

GC to CP1

Wire Name	Description
cp1_enable	=1: the pipeline should work; =0: the pipeline stalls

TS to CP1

Wire Name	Description
w_noise[9:0]	w
hest1[9:0]	hest(t-1)
hest2[9:0]	hest(t-2)
hest3[9:0]	hest(t-3)
exp_tb[7:0]	Exponential table looking-up result.

CP1 to TS

Wire Name	Description
delta_y[9:0]	Δy
h_pred[9:0]	hpred(t) written into the table
pdf[7:0]	p written into table

CP1 to CP2

Wire Name	Description
sigma_p[15:0]	$\sum p$

GC to CP2

Wire Name	Description
cp2_enable	=1: the pipeline should work; =0: the pipeline stalls

CP2 to CP3

Wire Name	Description
base_weight[9:0]	N/∑p

TS to CP3

Wire Name	Description
pdf_in[7:0]	p read out from talbe.
H_pred_in[9:0]	hpred(t) read out from the table

GC to CP3

Wire Name	Description
cp3_enable	=1: the pipeline should work; =0: the pipeline stalls

CP3 to TS

Wire Name	Description
h_updated[10:0]	Hupdated written into the talble
h_update_addr[8:0]	Address to write
h_update_wen	Write enable

CP3 to HF

Wire Name	Description
h_final[9:0]	Final estimation: h(t)

D Register Specifications

Access Legend:

- RO: Read Only. Write is ignored.
- RW: Read and Write.
- RWS: Write Side effect. Write of the register will trig a particular event.
- RW1C: Write one Clear. Write a "one" will clear the register. Used for interrupt status.
- RCL: Read Clear. Read from the register will clear the register. Used for event counter.

Control Register

MC (Master_Control) Address: 0000h

Bit	Name	Access	Reset Value	Description
15:5	RSV	RWS	0	Reserved
4	LOW_POWER	RWS	0	Puts the calculation core in low-power mode by gating its clock.
3	SINGLE_STEP_TRIGGER	RWS	0	Triggers a step forward when single-step mode is active.
2	SINGLE_STEP_MODE	RW	0	Enables single-step mode.
1	READY	RO	0	This bit indicates that the configuration of the chip is finished, and the chip could begin to work.
0	RESET	RWS	0	This bit should be written a "1" to initialize a global reset.

Configuration and Status Registers

Yrcv_FIFO (Yrcv FIFO Write) Address: 0020h

Bit	Name	Access	Reset Value	Description
15:12	RSV	RW	0	Reserved
11:0	YRCV	RWS	0	Yreceived should be written into this register when YRCV_NOT_FULL_BIT of GS register is set (i.e. the Yrcv FIFO could accept more data) is set.

Hest_FIFO (Hest FIFO Read) Address: 0030h

Bit	Name	Access	Reset Value	Description
15:10	RSV	RW	0	Reserved
9:0	HEST	RO	0	Hestimated' could be read out from this register when HEST_NOT_EMPTY_BIT of GS register is set (i.e. the Hest FIFO contains at least one data).

GS (General_Status) Address: 0044h

Bit	Name	Access	Reset Value	Description
15:12	RSV	RO	0	Reserved.
11	YRCV_LAST_WRITE_FAILURE	RO	0	=1: Last Yrcv FIFO write failed because it was full.
10	YRCV_EMPTY	RO	0	=1: Yrcv FIFO is empty.
9	YRCV_ALMOST_EMPTY	RO	0	=1: Yrcv FIFO is almost empty.
8	YRCV_NOT_FULL	RO	0	=1: at least one vacancy is available in the Yrcv FIFO.
7:4	RSV	RO	0	Reserved.
3	HEST_LAST_READ_FAILURE	RO	0	=1: Last Hest FIFO read failed because it was empty.
2	HEST_FULL	RO	0	=1: Hest FIFO is full.
1	HEST_ALMOST_FULL	RO	0	=1: Hest FIFO is almost full (only one free entry left).
0	HEST_NOT_EMPTY	RO	0	=1: at least one data is available in the Hest FIFO.

IS (Interrupt_Status) Address: 0046h

Bit	Name	Access	Reset Value	Description
15	ANY	RO	0	=1: at least one interrupt condition has happened.
14:11	RSV	RO	0	Reserved.
10	YRCV_EMPTY	RW1C	0	=1: Yrcv FIFO is empty.
9	YRCV_ALMOST_EMPTY	RW1C	0	=1: Yrcv FIFO is almost empty.
8	YRCV_NOT_FULL	RW1C	0	=1: at least one vacancy is available in the Yrcv FIFO.
7:3	RSV	RO	0	Reserved.
2	HEST_FULL	RW1C	0	=1: Hest FIFO is full.
1	HEST_ALMOST_FULL	RW1C	0	=1: Hest FIFO is almost full.
0	HEST_NOT_EMPTY	RW1C	0	=1: at least one data is available in the Hest FIFO.

IE (Interrupt_Enable) Address: 0048h

Bit	Name	Access	Reset Value	Description
15	GLOBAL_ENABLE	RW	0	=1: enable relevant interrupt; =0: disable relevant interrupt, however the bits in IS will still be set when corresponding events happen. GLOBAL_ENABLE must be set to allow any individual interrupt source to generate interrupt.
14:11	RSV	RO	0	
10	YRCV_EMPTY	RW	0	
9	YRCV_ALMOST_EMPTY	RW	0	
8	YRCV_NOT_FULL	RW	0	
7:3	RSV	RO	0	
2	HEST_FULL	RW	0	
1	HEST_ALMOST_FULL	RW	0	
0	HEST_NOT_EMPTY	RW	0	

COEF_A (Coefficient A) Address: 0080h

Bit	Name	Access	Reset Value	Description
15:10	RSV	RW	0	Reserved
9:0	COA	RW	0	The value of Coefficient A used to calculate the impulse response of the channel.

COEF_B (Coefficient B) Address: 0082h

Bit	Name	Access	Reset Value	Description
15:10	RSV	RW	0	Reserved
9:0	COB	RW	0	The value of Coefficient B used to calculate the impulse response of the channel.

COEF_C (Coefficient C) Address: 0084h

Bit	Name	Access	Reset Value	Description
15:10	RSV	RW	0	Reserved
9:0	COC	RW	0	The value of Coefficient C used to calculate the impulse response of the channel.

COEF_D (Coefficient D) Address: 0086h

Bit	Name	Access	Reset Value	Description
15:10	RSV	RW	0	Reserved
9:0	COD	RW	0	The value of Coefficient D used to calculate the impulse response of the channel.

PNS (Pseudo-Random Number Seed) Base Address: 0088h

Displacement Address	BIT 15:0	Access	Reset Value	Description
00h	PNS0	RW	xx	Seed 0 to generate the random number.
01h	PNS1	RW	xx	Seed 1 to generate the random number.
…				
07h	PNS7	RW	xx	Seed 7 to generate the random number.

Iteration_Counter Address: 0090h

Bit	Name	Access	Reset Value	Description
15:0	Counter	R	0	Last iteration processed by calculation core.

FSM_Status Address: 0092h

Bit	Name	Access	Reset Value	Description
15	RSV	RO	0	Reserved
14	CP3	RO	0	Calculation core is in CP3 state.
13	CP2	RO	0	Calculation core is in CP2 state.
12	CP1	RO	0	Calculation core is in CP1 state.
11	IDLE	RO	0	Calculation core is in idle state.
10:0	FSM_COUNT	RO	0	FSM step of calculation core.

PRODUCT_ID Address: 009Eh

Bit	Name	Access	Reset Value	Description
15:0	ID	RO	0x89AB	Hard-wired product ID

Hest1_Table Base Address: 0800h

Displacement Address	BIT 15:11	BIT 10	BIT 9:0	Access	Reset Value	Description
00h	RSV	NEW_REC	H(t-1)(0)	RW	xx[1]	Sample 0 of h(t-1).
01h	RSV	NEW_REC	H(t-1)(1)	RW	xx	Sample 1 of h(t-1).
…		NEW_REC				
01FFh	RSV	NEW_REC	H(t-1)(511)	RW	xx	Sample 511 of h(t-1).
03FF:0200h	RSV	RSV	RSV	RW	xx	Reserved (for 1K sample space)

Hest2_Table Base Address: 1000h

Displacement Address	BIT 15:11	BIT 10	BIT 9:0	Access	Reset Value	Description
00h	RSV	NEW_REC	H(t-2)(0)	RW	xx	Sample 0 of h(t-2).
01h	RSV	NEW_REC	H(t-2)(1)	RW	xx	Sample 1 of h(t-2).
…		NEW_REC				
01FFh	RSV	NEW_REC	H(t-2)(511)	RW	xx	Sample 511of h(t-2).
03FF:0200h	RSV	RSV	RSV	RW	xx	Reserved (for 1K sample space)

Hest3_Table Base Address: 1800h

Displacement Address	BIT 15:11	BIT 10	BIT 9:0	Access	Reset Value	Description
00h	RSV	NEW_REC	H(t-3)(0)	RW	xx	Sample 0 of h(t-3).
01h	RSV	NEW_REC	H(t-3)(1)	RW	xx	Sample 1 of h(t-3).
…		NEW_REC				
01FFh	RSV	NEW_REC	H(t-3)(511)	RW	xx	Sample 511of h(t-3).
03FF:0200h	RSV	RSV	RSV	RW	xx	Reserved (for 1K sample space)

EXP_Table (Exponential Table) Base Address: 2000h

Displacement Address	BIT 15:8	BIT 7:0	Access	Reset Value	Description
00h	RSV	EXP(0)	RW	xx	Entry 0 of EXP.
01h	RSV	EXP(1)	RW	xx	Entry 1of EXP.
…					
03Fh	RSV	EXP(1023)	RW	xx	Entry 1023of EXP.

[1] This table will be implemented in memory. Although in FPGA we could initialize the memory while configuring the FPGA, we generally do not assume to initialize the table by that method.

PDF_Table (Posteriori Distribution Table) Base Address: 2800h

Displacement Address	BIT 15:8	BIT 7:0	Access	Reset Value	Description
00h	RSV	P (0)	RO	xx	Entry 0 of PDF.
01h	RSV	P(1)	RO	xx	Entry 1of PDF.
…					
03Fh	RSV	P(1023)	RO	xx	Entry 1023of PDF.

HPRED_Table (Prediction Table) Base Address: 3000h

Displacement Address	BIT 15:11	BIT 9:0	Acc ess	Reset Value	Description
00h	RSV	HPRED(0)	RW	xx	Sample 0 of hpred
01h	RSV	HPRED(1)	RW	xx	Sample 1 of hpred
…					
01FFh	RSV	HPRED (511)	RW	xx	Sample 511of hpred
03FF:0200h	RSV	RSV	RW	xx	Reserved (for 1K sample space)

E PAR report (whole system resource usage)

```
Release 6.2.02i Par G.30
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

PCMAHDI::  Sun Jun 13 13:16:04 2004


par -w -ol high -pl high -rl high system_map.ncd system.ncd system.pcf


Constraints file: system.pcf

Loading device database for application Par from file "system_map.ncd".
   "system" is an NCD, version 2.38, device xc2v2000, package ff896, speed -4
Loading device for application Par from file '2v2000.nph' in environment
F:/Xilinx.
The STEPPING level for this design is 1.
Device speed data version:  PRODUCTION 1.118 2004-03-12.


Resolved that IOB <sys_clk> must be placed at site AH15.
Resolved that IOB <sys_rst> must be placed at site AH7.
Resolved that IOB <RS232_req_to_send> must be placed at site B8.
Resolved that IOB <RS232_RX> must be placed at site C8.
Resolved that IOB <RS232_TX> must be placed at site C9.


Device utilization summary:

   Number of External IOBs             5 out of 624     1%
      Number of LOCed External IOBs    5 out of 5     100%

   Number of MULT18X18s                9 out of 56     16%
   Number of RAMB16s                  39 out of 56     69%
   Number of SLICEs                 2264 out of 10752  21%

   Number of BSCANs                    1 out of 1     100%
   Number of BUFGMUXs                  5 out of 16     31%
   Number of DCMs                      1 out of 8      12%
   Number of TBUFs                   368 out of 5376    6%



Overall effort level (-ol):   Not applicable because -pl and -rl switches are used
Placer effort level (-pl):    High (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    High (set by user)

Starting initial Timing Analysis.  REAL time: 4 secs
Finished initial Timing Analysis.  REAL time: 10 secs
```

F PAR report (MCCE core alone)

```
Release 6.2.02i Par G.30
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

PCMAHDI::  Sun Jun 13 12:25:46 2004


F:/Xilinx/bin/nt/par.exe -w -intstyle ise -ol high -t 1 mcce_map.ncd mcce.ncd
mcce.pcf


Constraints file: mcce.pcf

Loading device database for application Par from file "mcce_map.ncd".
   "mcce" is an NCD, version 2.38, device xc2v2000, package ff896, speed -4
Loading device for application Par from file '2v2000.nph' in environment
F:/Xilinx.
The STEPPING level for this design is 1.
Device speed data version:  PRODUCTION 1.118 2004-03-12.

Device utilization summary:

   Number of External IOBs           111 out of 624    17%
      Number of LOCed External IOBs    0 out of 111     0%

   Number of MULT18X18s                6 out of 56     10%
   Number of RAMB16s                   7 out of 56     12%
   Number of SLICEs                 1170 out of 10752  10%

   Number of BUFGMUXs                  3 out of 16     18%
   Number of DCMs                      1 out of 8      12%
   Number of TBUFs                   368 out of 5376    6%


Overall effort level (-ol):   High (set by user)
Placer effort level (-pl):    High (set by user)
Placer cost table entry (-t): 1
Router effort level (-rl):    High (set by user)

Starting initial Timing Analysis.  REAL time: 2 secs
Finished initial Timing Analysis.  REAL time: 6 secs

Phase 1.1
Phase 1.1 (Checksum:9914df) REAL time: 11 secs

Phase 2.2
.
Phase 2.2 (Checksum:1312cfe) REAL time: 20 secs

Phase 3.3
Phase 3.3 (Checksum:1c9c37d) REAL time: 21 secs

Phase 4.5
Phase 4.5 (Checksum:26259fc) REAL time: 21 secs

Phase 5.8
.....................................................................
..
..
```

```
.........
.........................
..
Phase 5.8 (Checksum:c791aa) REAL time: 2 mins 39 secs

Phase 6.5
Phase 6.5 (Checksum:39386fa) REAL time: 2 mins 40 secs

Phase 7.18
Phase 7.18 (Checksum:42c1d79) REAL time: 2 mins 57 secs

Phase 8.24
Phase 8.24 (Checksum:4c4b3f8) REAL time: 2 mins 57 secs

Phase 9.27
Phase 9.27 (Checksum:55d4a77) REAL time: 2 mins 57 secs

Writing design to file mcce.ncd.

Total REAL time to Placer completion: 2 mins 59 secs
Total CPU time to Placer completion: 2 mins 43 secs

Phase 1: 9181 unrouted;       REAL time: 2 mins 59 secs

Phase 2: 7181 unrouted;       REAL time: 3 mins 8 secs

Phase 3: 1702 unrouted;       REAL time: 3 mins 11 secs

Phase 4: 1702 unrouted; (9537)      REAL time: 3 mins 11 secs

Phase 5: 1766 unrouted; (127)      REAL time: 3 mins 13 secs

Phase 6: 1775 unrouted; (0)      REAL time: 3 mins 13 secs

Phase 7: 0 unrouted; (104)      REAL time: 3 mins 20 secs

Writing design to file mcce.ncd.

Phase 8: 0 unrouted; (70)      REAL time: 3 mins 31 secs

Phase 9: 0 unrouted; (0)      REAL time: 3 mins 55 secs

Phase 10: 0 unrouted; (0)      REAL time: 3 mins 56 secs

Total REAL time to Router completion: 3 mins 57 secs
Total CPU time to Router completion: 3 mins 39 secs

Generating "par" statistics.

**************************
Generating Clock Report
**************************

+------------------------+----------+------+------+------------+-------------+
|        Clock Net       | Resource |Locked|Fanout|Net Skew(ns)|Max Delay(ns)|
+------------------------+----------+------+------+------------+-------------+
| CoreClk_port_OBUF      | BUFGMUX0P| No   |  648 |  0.360     |  1.527      |
+------------------------+----------+------+------+------------+-------------+
|          clk0_out      | BUFGMUX2P| No   |  221 |  0.323     |  1.490      |
+------------------------+----------+------+------+------------+-------------+
|CoreClk_ungated_port_OBU |          |      |      |            |             |
|                       F | BUFGMUX5S| No   |    4 |  0.007     |  1.424      |
+------------------------+----------+------+------+------------+-------------+
```

```
   The Delay Summary Report

   The SCORE FOR THIS DESIGN is: 324

The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        1.420
   The MAXIMUM PIN DELAY IS:                               9.877
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   9.099

   Listing Pin Delays by value: (nsec)

    d < 2.00   < d < 4.00  < d < 6.00  < d < 8.00  < d < 10.00  d >= 10.00
   ---------   ---------   ---------   ---------   ---------   ---------
        6927        1522         272          93          15           0

Timing Score: 0

Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

--------------------------------------------------------------------------------
  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_OPB_Clk = PERIOD TIMEGRP "OPB_Clk"  37 | N/A        | N/A        | N/A
   nS   HIGH 50.000000 %                    |            |            |
--------------------------------------------------------------------------------
  TS_clock_instance_name_CLK0_BUF = PERIOD  | 37.000ns   | 19.000ns   | 3
  TIMEGRP "clock_instance_name_CLK0_BUF"  T |            |            |
  S_OPB_Clk * 1.000000 HIGH 50.000 %        |            |            |
--------------------------------------------------------------------------------
  TS_clock_instance_name_CLKFX_BUF = PERIOD | 7.400ns    | 7.382ns    | 8
   TIMEGRP  "clock_instance_name_CLKFX_BUF" |            |            |
   TS_OPB_Clk / 5.000000 HIGH 50.000 %      |            |            |
--------------------------------------------------------------------------------

All constraints were met.
INFO:Timing:2761 - N/A entries in the Constraints list may indicate that the
   constraint does not cover any paths or that it has no requested value.
Generating Pad Report.

All signals are completely routed.

Total REAL time to PAR completion: 4 mins 1 secs
Total CPU time to PAR completion: 3 mins 42 secs

Peak Memory Usage:  189 MB

Placement: Completed - No errors found.
Routing: Completed - No errors found.
Timing: Completed - No errors found.

Writing design to file mcce.ncd.

PAR done.
```

G System floorplan

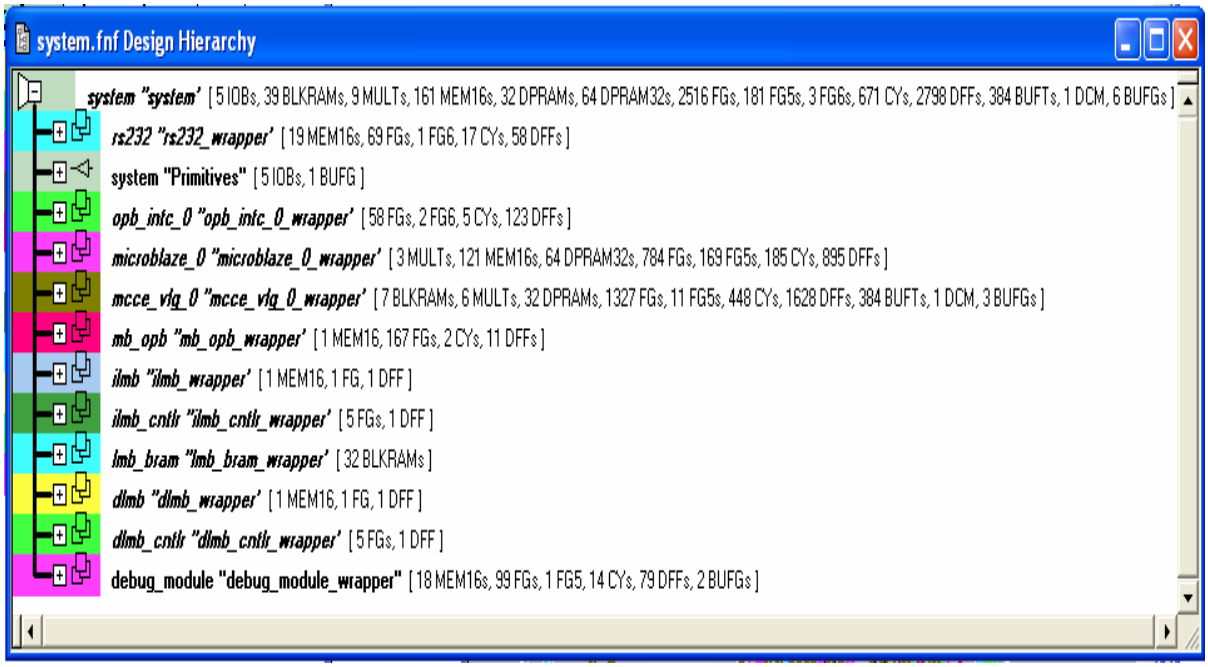

Figure G-1: Resource usage of system modules and their associated colors in FloorPlanner view

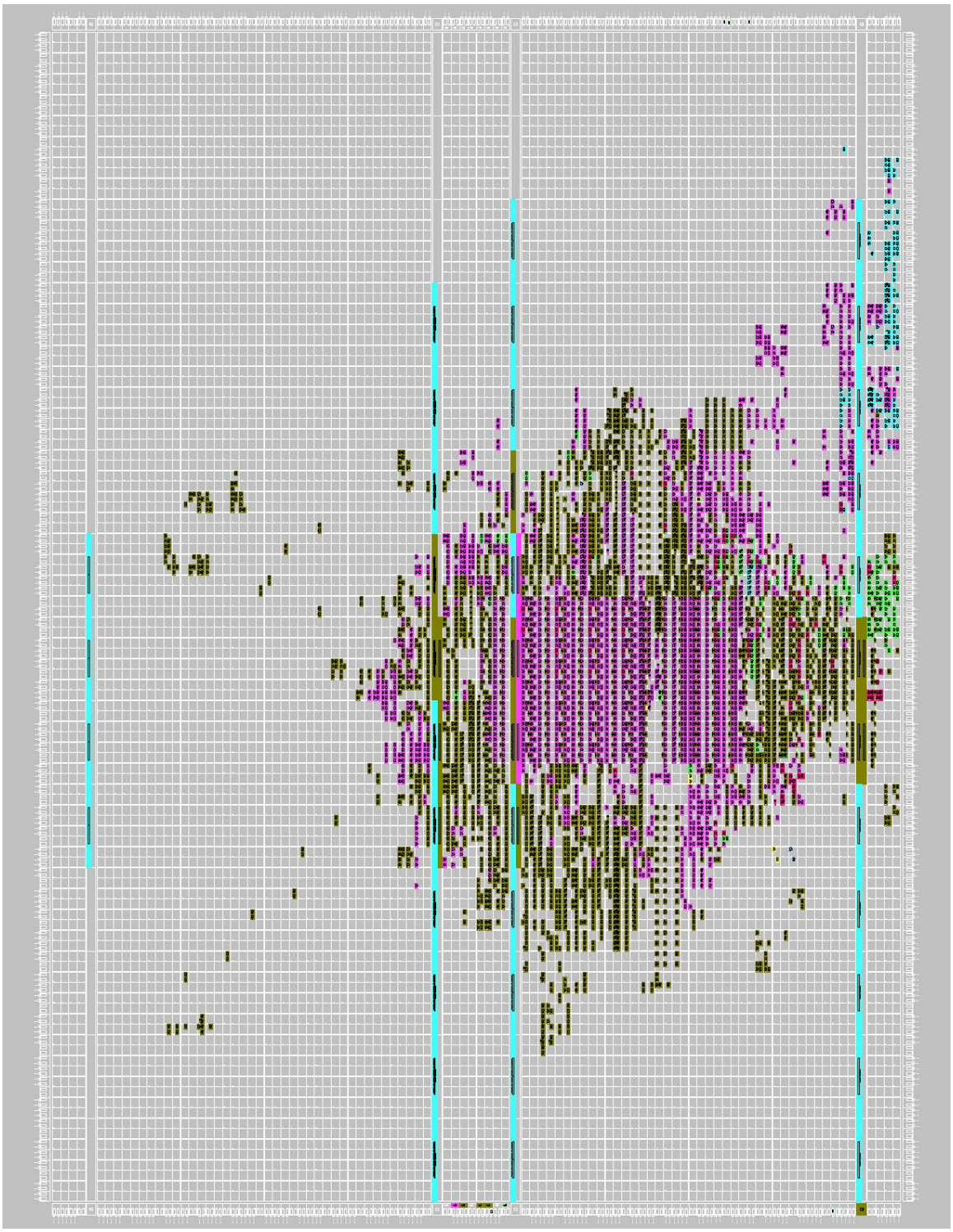

Figure G-2: Floor plan of the embedded system on XC2V2000

H Hierarchy of checked-in code

Below is description of the code hierarchy checked-in to CVS for this project. Please note that all paths are relative to the MCCE root path (e.g. */pc/r/r2/vlsicourse/vlsi2004/mcce*).

MCCE_rel: Contains the whole EDK project including embedded system, MicroBlaze software, MCCE core and its simulation code.

- MCCE core resides under $MCCE\_rel/pcores/mcce\_vlg\_v1\_00\_a$. Its */data* sub-directory includes MCCE core's Microprocessor Peripheral definition (MPD), Peripheral Analyze Order (PAO) and Black-Box Definition (BDD) files required by EDK for IP cores. The ISE project */hdl/verilog/MCCE.npl* under this path contains all HDL code for MCCE core. There are three source hierarchy there:
  - *TB_TOP* module which encapsulates the testbench for calculation core top module *top*. Sub-directories *table50* and *table1500* provide input, intermediate and expected data used by *TB_TOP* testbench.
  - *opb_bus_test_b* module generated by OPB BFC (Bus Functional Compiler) including the OPB bus *functional* testbench for the whole MCCE core (with top module *mcce*). All OPB Bus Toolkit Verilog HDL and test script files are put under $MCCE\_rel/pcores/mcce\_vlg\_v1\_00\_a/hdl/verilog/opb\_verilog$ folder.
  - *opb_bus_test_b_post_synthesis* module generated by OPB BFC (Bus Functional Compiler) including the OPB bus *post-PAR* testbench for the post-PAR model of MCCE core (with top module *mcce_timesim_post_synthesis*).
- MircoBlaze source codes are under $MCCE\_rel/Apps$. There are three revision of system software:
  - *basic* revision. It is capable of issuing our own MicroBlaze register read/write commands instead of relying on XMD buggy half-word access commands. Need to launch a terminal session on the PC side (at 9600 bps). This system software provides echo back and command string parsing without relying on *stdio* input and its string functions (linking with those library takes lots of space).

- *testbench* revision. Added feature compared to *basic* edition is the 50 iterations *testbench* capability.
- *demo* revision. Relys on PC software to parse user command, prepare and submit the commands. Only process command packets. Supports both interrupt and polling-based FIFO read/write access. Simple register access commands supported as previous editions of system software.

hdl: Contains same copy of MCCE core source files.

PC_demo: Includes PC software created for demo. It is based on Microsoft Foundation Class (MFC) written with Visual C++.Net. Refer to section 6.3 of report for more info.

Matlab: Includes all MATLAB code simulating the algorithm in both ideal and bit-true modes.

doc: Contains this document.

TestBuilder: TestBuilder code.