Logic circuit design for high-speed computing of dynamic response in real-time hybrid simulation using FPGA-based system

Akira Igarashi*

Disaster Prevention Research Institute, Kyoto University, Gokasho, Uji-shi, Kyoto 611-0011 Japan

(Received March 30, 2014, Revised July 20, 2014, Accepted August 2, 2014)

Abstract. One of the issues in extending the range of applicable problems of real-time hybrid simulation is the computation speed of the simulator when large-scale computational models with a large number of DOF are used. In this study, functionality of real-time dynamic simulation of MDOF systems is achieved by creating a logic circuit that performs the step-by-step numerical time integration of the equations of motion of the system. The designed logic circuit can be implemented to an FPGA-based system; FPGA (Field Programmable Gate Array) allows large-scale parallel computing by implementing a number of arithmetic operators within the device. The operator splitting method is used as the numerical time integration scheme. The logic circuit consists of blocks of circuits that perform numerical arithmetic operations that appear in the integration scheme, including addition and multiplication of floating-point numbers, registers to store the intermediate data, and data busses connecting these elements to transmit various information including the floating-point numerical data among them. Case study on several types of linear and nonlinear MDOF system models shows that use of resource sharing in logic synthesis is crucial for effective application of FPGA to real-time dynamic simulation of structural response with time step interval of 1 ms.

Keywords: real-time processing; fast computing; parallel processing; logic circuit

1. Introduction

As a testing method to experimentally evaluate the dynamic response of structures and structural components, in which nonlinearity and experimental measurement of force-displacement response are of main concern, the hybrid simulation has been widely accepted as an efficient approach for the dynamic testing; in the hybrid simulation, response of the original structural system is simulated by loading test performed only for the part for which numerical modeling is difficult, and numerical analysis of the rest of the structural system using the computer, and the two process are synchronously executed exchanging the information each other. The test method to perform the hybrid simulation in a real-time basis in order to evaluate the response considering the dynamic effect and loading rate dependence is referred to as the real-time hybrid simulation (Shao 2011). In a test of this type, a shake table and/or a dynamic actuator that allow high-rate loading are assumed to be used as the test equipments. An example of real-time hybrid simulation

*Corresponding author, Professor, E-mail: igarashi.akira.7m@kyoto-u.ac.jp concept using a shake table is shown in Fig. 1.

One of the issues in extending the range of applicable problems of real-time hybrid simulation is the computation speed of the simulator when a large-scale computational model is used. If the number of DOF of the structural model is large, one of the most important problems is the duration of the time required for a single-step test control, starting from the timing of the computer's acquiring the measurement data from the test structure response, time integration to evaluate the structural response after a unit step, up to the computation of input control signal to loading equipment such as a shake table. Response analyses of a model with a large number of DOF or a model including complicated nonlinear behavior require longer computation time, and when such factors dominate the computation process on a computer with limited computational capacity, the computation eventually results in failure to complete a single step calculation, thus losing the real-time computational process. In real-time tests, high-speed computation is expected to ensure the real-time processing and control of the test, including another important aspect of dynamic characteristics compensation of the experimental loading equipments. Saouma *et al.* (2012) pointed out the importance of the performance of computational engine for real-time hybrid simulation, and developed a specialized structural analysis software applicable to real-time hybrid simulation running on a computer with real-time Linux, and conducted a real-time hybrid simulation of a three-story three-bay reinforced concrete frame, numerically modeled with nonlinear elements and more than 400 DOF with the integration interval of 0.01s (Saouma 2013).

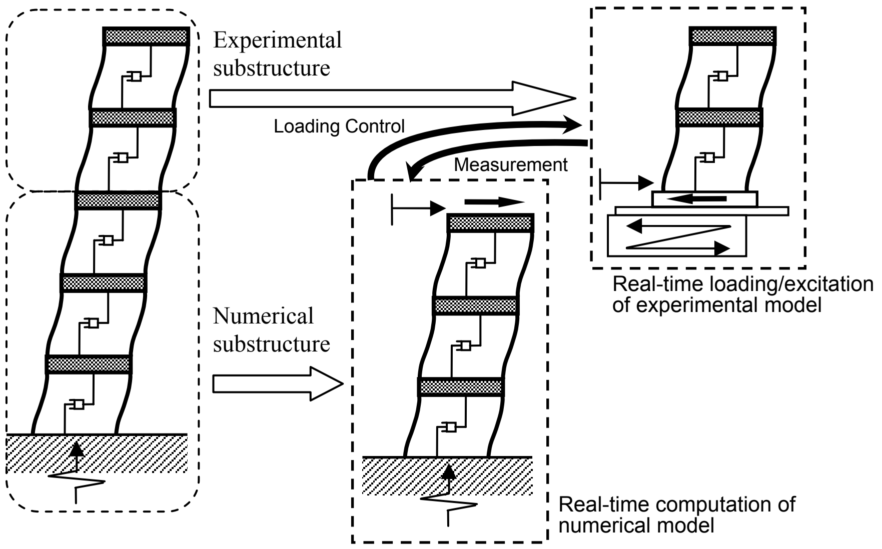

Experimental substructure
Loading Control
Measurement
Real-time loading/excitation of experimental model
Numerical substructure
Real-time computation of numerical model

Fig. 1 Example of real-time hybrid simulation concept

The potential performance of FPGA application to numerical computation is shown in past research. Schäfer *et al.* (2002) applied FPGA to DEM (discrete element method) computation showing FPGA's advantage in the exploitation of the intrinsic parallelism of DEM, and achieved a faster computation speed. Hamada *et al.* (2006) constructed an FPGA-based computer system dedicated to SPH method (Smoothed Particle Hydrodynamics method) fluid dynamic simulation in astrophysics, achieving computing performance and a cost performance ratio far superior to mainframe parallel computer systems at the time and feasibility of the floating point number operation by FPGA is demonstrated. Cui *et al.* (2003) proposed a logic circuit exclusively for matrix operations using FPGA, and showed that the matrix operation processing time can be shortened compared with mainframe computers. Onouchi *et al.* (2003) constructed a quantum computer emulator capable of large-scale parallel processing taking advantage of the parallel computing capability of FGPA, in which numerical arithmetic operators can be configured as many units as required, and showed that for an NP problem (Non-deterministic Polynomial problem) that requires exponentially growing processing time with respect to the size of the problem, computational time to solve the satisfiability problem (SAT problem) can be shortened in the order of 1/100 compared with the conventional computer architecture. Application of FPGA is a promising in achieving an improvement of computation speed and expansion of applicable problems and fields, compared with the use of DSP (digital signal processor) which has been used in the conventional hybrid simulation. The performance and capacity of FPGA device products available in the commercial market are remarkably growing, owing to the increasing need in the electronic device product industry, and expected to effectively take advantage of their capabilities. In this study, design, verification and implementation of logic circuits are investigated in order to achieve high-speed execution to be used in real-time hybrid simulation of MDOF structural system models using FPGA.

2. General feature of FPGA

The FPGA is classified as a kind of PLD (Programmable Logic Device) that is regarded as an intermediate IC (Integrated Circuit) category placed between the general-purpose LSI (Large Scale Integrated circuit) and ASIC (Application Specific Integrated Circuit). The FPGA consists of logic circuit blocks (which is referred to as Logic Elements, or LE), and a logic circuit designated by the user can be constructed by changing the combination of the logic elements and wiring among them at the time of set-up. Thus FPGA allows a high level of flexibility for the users to design the logic circuit in the device. Concept of the internal structure of FPGA is shown in Fig. 2. As shown in the figure, a large number of logic elements that are arranged to form a grid, and the logic circuit functionality is obtained by the assignment of the connection pattern at the grid nodes. Furthermore, several peripheral elements are added, including the I/O elements, which are the assemblies of input/output pins to be used as the interface with external devices, the DSP blocks to provide number multiplication functionality often used in digital data processing, and the configuration memories to store the program file (bit stream data) which works as the logic circuit information implemented in FPGA.

In general, a Hardware Description Language (HDL) is used for the purpose of expressing the logic circuit implemented on the FPGA device. Based on the description, the logic synthesis is performed to convert the HDL code into the logic circuit wiring information, arrangement of geometric circuit element layout and signal line routing on the target device. The resulting information expressed as bit stream data is downloaded into a FPGA device to implement the logic circuit and I/O pin assignment. After completing these preparation processes, FPGA can be utilized for the specific purpose at the time of design.

3. Design of logic circuit for real-time dynamic response simulator

3.1 General

Real-time dynamic response simulation is executed on a FPGA device by implementing a logic circuit that performs binary numerical computation. In this study, the logic circuit is designed using Verilog-HDL as one of the commonly used hardware description languages.

Owing to the needs of rapid development of embedded systems by electrical and device engineers, software and hardware vendors provide various coding tools and environments nowadays. For example, there exists even a commercial product with which HDL codes can be automatically generated from MATLAB Simulink block diagrams, called MATLAB HDL Coder (2012). However, the MATLAB HDL Coder deals with computation with only fixed-point arithmetic, that requires adjustments to obtain appropriate performance of the system in terms of accuracy and efficiency in using the device resources. Although existing computational simulation software codes used for a large computational simulation in real-time hybrid simulation may be implemented to FPGA if a tool to convert those codes written in software programming languages into HDL code is developed, such conversion tools have not appeared yet up to the present. In this study, in order to use the floating-point computation in the dynamic simulation of the numerical structural model to eliminate the problems and errors associated with the fixed-point arithmetic, the logic circuit for the numerical system is explicitly designed, and elementary components of the numerical floating-point arithmetic operation are developed. The data bus to send and receive numerical values in logic circuits is based on the single precision 32-bit floating point number representation (8-bit exponent and 23-bit mantissa) designated by IEEE 754 standard; for comparison, design cases using 16-bit representation (5-bit exponent and 10-bit mantissa) are also investigated.

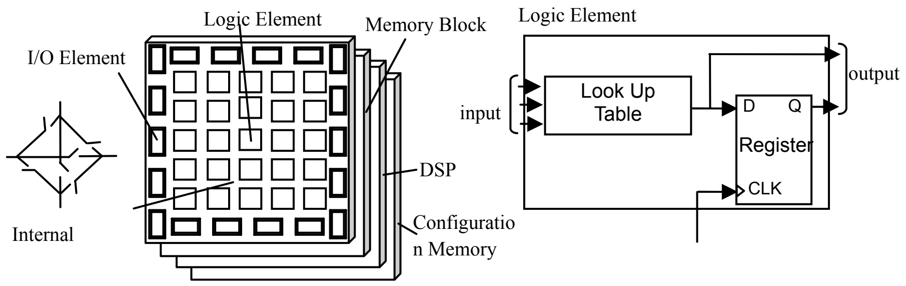


Fig. 2 Internal structure of FPGA

Functionality of real-time dynamic simulation of MDOF systems is achieved by creating a logic circuit that performs the step-by-step numerical time integration of the equations of motion of the system. The operator splitting method is used as the numerical time integration scheme. In order to execute the computational steps of the numerical time integration, the outline of the logic circuit consists of blocks of logic circuits that perform numerical arithmetic operations that appear in the integration scheme, such as addition and multiplication of floating-point numbers (that are referred to as adders and multipliers, respectively), registers to store the intermediate data, and data busses connecting these elements to transmit various information including the floating-point numerical data among them.

3.2 Numerical arithmetic operators

The logic circuit blocks that perform the basic arithmetic operations are the most essential and indispensable components in dealing with numerical computation by FPGA. The dynamic response simulation is performed by associating these arithmetic operators, including adders and multipliers, in the top-level module in the HDL script.

The amount of logic element consumption for the logic circuit implemented in the FPGA changes depending on the data bus bit width for the floating point number representation. In order to investigate the difference in the floating point number precision, generalized basic arithmetic operators that allow arbitrary bit numbers of the exponent and mantissa are designed. The flowcharts of the adder and the multiplier are shown in Fig. 3. The procedure consists of checking the input arguments (ex. zero exponent), arithmetic operation, correction and exception handling (ex. overflow/underflow) and output. The adder is relatively complicated due to additional operations of comparison of the two input arguments, rearrangement of the input arguments in the order of the absolute value, and digit shift operation to the smaller argument for digit matching. The Verilog-HLD codes for the adder and the multiplier are shown in Appendix A and Appendix B, respectively.

3.3 Implementation of numerical time integration scheme

In order to implement the computation of the numerical time integration of the equations of motion - an essential function to perform real-time hybrid simulation - with logic circuits, the logic system is designed so that adders, multipliers etc. that perform basic arithmetic operations appearing in the integration scheme, and interim data registers are mutually connected with wirings that transmit information including numerical values.

At each time step i (i=1,2,…), the equation of motion for a linear MDOF system is assumed to be generally expressed by the following expression.

$$[M]\{\ddot{x}_{i+1}\}+[C]\{\dot{x}_{i+1}\}+[K]\{x_{i+1}\}=-[M]\{1\}\ddot{z}_{i+1}-\{F_e^{i+1}\} \tag{1}$$

where $[M]$, $[C]$ and $[K]$ are the mass, damping and stiffness matrices, respectively, $\{x_i\}$ is the displacement vector at time step i, $\{F_e^{\,i}\}$ is the measured force acting on the experimental substructure. In this expression, the external force acting on the structural system is assumed to be the dynamic excitation due to the ground acceleration $\ddot{z}$ at the support of the structure. For nonlinear structures, the term $[K]\{x_i\}$ is replaced with the nonlinear restoring force vector $\{Q_i\}$. For illustrative purpose, application of the operator splitting scheme to a linear MDOF system and resulting implementation is described. The set of formula in this case is as follows.

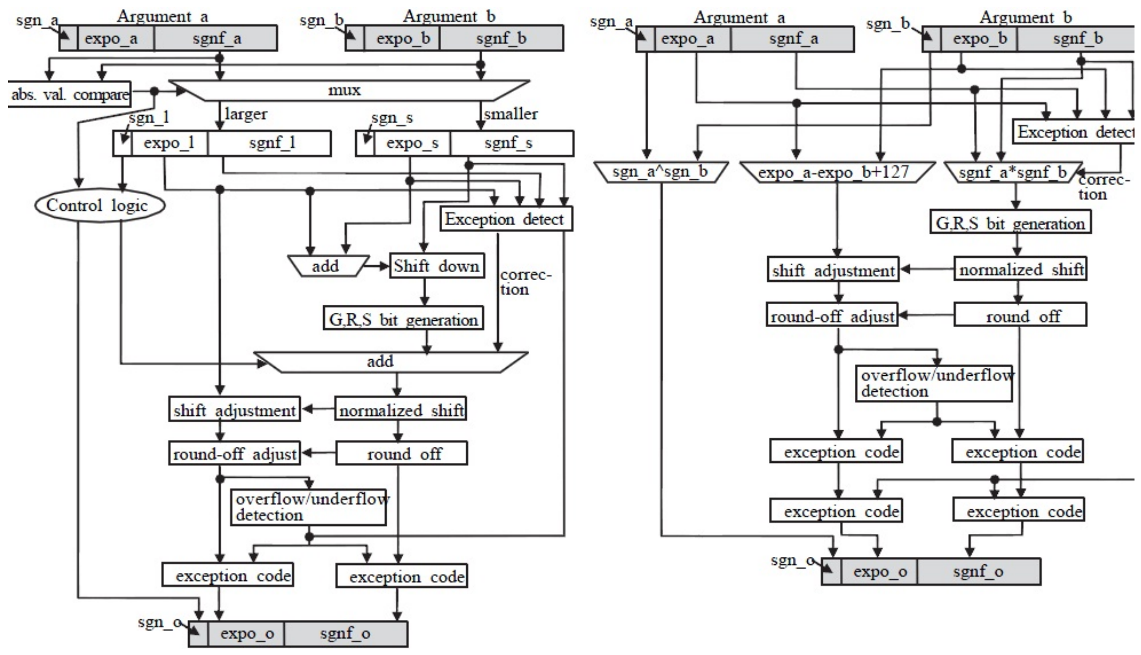
$$\{\tilde{x}_{i+1}\} = \{x_i\} + \Delta t\{\dot{x}_i\} + \frac{\Delta t^2}{4}\{\ddot{x}_i\} \tag{2}$$

$$\{\ddot{x}_{i+1}\} = \left[[M] + \frac{\Delta t}{2}[C] + \frac{\Delta t^2}{4}[K]\right]^{-1}\left[-[M]\{1\}\ddot{z}_{i+1} - [C]\left\{\{\dot{x}_i\} + \frac{\Delta t}{2}\{\ddot{x}_i\}\right\} - [K]\{\tilde{x}_i\} - \{F_e{}^i\}\right] \tag{3}$$

$$\{\dot{x}_{i+1}\} = \{\dot{x}_i\} + \frac{\Delta t}{2}\left(\{\ddot{x}_i\} + \{\ddot{x}_{i+1}\}\right) \tag{4}$$

$$\{x_{i+1}\} = \{\tilde{x}_{i+1}\} + \frac{\Delta t^2}{4}\{\ddot{x}_{i+1}\} \tag{5}$$

where $\{\tilde{x}_i\}$ is the displacement predictor, which is a vector with the identical dimensions as $\{x_i\}$. In order to reduce the amount of computation in real-time computation in FPGA, all the values that can be prepared are separately computed and stored to registers in FPGA before the beginning of the step-by-step time integration process. The procedure steps consisting of Step 1 through Step 4 that have been obtained to implement to a logic circuit is shown Table 1.



(a) Adder (b) Multiplier

Fig. 3 Flowcharts of adder and multiplier

The symbols $\{r_1\}$～$\{r_{10}\}$ appearing in the table are the variables that express the interim computation results stored in registers, and N is the number of DOF of the numerical model. The operations described in the same clock are performed simultaneously in parallel. The required numbers of clocks and operations for each step are for the case such that $[M]$ is a diagonal matrix, and $[C]$, $[K]$ are banded matrices with a width of 3.

The most time-consuming and memory-consuming phase in the time integration is the product-and-sum operation appearing in Step 3 in Table 1. For the case of an N-DOF system model, N sets of product-and-sum numerical operations for N terms take place. If the number of adders that can be used in parallel are not limited, the required number of clocks of those product-and-sum operations is equal to j that satisfy the following expression.

$$2^{j-1} < N \le 2^j \tag{6}$$

In the sum operation of 100 terms, for example, the number of the terms to be added changes as 100→50→25→13→7→4→2→1 as the clock proceeds (in j=7 clocks), and the number of required operations is 50+25+12+6+3+2+1=99.

Table 1 Procedure of Operator Splitting scheme in parallel computing

Step	No.of opera-tions	Clock	Operation	No.of opera-tions	Note
1	$4N$	1	$\Delta t \times \{\dot{x}_i\} \to \{r_1\}$ $(\Delta t^2/4) \times \{\ddot{x}_i\} \to \{r_2\}$	$2N$	Computation of $\{\tilde{x}_{i+1}\}$
		2	$\{x_i\} + \{r_1\} \to \{r_3\}$	N	
		3	$\{r_2\} + \{r_3\} \to \{\tilde{x}_{i+1}\}$	N	
2	$15N$-7	1	$-[M]\{1\} \times \ddot{z}_{i+1} \to \{r_4\}$ $(\Delta t/2) \times \{\ddot{x}_i\} \to \{r_5\}$	N N	Computation of [N×1 matrix] = $\left[-[M]\{1\}\ddot{z}_{i+1} - [C]\left\{\{\dot{x}_i\} + \frac{\Delta t}{2}\{\ddot{x}_i\}\right\} - [K]\{\tilde{x}_{i+1}\} - \{F_e^{i+1}\}\right]$
		2	$\{\dot{x}_i\} + \{r_5\} \to \{r_6\}$	N	
		3	The elements of $-[C] \times \{r_6\} \to \{r_7\}$	$3N$-2	
		4～7	The elements of $-[K] \times \{\tilde{x}_{i+1}\} \to \{r_8\}$ Sums of the result → [N×1matrix]	$9N$-5	
3	N^2+ $O(N^2)$	1	Product between the elements of [N×N matrix] and [N×1matrix]	N^2	Computation of $\{\ddot{x}_{i+1}\}$ =[N×N matrix] [N×1 matrix]
		2～ $[\log_2 N^2]$	Sum of the results	$O(N^2)$	
4	$4N$+1	1	$(\Delta t/2) \times \{\ddot{x}_{i+1}\} \to \{r_9\}$ $(\Delta t^2/4) \times \{\ddot{x}_{i+1}\} \to \{r_{10}\}$ $\ddot{z}_{i+1} + \{\ddot{x}_{i+1}\} \to a_{i+1}$	$2N$+1	Computation of $\{\dot{x}_{i+1}\}$, $\{x_{i+1}\}$, a_{i+1} (for test control)
		2	$\{\dot{x}_i\} + \{r_9\} \to \{\dot{x}_{i+1}\}$ $\{\tilde{x}_{i+1}\} + \{r_{10}\} \to \{x_{i+1}\}$	$2N$	

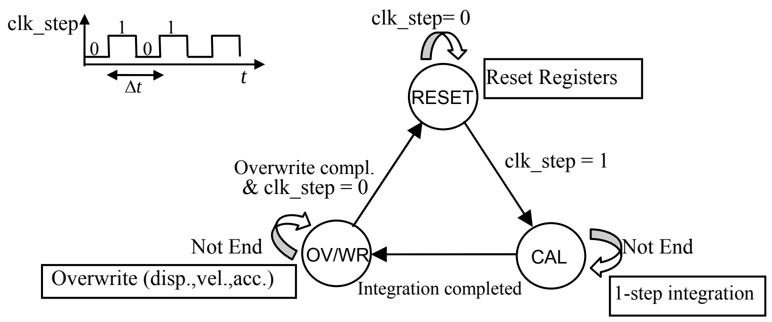


Fig. 4 State transition diagram

This procedure is iteratively repeated for each time step *i* using the result of the previous time step. For implementation of this iterative algorithm in a logic circuit, a finite automaton structure is employed as a synchronous circuit with a single phase clock with synchronous reset. The state transition diagram of the circuit is shown in Fig. 4. The action of the circuit is determined by the step clock signal and “states” modified in accordance with the progress of the calculation and timing. There are three states in the circuit, RESET, CAL and OV/WR. The initial state is RESET, at which the resisters are reset, and holds until a rising edge of the time step clock signal at the start of each time step. A time integration per step is performed at the state CAL, which is followed by the overwrite operation of the response variables at the state OV/WR. The timing of the real-time execution speed of the time steps can be adjusted by changing the frequency of the time step clock signal (ex. 1 kHz).

4. Verification of FPGA implementation

Logic circuits are designed for the following four dynamic models in accordance with the methodology described in the previous section.

(1) Linear SDOF model: As shown in Fig. 5(a), the structural model is assumed to be a linear 2-DOF system, in which the numerical substructure in the response simulator is a linear SDOF system, and the hypothetical experimental substructure also is assumed to be a linear SDOF system for the convenience of simulation and is included in the FPGA simulation.

(2) Shear-type Linear MDOF model: Structural model is assumed to be a shear-type multi-story MDOF system, in which the top story is the experimental substructure and the rest of the system is the numerical substructure, as shown in Fig. 5(b). The number of DOF of the numerical substructure, denoted by *N*, can be changed to investigate the influence of *N* to the implementation capacity of FPGA, as described later.

(3) General Linear MDOF model: The assumed numerical substructure is a linear MDOF system expressed with a general representation by the characteristic matrices (mass, damping and stiffness matrices), while the response of the hypothetical experimental substructure is simulated by implementing a linear SDOF system.

(4) Shear-type Nonlinear MDOF model: As shown in Fig. 5(d), the assumed numerical substructure is a shear-type multi-story structural model, and each shear spring is modelled as the

bilinear hysteretic model. The response of the hypothetical experimental substructure is simulated by implementing a linear SDOF system as in the previous models.

An example of the logic circuit output is shown by the most fundamental Linear SDOF model. The logic circuit of response simulator is configured to calculate the dynamic response of a SDOF system with a mass of 3450 kg, natural frequency of 1.0 Hz, and damping ratio of 0.05. In order to simulate the situation of real-time hybrid simulation, response of a hypothetical experimental substructure of a mass of 34.5 kg, natural frequency of 2.0 Hz and damping ratio 0.05 is synchronously simulated. The integration time interval is 0.001sec, and the El Centro record NS component is used as the ground excitation input. The logic circuit for the dynamic simulation is verified by using the logic circuit simulator ModelSim.

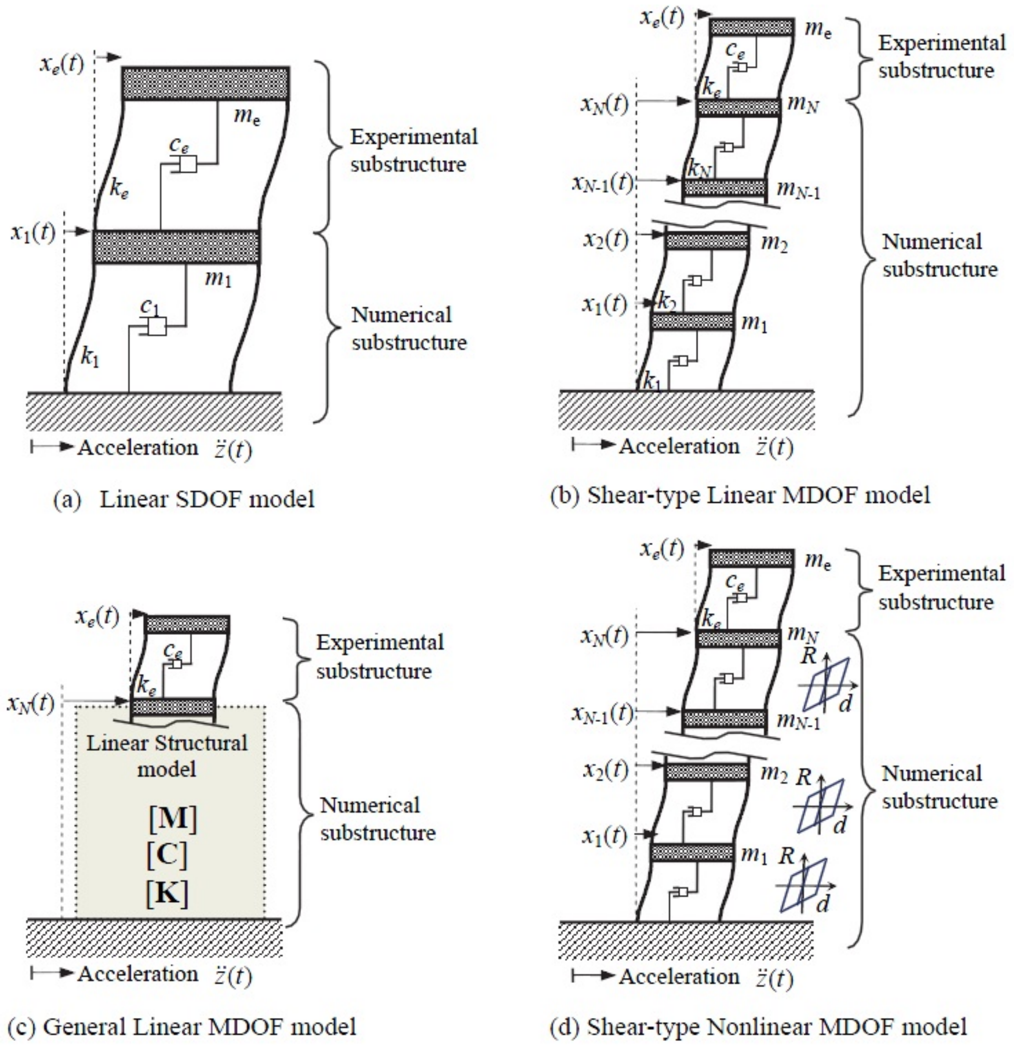


Fig. 5 Assumed structural models

The result is shown in Fig. 6. The numerical simulation of the dynamic response of the same MDOF structural system by MATLAB software is also shown in the figure for comparison. The result of the output of the logic circuit is confirmed to be identical to the result of the software numerical simulation. Relevance and accuracy of the output of the logic circuit for Shear-type Linear MDOF model, General Linear MDOF model, and Shear-type Nonlinear MDOF model are also verified in the same manner.

As the next step, implementation of the designed logic circuit to an actual FPGA and verification are performed. The FPGA board used for implementation and verification is DE2-70, shown in Fig. 7, accommodating Cyclone II EP2C70 as the FPGA device (Altera Co. 2007). The output of the dynamic response simulator implemented to the FPGA device and usual numerical calculation result show perfect agreement within the accuracy of numerical computation, confirming that the designed logic circuit functions as designed on the actual FPGA device.

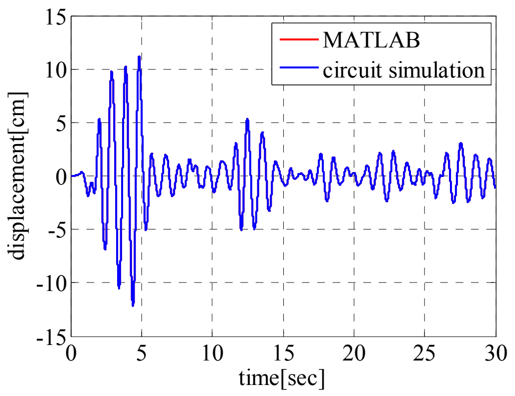


Fig. 6 Logic circuit output for Linear SDOF case

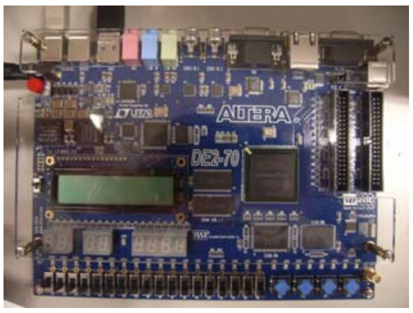


Fig. 7 Altera DE2-70 board accommodating Cyclone II FPGA

Table 2 Specification of Cyclone II EP2C70

Logic Elements	DSP blocks	I/O pin	RAM
68416	150	622	1152000 bits

Table 3 Specification of DE2-70 board

7-segment LED	Red/green LED	Toggle switches	Oscillators
9	18/9	18	50MHz, 28.63MHz

5. Evaluation of computational performance limit

5.1 Conditions and assumptions

Limitation of computational performance of a single FPGA unit in terms of the speed of the processing and the number of DOF of the applied structural model is investigated. The two precision cases (32bit and 16bit accuracies) are considered.

The Shear-type Linear MDOF model, General Linear MDOF model and Shear-type Nonlinear MDOF model are used for the evaluation. For the Shear-type Linear MDOF model, the characteristic matrices are narrowly banded as in Eq. (7).

$$[M]=\begin{bmatrix} * & & & 0 \\ & * & & \\ & & \ddots & \\ 0 & & & * \end{bmatrix},\ [C],[K]=\begin{bmatrix} * & * & & 0 \\ * & * & \ddots & \\ & \ddots & \ddots & * \\ 0 & & * & * \end{bmatrix} \tag{7}$$

where the symbol * indicates the non-zero components. It should be noted that all the characteristic matrices [*M*], [*C*], [*K*] become fully specified dense matrices in the General Linear MDOF models. In the General Linear MDOF model in which the characteristic matrices are assumed to be dense, the number of required computation steps more rapidly increases compared with the shear-type counterpart. The Shear-type Nonlinear MDOF model is specified as a shear-type model using shear springs with bilinear hysteretic restoring force characteristics, as previously described.

For simplicity of the evaluation, only the use of a single FPGA is assumed, and Altera Stratix IV (Altera 2009) is employed as the target FPGA device in the case study, since this product is a commercially available FPGA device with a reasonably high capacity. Since the number of logic elements of Stratix IV is 681000, and it is recommended to design the implemented logic circuit to use up to 80% of the number of logic elements, the limit of the number of logic elements used for implementation is assumed to be 544880.

Estimation of the number of consumed logic elements n_{LE} is given by the following equation

$$n_{LE}=\sum m_{op\,i}n_{LE\,op\,i}+m_{reg}n_{bit} \tag{8}$$

where $m_{op\,i}$ is the number of arithmetic operators type *i* (adder, multiplier, divider), $n_{LE\,op\,i}$ is the number of required logic elements for the type of arithmetic operator *i*, m_{reg} is the number of required registers, and n_{bit} is the data bit width. The values of $n_{LE\ op\ i}$ determined by logic synthesis with the design tool software Quartus II Web Ed. ver 9.2 from the Verilog-HDL codes are presented in Table 4.

The number of required register is estimated by Eq. (9).

$$m_{reg} = n_{operation} + 3N \tag{9}$$

where $n_{operation}$ is the number of arithmetic operations and N is the number of DOF of the structural model. Accuracy of the estimated number of logic elements computed by the above equations has been checked by comparison with the corresponding result of logic synthesis using Quartus II software.

Evaluation of the required number of clocks is performed by summing all the clock numbers appearing in the step timing tabulation for the assumed procedure, in the form as illustrated in Table 1. Processing time per time step can also be derived by the number of required clocks divided by the maximum clock frequency, which depends on logic synthesis result and signal delay due to routing length and layout within FPGA. In this study, the maximum clock frequency is assumed to be 23.4MHz for 32 bit data case, and 30.5 MHz for 16 bit data case, based on the result from the logic synthesis result from Quartus II software.

5.2 Design case for fastest processing

Required numbers of consumed logic elements and of clocks are examined, for the design of logic circuits aiming at the maximum response simulation speed, implying the shortest completion time for computation of each time step.

For the Shear-type Linear MDOF model, the required numbers of operations are obtained as

$$n_{multiplication} = N^2 + 12N - 3 \tag{10}$$

$$n_{addition} = 11N - 1 + N^2 + O(N^2) \tag{11}$$

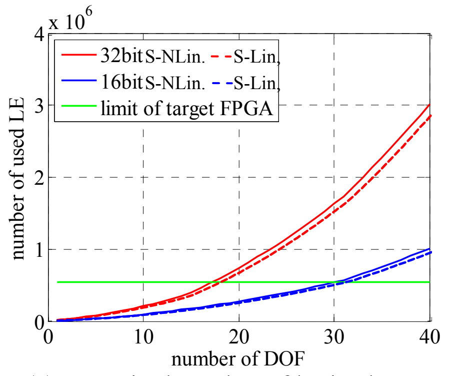
For the Shear-type Nonlinear MDOF model, the required numbers of operations are obtained as in the following equations.

$$n_{multiplication} = N^2 + 13N - 1 \tag{12}$$

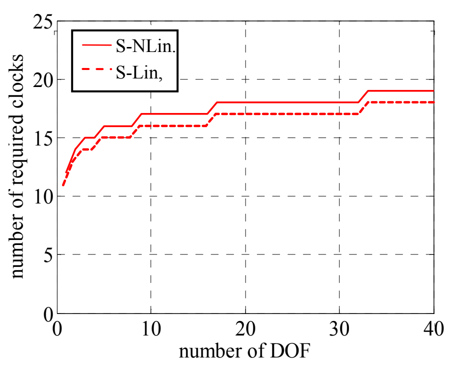
$$n_{addition} = 14N + N^2 + O(N^2) \tag{13}$$

Table 4 The number of required logic elements for arithmetic operators

$n_{LE\ op\ i}$	Adder	Subtracter	Multiplier	Divider
16 bit	298	322	251	391
32 bit	672	714	951	1451



(a) Required number of logic elements



(b) Required number of clocks

Fig. 8 Growth of requirements for Shear-type Linear and Nonlinear MDOF models

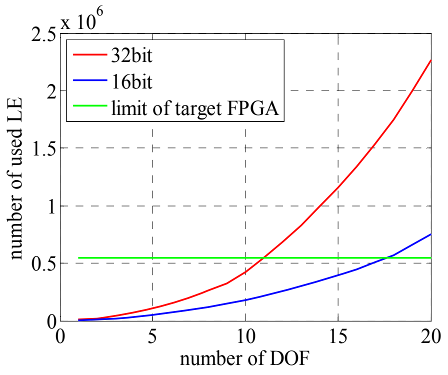
Plots of growth of required numbers of consumed logic elements, and number of clocks per time step for increasing number of DOF computed by Eqs. (10) through (13) are shown in Fig. 8. It is indicated that the difference between the Shear-type Linear MDOF model and Shear-type Nonlinear MDOF model is relatively small, partly due to the simplicity of the bilinear rule introduced as the nonlinear restoring force modelling.

For the General Linear MDOF model, in which the characteristic matrices are dense, evaluation of the required numbers of operations are

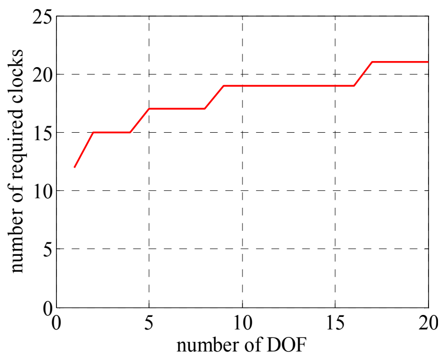
$$n_{multiplication} = 4N^2 + 5N + 1 \tag{14}$$

$$n_{addition} = 2N^2 + 5N + 2 + N^2 + O(N^2) \tag{15}$$

Plots of growth of consumed logic elements and clocks for the General Linear MDOF model is shown in Fig. 9.



(a) Required number of logic elements



(b) Required number of clocks

Fig. 9 Growth of requirements for General Linear MDOF model

Table 5 Maximum limit performance for assumed FPGA

Case	bits	Maximum No. of DOF	Processing time per step (μsec)
Shear-type Linear	32	18	0.680
MDOF	16	32	0.510
Shear-type	32	17	0.720
Nonlinear MDOF	16	30	0.545
General Linear	32	11	0.760
MDOF	16	17	0.693

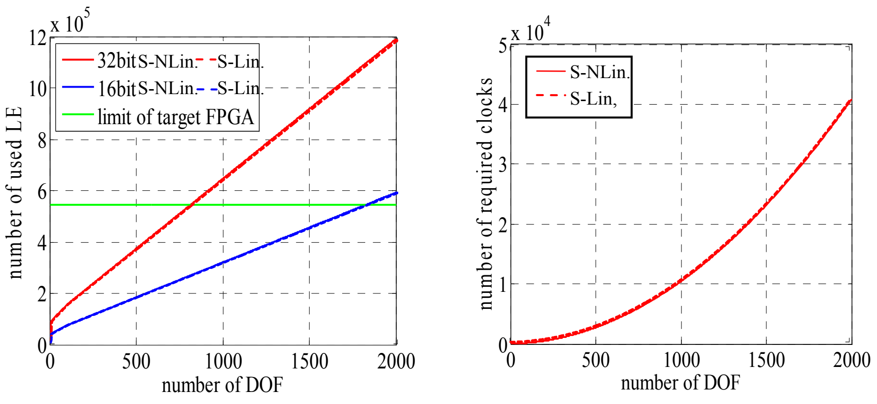
The maximum limit performance of the simulator considering the limit number of logic elements and associated limit clock frequency for the assumed FPGA is summarized in Table 5.

For all cases, while fast-speed computation with the step rate that well exceeds 1MHz is possible, it is indicated that the limit of the number of logic elements becomes the restriction in increasing the number of DOF. For this reason, the range of applicable numbers of DOF of the structural model is strictly limited.

5.3 Design case using Resource Sharing

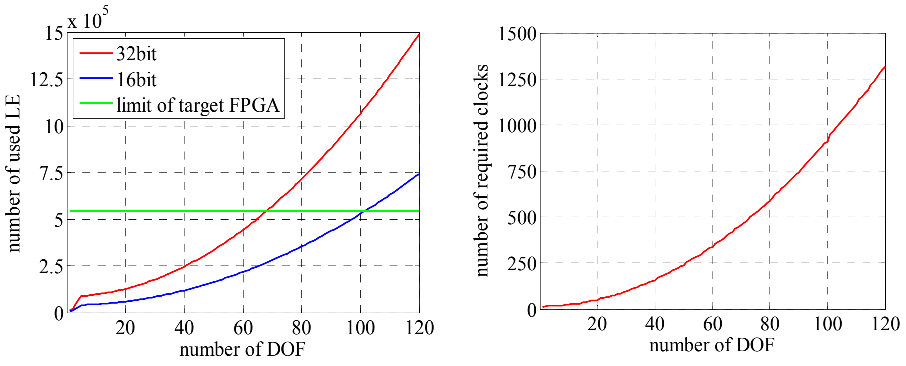
In the investigation of the previous case, the required computation time per step is found to be considerably smaller than 1ms (a typical time integration time interval for MDOF systems) when the number of DOF increases up to the performance limit. However, the fact that the required number of logic elements rapidly grows poses the limit of the applicable number of DOF. To circumvent this problem, a method called Resource Sharing (RS) is introduced. Resource sharing (Hadjis 2012) implies that the logic circuit is designed so that a resource (arithmetic operator) is reused for plural operations if there exit arithmetic operators that are not simultaneously executed in the HDL description. Application of RS is expected to reduce the required number of logic elements and to improve the performance limit of the logic circuit implemented in the FPGA. The result of the requirement growth for the case the number of adders and that of multipliers are designed to be limited to 100 is shown in Figs. 10 and 11. The maximum limit performance for the assumed FPGA with RS application is summarized in Table 6.

It is shown that although the required processing time increases with the introduction of RS, it does not reach the processing time limitation of 1ms for most of the cases, with the exception of the Shear-Type Linear and Nonlinear MDOF models with 16 bit precision. In exchange of the increase of processing time, the remarkable improvement of the limit number of DOF is achieved by the reduction of required logic elements. For the Nonlinear MDOF case with 16 bit precision, it is shown that 1725-DOF simulation is possible with the assumed FPGA, implying that real-time hybrid simulation with wider range of application than the conventional devices becomes possible with the use of FPGA implementation.



(a) Required number of logic elements (b) Required number of clocks

Fig. 10 Growth of requirements for Shear-type Linear and Nonlinear MDOF models including Resource Sharing



(a) Required number of logic elements (b) Required number of clocks

Fig. 11 Growth of requirements for General Linear MDOF model including Resource Sharing

Table 6 Maximum limit performance for assumed FPGA (Resource Sharing)

Case	bits	Maximum No. of DOF	Processing time per step (sec)
Shear-type Linear	32	816	298.9
MDOF	16	1726	999.6
Shear-type	32	815	298.5
Nonlinear MDOF	16	1725	996.9
General Linear	32	68	18.61
MDOF	16	101	31.38

6. Conclusions

In this study, application of FPGA (Field Programmable Gate Array) to a numerical simulator for the purpose of achieving fast computation of MDOF structural model in the real-time experimental hybrid simulation system is investigated. Design and performance analysis of logic circuits required for high-speed dynamic simulation of MDOF structural system models are performed, and the implementation of the designed logic circuit to an actual FPGA device is verified. Furthermore, limitation of computational performance of a single FPGA unit in terms of the speed of processing and the number of DOF of the applied structural model is investigated. The findings obtained in this investigation are summarized as follows.

- A logic circuit is designed to perform real-time dynamic response simulation executed on FPGA device, and implementation of the designed logic circuit to an actual FPGA and verification are performed. The output of the dynamic response simulator implemented to the FPGA device and usual numerical calculation result show perfect agreement within the accuracy of numerical computation, confirming that the designed logic circuit functions as designed on the actual FPGA device.
- Comparison among the shear-type linear MDOF model, general linear MDOF model (not necessarily the shear-type) and shear-type nonlinear MDOF models is made for the performance evaluation in various models with different computational efficiency. In all cases, the limit can be evaluated by the number of required logic elements to form the circuit for the designated model.
- It is shown that for the fastest processing design, required number of logic elements rapidly grows as the number of DOF increases. To mitigate this problem, the resource sharing method is used to reduce the number of required logic elements, and this methodology is shown to be effective in expanding the range of applicable structural models and computing conditions.

Development of a real-time hybrid simulation system using a test facility with the FPGA implementation presented in this paper is currently underway. The system development involving additional issues including external I/O, communication and loading system delay compensation schemes will be reported in future publication.

Acknowledgments

The research is financially supported by the Japan Society of Promotion of Science, Grant-in-Aid for Challenging Exploratory Research (Grant No.23656288). The author acknowledges the contribution of Mr. Jun-ichi Mashima (Nomura Research Institute, Ltd.) in the research presented in this paper.

References

Altera Co. (2007), *Cyclone II Device Handbook*,Vol.1.
Altera Co. (2009), *Stratix IV Device Handbook*, Vol.1.
Bensaali, F., Amira, A. and Sotudeh, R. (2007), "Floating-point matrix product on FPGA", *Proceedings of*

the 2007 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2007), Amman, Jordan, May.

Cui, W. and Pham, C.K. (2003), “Implementation of a hardware for matrix calculation on FPGA”, *IEICE Technical report*, **103**(406), 1-4.

Hadjis, S., Canis, A., Anderson, J.H., Choi, J., Nam, K. Brown, S. and Czajkowski, T. (2012), “Impact of FPGA architecture on resource sharing in high-level synthesis”, *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp.111-114, Monterey, US, February.

MathWorks (2012), “HDL Coder”, http://www.mathworks.com/products/hdl-coder/

Nakasato, N. and Hamada, T. (2006), “Acceleration of astrophysical simulations using a FPGA board (special purpose system)”, *Inform. Proc. Soc. Japan*, **47**(SIG 7), 162-173 (in Japanese).

Onouchi, M., Saito, K., Fujishima, M. and Hoh, K. (2003), “Quantum computing emulator using FPGA”, *IEICE Technical Report*, **102**(611), 67-72.

Saouma, V., Kang, D. and Haussman, G. (2012), “A computational finite-element program for hybrid simulation”, *Earthq. Eng. Struct. D.*, **41**(3), 375-389.

Saouma, V., Haussmann, G., Kang, D. and Ghannoum, W. (2013), “Real time hybrid simulation of a non ductile reinforced concrete frame”, *J. Struct. Eng. – ASCE*, DOI: 10.1061/(ASCE)ST.1943-541X.00008

Shäfer, B.C., Kingley, S.F. and Chan, A.H.C. (2002), “Implementation of the discrete element method using reconfigurable computing (FPGAS)”, *Proceedings of the 15th ASCE Engineering Mechanics Conference (EM2002)*, New York, USA, June.

Shao, X., Reinhorn, A. and Sivaselvan, M. (2011), “Real-time hybrid simulation using shake tables and dynamic actuators”, *J. Struct. Eng. – ASCE*, **137**(7), 748-760.

Appendix A. Verilog-HDL code for adder

In the following code, e and s are the constants for exponent bit width and mantissa bit width, respectively.

```
module  adder(a,b,ans);
   input  [e+s:00] a,b ;
   output [e+s:00] ans ;
   parameter  zero   = {e'h00,s'h00} ;
   parameter  error  = {e'hff,s'hff} ;
// step0 Absolute value comparison and input re-assignment
   wire       comp = ( a[e+s-1:00] > b[e+s-1:00] ) ;
   wire [e+s:00]  fa   = (comp)? a:b ;
   wire [e+s:00]  fb   = (comp)? b:a ;
// step1 Exception handling signal
   wire sign_xor   = fa[e+s] ^ fb[e+s] ;
   wire sign_1     = fa[e+s] ;
   wire equal      = ( fa[e+s-1:00] == fb[e+s-1:00] ) ;
   wire expo_a00   = ( fa[e+s-1:s] == e'h00 ) ;
   wire expo_b00   = ( fb[e+s-1:s] == e'h00 ) ;
// step2 Shift
   wire [e-1:00]   expo_fa      = fa[ e+s-1 : s ] ;
   wire [e-1:00]   expo_dif     = fa[e+s-1:s] - fb[e+s-1:s] ;
   wire            expo_u_sp3   = ( expo_dif >= e'd(s+3) ) ;
   wire [2s+3:00]  fb_shift     = (expo_u_sp3)? { s+3'h00 , !exp_b00 , fb[s-1:00] } :
                                  ( { !expo_b00 , fb[s-1:00] , s+3'h00 } >>
expo_dif ) ;
   wire            fb_low_or    = |fb_shift[s:00] ;
   wire [s+4:00]   sgnf_fb_t    = { 1'b0 , fb_shift[2s+3:s+1] , fb_low_or } ;
   wire [s+4:00]   sgnf_fa      = { 1'b0 , !expo_a00 , fa[s-1:00] , 3'h0 } ;
// step3 Addition
   wire [s+4:00]   sgnf_fb  = (sign_xor)? ~sgnf_fbt + 1'b1 : sgnf_fb_t ;
   wire [s+4:00]   sgnf_3   = sgnf_fa + sgnf_fb ;
// step4 Normalization
   wire [04:00]    expo_shift_t ;
   assign          expo_shift_t[04] = ~|sgnf_3[s+4:s-11] ;
   wire [s+4:00]   sgnf_shift0    = ( !expo_shift_t[04] )? sgnf_3[s+4:00] :
                                  { sgnf_3[s-12:00] , 16'h0000 } ;
   assign          expo_shift_t[03] = ~|sgnf_shift0[s+4:s-3] ;
   wire [s+4:00]   sgnf_shift1    = ( !expo_shift_t[03] )? sgnf_shift0[s+4:00] :
                                  { sgnf_shift0[s-4:00] , 8'h00 } ;
   assign          expo_shift_t[02] = ~|sgnf_shift1[s+4:s+1] ;
   wire [s+4:00]   sgnf_shift2    = ( !expo_shift_t[02] )? sgnf_shift1[s+4:00] :
                                  { sgnf_shift1[s:00] , 4'h0 } ;
   assign          expo_shift_t[01] = ~|sgnf_shift2[s+4:s+3] ;
   wire [s+4:00]   sgnf_shift3    = ( !expo_shift_t[01] )? sgnf_shift2[s+4:00] :
                                  { snf_shift2[s+2:00] , 2'h0 } ;
   assign          expo_shift_t[00] = ~|sgnf_shift3[s+4] ;
   wire [s+4:00]   sgnf_shift4    = ( !expo_shift_t[00] )? sgnf_shift3[s+4:00] :
                                  { sgnf_shift3[s+3:00] , 1'h0 } ;

   wire [e+1:00]   expo_4         = expo_fa - expo_shift_t + 1'b1 ;
// step5 Carry-over bit
   wire  up0 = sgnf_shift4[04] ;
   wire  up1 = sgnf_shift4[03] ;
   wire  up2 = sgnf_shift4[02] ;
   wire  up3 = sgnf_shift4[01] ;
   wire  up4 = sgnf_shift4[00] ;
   wire  all_1    = &sgnf_shift4[s+4:04] ;
   wire  carry_b  = up1 & ( up0 | up2 | up3 | up4 ) ;
   wire [e+1:00] expo_5  = expo_4 + ( carry_b & all_1) ;
   wire [s:00]  sgnf_5  = sgnf_shift4[s+4:04] + carry_b ;
// step6 Overflow/underflow adjustment
   wire [e-1:00]        expo_6    =   (exp_5[e+1])? e'h00 :
                        (exp_5[e])? e'hff : expo_5[e-1:00] ;
   wire [s-1:00]        sgnf_6    =   (exp_5[e+1])? s'h00 :
                        (exp_5[e])? s'h00 : sgnf_5[s-1:00];
// step7 Exception handling
   function [e+s:00] fin ;
      input  sign_xor , equal ;
      input  expo_a00,expo_b00;
      input  sign_1;
      input [e+s:00] fa,fb;
      input [e-1:00] expo_6 ;
      input [s-1:00] sgnf_6 ;
      casex ( { sign_xor , equal , expo_a00 , expo_b00 } )
         4'bxx_01     : fin = fa;
         4'bxx_10     : fin = fb;
         4'bxx_11     : fin = {1'b0,zero};
         4'b00_00     : fin = {sign_1,expo_6,sgnf_6};
         4'b01_00     : fin = {sign_1,expo_6,sgnf_6};
         4'b10_00     : fin = {sign_1,expo_6,sgnf_6};
         4'b11_00     : fin = {1'b0,zero};
         default :     fin = {1'b1,error} ;
      endcase
   endfunction
   wire         sign_7 ;
   wire [e-1:00]  expo_7 ;
   wire [s-1:00]  sgnf_7 ;
   assign { sign_7,expo_7,sgnf_7 } = fin (sign_xor , equal , expo_a00,expo_b00
                                          sign_1,fa,fb,expo_6,sgnf_6  ) ;
// Output
   wire         sign_fin = sign_7 ;
   wire [e-1:00] expo_fin = expo_7 ;
   wire [s-1:00] sgnf_fin = sgnf_7 ;
   assign       ans = { sign_fin , expo_fin , sgnf_fin } ;
endmodule
```

Appendix B. Verilog-HDL code for multiplier

```
module multiplier(a,b,ans);
   input [e+s:00] a,b ;
   output [e+s:00] ans ;
   parameter geta = 2e-1 -1;
   parameter zero = {e'h00,s'h00};
   parameter error = {e'hff,s'hff};
// step0 Exception handling signal
   wire expo_a00 = (a[s+e-1:s] == e'h00);
   wire expo_b00 = (b[s+e-1:s] == e'h00);
// step1 multiplication
   wire sign_fin = a[s+e] ^ b[s+e];
   wire [e+1:00] expo_1 = a[s+e-1:s] + b[s+e-1:s] - geta;
   wire [2s+1:00] sgnf_1 = {1'b1,a[s-1:00]} * {1'b1,b[s-1:00]};
// step2 normalization
   wire [e+1:00] expo_2 = expo_1 + sgnf_1[2s+1] ;
   wire [2s+1:s-2] sgnf_2_t = {sgnf_1[2s+1:s-1],(|sgnf_1[s-2:00])};
   wire [2s:s-3] sgnf_2 =
               ( sgnf_1[2s+1] )?sgnf_2_t[2s+1:s-2] :{ sgnf_2_t[2s:s-2],1'b0 } ;
// step3 bit shift
   wire ulp_bit = sgnf_2[s] ;
   wire guard_b = sgnf_2[s-1] ;
   wire round_b = sgnf_2[s-2] ;
   wire stiky_b = sgnf_2[s-3] ;
   wire all_1 = &sgnf_2t[2s:s+1] ;
   wire carry_b = guard_b & ( ulp_bit | round_b | stiky_b ) ;
   wire [e+1:00] expo_3 = expo_2 + ( carry_b & all_1) ;
   wire [s:00] sgnf_3 = sgnf_2[2s:s] + carry_b ;
// step4 correction for overflow/underflow
   wire [e-1:00] expo_4 = ( expo_3[e+1] )? e'h00 :
                                        ( expo_3[e] )? e'hff : expo_3[e-1:00] ;
   wire [s-1:00] sgnf_4 = ( expo_3[e+1] )? s'h00 :
                                         ( expo_3[e] )? s'h00 : sgnf_3[s-1:00];
// step5 exception handling
   function [e+s-1:00] fin ;
      input expo_a00 , expo_b00 ;
      input [e-1:00] expo_4 ;
      input [s-1:00] sgnf_4 ;
      case ( { expo_a00 , expo_b00 } )
         2'b11 : fin = zero ;
         2'b10 : fin = zero ;
         2'b01 : fin = zero ;
         2'b00 : fin = { expo_4,sgnf_4 } ;
         default : fin = error ;
      endcase
   endfunction
   wire [e-1:00] expo_5 ;
   wire [s-1:00] sgnf_5 ;
   assign { expo_5,sgnf_5 } = fin ( expo_a00,expo_b00,expo_4,sgnf_4 ) ;
// Output
   wire [e-1:00] expo_fin = expo_5 ;
   wire [s-1:00] sgnf_fin = sgnf_5 ;
   assign ans = { sign_fin , expo_fin , sgnf_fin } ;
endmodule
```