Divider Testing With Parallel Programs

Rainer Karl Trummer

Department of Scientific Computing
University of Salzburg

November 26, 2004

Abstract

This paper describes a project in the field of parallel computation, carried-out in addition to writing a master's thesis about improved hardware dividers. Verifying new dividers usually involves exhaustive long-term tests, especially if the underlying algorithm is data-dependent and an average value of required clock cycles is sought. To reduce the duration of such tests, the algorithm can be implemented in a high-level programming language that supports parallel computation. Hence, executing the program on several processors in parallel causes a significant reduction of the test duration.

1 Introduction

Concerning execution time, most of sophisticated division algorithms, like the well known SRT Division, are data-dependent [1]. Therefore, when it comes to testing a new algorithm, we are primarily interested in two characteristic values, (1) the number of clock cycles required to handel the worst case, and (2) the number of clock cycles needed on the average. The worst case can usually be determined easily, whereas determining an accurate value for the average can sometimes be much more complicated, especially if the underlying algorithm is highly data-dependent. For instance, if we want to determine the exact average value of a 32-bit division algorithm that operates on *unsigned* operands, strictly speaking, we have to test all possible combinations of an unsigned 32-bit dividend with an unsigned 32-bit divisor. Even if we restrict the test set to pairs of operands that perform *valid integer divisions* only, we obtain the following number of combinations to be tested.

Definition 1. Let a denote the dividend and let b denote the divisor. The number of *valid integer divisions*, such that $a \geq b \wedge b \neq 0$, over the 32-bit range is given by:

$$\sum_{a=1}^{2^{32}-1} \sum_{b=1}^{a} = \sum_{a=1}^{2^{32}-1} a = \frac{\left(2^{32}-1\right) 2^{32}}{2} = \left(2^{32}-1\right) 2^{31}$$

$$= 4294967295 \times 2147483648 = 9223372034707292160$$

Now assume we have created a Verilog HDL (Hardware Description Language) design for the division algorithm and load the compiled image into the FPGA (Field-Programable Gate Array) of some testing board, for instance one of the Xilinx Spartan series [2]. We further assume that the FPGA requires approximately 0.1 μs per division. Thus, the overall computation time would be:

$$\begin{aligned} 9223372034707292160 \times 10^{-7} &\approx 922337203471 \text{ sec} \\ &\approx 256204779 \text{ hours} \\ &\approx 10675199 \text{ days} \\ &\approx \textbf{29247 years} \end{aligned}$$

Obviously, it is not possible to test the complete set in a meaningful amount of time—at least not on a single machine—due to current hardware restrictions.

Unfortunately, we are faced with even more constraints that effect the required computation time. (1) if we run the algorithm on an autonomous board, we have absolutely no opportunity to apply any parallelism to the test routines, i.e., as long as we do not have several boards, we cannot share the amount of computational work among several machines. (2) and not minor important, we also want to determine the average performance for different *word lengths*, precisely, for 16, 32, 64, and 128 bits.

According to these constraints, the goal is to develop a program for parallel computation, which involves as many machines as available. In this sense, we are still not able to test the complete set—unless we do not distribute the program over many thousand machines—however, feeding the algorithm with a large number of randomly chosen operands normally suffice for obtaining a good approximate average value. The resulting parallel program must also be capable to process all of the four desired word lengths, thus, we need to create *multi-word* algorithms. This challenge further implies the development of a random number generator that provides multi-word random numbers.

2 Solution Method

Due to the primary requirement of speed optimization, the high-level programming language C can be considered as an excellent choice for this purpose. Although there exists a version of C called *SystemC* designed for parallel programming, the most convenient way for extending a C program to support parallelism is simply to invoke the MPI (Message Passing Interface) library. This library represents an IEEE standard for interprocess communication, provides a great variety of communication functions, supports the high-level languages C/C++ and Fortran, and is available for all prevalent platforms [3].

Referring to the introduced problem, the solution consists of four different issues: (1) mapping the divider algorithms to functions that do not perform any redundant computation, i.e., given two arbitrary operands, they only need to determine the number of required clock cycles, (2) development of an abstract data type that supports multi-word operations, (3) development of a multi-word random number generator, (4) development of a parallel program based on MPI.

2.1 Performance Functions

There are four Verilog HDL divider *modules*, i.e., descriptions about connectivity and behavior of the underlying hardware, we would like to determine their average performances. Each of them is data-dependent and belongs to the category of so-called "subtract-and-shift" dividers that perform *sequential division* [4]. Depending on fundamental differences in their ways of processing, implementing the underlying algorithms efficiently in software is one of the most challenging parts of this project.

The first divider is a simple *Radix-Two Divider*, modified to initially shift left the remainder-quotient register until the remainder becomes non-zero, i.e., until the first 1-bit of the dividend, initially loaded into the low-order half of the remainder-quotient register, appears in the high-order half. This divider is actually the slowest of the four and primarily used for performance comparisons. Nevertheless, a function returning the number of required clock cycles can be implemented very efficiently, since all we need to know is the *number of leading zeros* of the given dividend. This information suffice to compute the number of initial left-shifts, which also implies the number of remaining steps.

The second one is a so-called *Self-Aligning Divider*, which initially shifts left the divisor until its MSB (Most Significant Bit) matches the MSB of the dividend. This initial alignment causes a reduction of the dividend by a magnitude representing the divisor multiplied by a power of two, rather than subtracting the original divisor. The algorithm then shifts back the aligned divisor until it reaches its original position. Due to this technique, mapping the divider's behavior to a function returning the correct number of required clock cycles is much more complex, since it depends on several stopping criteria that must be taken into account.

The third one, called *Direct-Aligning Divider*, can be viewed as a more sophisticated version of the previous divider. It replaces many aligning steps by a two-step logical shift operation, i.e., it *directly* generates a correct aligned divisor within two steps. Since the number of needed steps for aligning is constant for this divider, the total number of required clock cycles merely depends on the *bit difference* of the given operands. Except for a few stopping criteria that must be handled separately, the behavior of this divider can be mapped easily to a performance function.

The fourth and most complex divider is called *Hybrid-Aligning Divider*, a denotation that will become clear soon. Although the foregoing Direct-Aligning Divider stands-out from all others concerning its

constant number of aligning steps, this feature also causes a considerable drawback. More precisely, in the worst case, whenever a dividend with most of its bits turned on is divided by a very small divisor, the number of required clock cycles is even worse compared to the Self-Aligning Divider. This circumstance led to the development of the Hybrid-Aligning Divider, which combines the constant aligning-step property with an improved worst case processing. Although this divider is the most complex of the four, mapping its behavior to a performance function can be realized without any difficulties.

Verification of the four performance functions happens through comparisons with the original Verilog dividers. The Verilog design of each divider is tested by running a so-called *test bench*, which allows a runtime analysis of simulated hardware by applying arbitrary stimuli to the tested component. Furthermore, each stimuli, as well as each input/output value, can be written directly to a file. In this sense, we can create individual sequences of operands and simply store the numbers of required clock cycles in destined output files. These files are then used as input for comparisons with the values obtained from the corresponding performance functions, which we let process the same sequences of operands.

2.2 Multi-Word Class

As already mentioned in the introduction, the goal is to determine each divider's average performance for different word lengths, namely, 16, 32, 64, and 128 bits, and of course, by running the parallel program on a cluster of, say, 16 nodes, we hope it will gather a very large amount of data. Most of operating systems, like Windows and Linux, provide a built-in 64-bit data type, however, for longer word lengths exists hardly any support directly by the system. So we need to create our own data type for handling word lengths that exceed the boundary of 64 bits. For this reason, the extension from C to the object-oriented version C++ is exactly what we need for defining an *abstract* 128-bit data type. This task can be managed easily by creating a new *class* for the desired type. Within this class we have the freedom to define specific *constructors*, overload *operators*, and develop arbitrary *methods* for modifying data *members*.

Depending on current computer architectures, CPU registers for *integral* types are mostly either 32-bit or 64-bit wide. Therefore, class `int128` is designed as platform-independent data type that holds an array consisting of either four or two `unsigned long int` elements, which usually represent the full size of an integral-type register.

2.3 Multi-Word Random Number Generator

Designing the multi-word random number generator is the most critical part of this project. That is, because the "quality" of the data used to determine the desired average values depends primarily on the randomly generated operands. Concerning the underlying algorithm, there are three major requirements that must be met: (1) we must ensure *reproducibility*, (2) we need a sufficiently large *distribution*, and thus, (3) the *period* of the generated sequence must be as long as possible.

Rather than designing a random number generator right from scratch, we make use of the portable system function `rand()` and incorporate it in a suitable way. In case of 32-bit architecture, it returns a value in the range $0, \ldots, 2^{15}–1$ and has a period of 2^{31}, i.e., the sequence repeats after 2147483648 numbers. Hence, assuming a good statistical distribution, each of the 32768 possible values will appear approximately 65536 times per period. Since we need two operands for performing a division, this circumstance of repetition actually does not matter, unless there are not too many pairs that repeat frequently. Although a period of 2^{31} might be sufficient for most of non-critical applications, e.g., screen savers, however, for our purpose we need a much longer period. In case of 64-bit architecture, function `rand()` returns a value in the range $0, \ldots, 2^{31}-1$ and has a period of 2^{63}, which is basically long enough.

Reproducibility is guaranteed through "seeding" the local random number generator of each involved process by calling the system function `srand()` with the *rank* of the process as argument. This also minimizes possible overlaps of partial sequences if the complete sequence is sufficiently long. Note that seeding the generator does not change the original sequence, it only moves the starting point to some arbitrary position. To reduce the probability of overlaps, an extension of the `rand()` sequence is achieved by applying some additional "information" to the generated random value, i.e., the old value is kept and combined with the new one using an XOR operator. Due to this technique, a given sequence of length $2^{31}, \ldots, 2^{63}$ normally extends to last for many years until it repeats for the first time [5].

We first give a short, formal definition and thereafter present the two algorithms designed for the 32-bit and 64-bit platform, respectively. (The used symbols are as follows: $\oplus$ denotes the XOR operator, $\gg$ the Shift-Right operator, and $\circ$ the AND operator.)

Definition 2. Let S denote a *sequence*, where $s_i \in \{0,\ldots,2^k-1\}$ $(i,k \in \mathbb{N})$, let π denote a *period*, where $\pi_S = |S|$, and let γ denote a function that maps S to γ_n $(n \in \mathbb{N})$.

Algorithm 1 (32 Bit). $S = (s_1,\ldots,s_{\pi_S})$, $s_i \in \{0,\ldots,2^{15}-1\}$, $\pi_S = 2^{31}$,

$$\begin{aligned}\gamma : \{0,\ldots,2^{16k}-1\} &\rightarrow \{0,\ldots,2^{16k}-1\}\\ (t_0,\ldots,t_{k-1}) &\mapsto s_i,\ldots,s_{i+k-1}\\ (x_0,\ldots,x_{k-1}) &\mapsto (\boldsymbol{x}\oplus\boldsymbol{t}) \ggg (s_{i+k}\circ(16k-1)),\end{aligned}$$

where $k \in \{1,2,4,8\}$ denotes the number of 16-bit half-words,

$$\Rightarrow \pi_\gamma \in \{\pi_S,\ldots,2^{16k}\pi_S\} = \{2^{31},\ldots,2^{16k+31}\}.$$

Algorithm 2 (64 Bit). $S = (s_1,\ldots,s_{\pi_S})$, $s_i \in \{0,\ldots,2^{31}-1\}$, $\pi_S = 2^{63}$,

$$\begin{aligned}\gamma : \{0,\ldots,2^{16k}-1\} &\rightarrow \{0,\ldots,2^{16k}-1\}\\ (t_0,\ldots,t_{k-1}) &\mapsto s_i,\ldots,s_{i+\lceil k/2\rceil-1}\\ (x_0,\ldots,x_{k-1}) &\mapsto (\boldsymbol{x}\oplus\boldsymbol{t}) \ggg (s_{i+\lceil k/2\rceil}\circ(16k-1)),\end{aligned}$$

where $k \in \{1,2,4,8\}$ denotes the number of 16-bit half-words,

$$\Rightarrow \pi_\gamma \in \{\pi_S,\ldots,2^{16k}\pi_S\} = \{2^{63},\ldots,2^{16k+63}\}.$$

2.4 MPI Parallel Program

The program is designed to share computation among an arbitrary number of processes, preferably as many as possible. It accepts two specific arguments—in addition to individual MPI arguments—for specifying the duration of a test and an interval for reporting intermediate results, both inputs interpreted as minutes. A *structure* consisting of an `unsigned int64` array for holding the computation data is used by all processes for collecting and transferring their results. In case of the *root process*, this data structure is allocated additionally as often as many processes are involved currently, needed for calculating the average values. After entering the main loop, all processes are synchronized through an `MPI_Barrier` message before receiving the broadcasted interval value via `MPI_Bcast` from the root process. This value is either zero or the specified length, depending on the remaining duration. In case of zero, all processes are forced to leave the main loop and return, otherwise, they enter the time-triggered loop. Within this loop, the two operands a and b are assigned random numbers that are additionally incremented by one to avoid zero-operands. If $a \geq b$ holds, all four performance functions are called one after the other, in case of $a < b$, they are called with exchanged arguments a and b. The returned *numbers of required clock cycles*, as well as the *division count*, are summed up and stored into the local data structure. Upon completion of an interval, each process establishes a connection to the root process through an `MPI_Sendrecv` message to ensure connectivity before transferring its actual data using `MPI_Send`. The root process receives each of these data packages via `MPI_Recv` and writes it to the corresponding, pre-allocated data structure. In addition to the current subtotals, the four average values are calculated and written to the report file. Lastly, the duration value is decremented by the interval value. If the remaining duration is less than the interval, then the interval value is set to zero, which causes the program to finish properly with the subsequent broadcast message (for more details see Appendix A).

3 Obtained Results

All of the following results were obtained by running the parallel program on a 16-node cluster, consisting of 64-bit dual-processor boards, at the Jozef Stefan Institute in Ljubljana, Slovenia. Since the cluster was not fully setup when the tests were performed, unfortunately, only 10 of the 32 processors could be accessed. Nevertheless, after executing each of the four tests for several hours, absolutely satisfying results were achieved. The most essential results are listed in Table 1 (the complete reports containing all detailed data and time stamps can be found in Appendix B). They perfectly meet all of the expectations, since

SPECIFICATION		AVERAGE NUMBER OF CLOCK CYCLES			
WORD LENGTH	PERFORMED DIVISIONS	RADIX-TWO	SELF-ALIG.	DIRECT-A.	HYBRID-A.
16	196015727400	36	25	10	8
32	111087201570	73	49	22	16
64	89632936721	148	97	48	33
128	3491168623	297	193	100	68

Table 1: Results of the four long-term tests performed on the 16-node cluster.

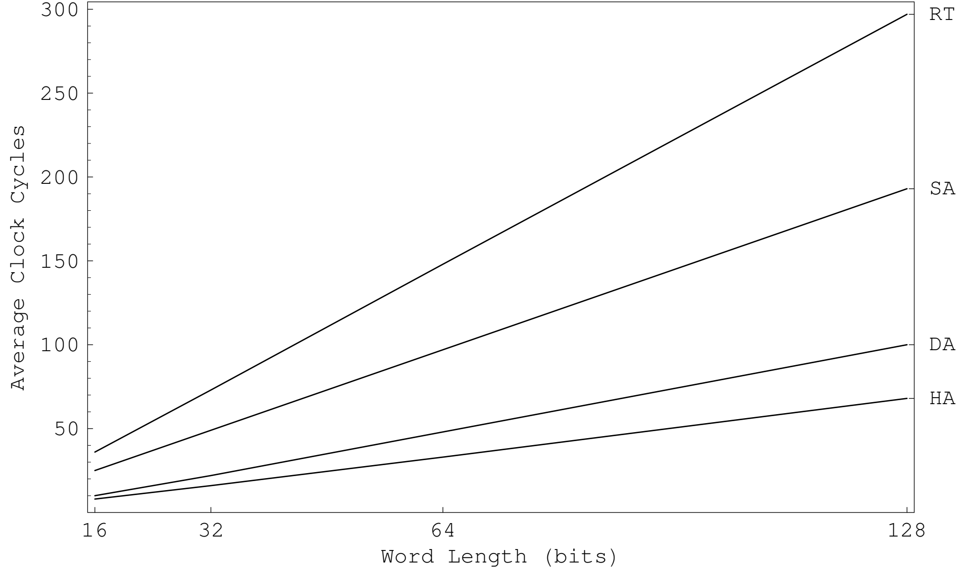

Figure 1: Average performance of the four dividers.

the determined average values are not very much higher than the values obtained previously by verifying each divider module with the Xilinx XE II 5.7g model simulator.

Figure 1 shows a graphical interpretation of the values listed in Table 1. It points out that the average number of required clock cycles seems to increase absolutely linearly for all four dividers. Due to this circumstance, we can derive approximate expressions that indicate their average performances. One possibility we will use here is to calculate the average *gradient* of each characteristic line.

Definition 3. Let $W = \{w_1, \ldots, w_n\}$ $(n \in \mathbb{N})$ denote a set of *word lengths* and let $V = \{v_1, \ldots, v_n\}$ denote a set of corresponding *values*, then the average *gradient* is given by:

$$G = \binom{n}{n-2}^{-1} \sum_{i=2}^{n} \sum_{k=1}^{i-1} \frac{v_i - v_k}{w_i - w_k}$$

Inserting the specific data in the given formula yields the following gradients:

$$G_{\mathrm{RT}} = \tfrac{1}{6}\left(\tfrac{37}{16} + \tfrac{112}{48} + \tfrac{75}{32} + \tfrac{261}{112} + \tfrac{224}{96} + \tfrac{149}{64}\right) = 2.3286$$

$$G_{\mathrm{SA}} = \tfrac{1}{6}\left(\tfrac{24}{16} + \tfrac{72}{48} + \tfrac{48}{32} + \tfrac{168}{112} + \tfrac{144}{96} + \tfrac{96}{64}\right) = 1.5$$

$$G_{\mathrm{DA}} = \tfrac{1}{6}\left(\tfrac{12}{16} + \tfrac{38}{48} + \tfrac{26}{32} + \tfrac{90}{112} + \tfrac{78}{96} + \tfrac{52}{64}\right) = 0.7971$$

$$G_{\mathrm{HA}} = \tfrac{1}{6}\left(\tfrac{8}{16} + \tfrac{25}{48} + \tfrac{17}{32} + \tfrac{60}{112} + \tfrac{52}{96} + \tfrac{35}{64}\right) = 0.5294$$

The four dividers are fully verified to work properly at a clock rate of 200 MHz, which corresponds to a clock period of 5 ns. Hence, if we multiply the determined gradients by this period, we obtain the formulas listed in Table 2 below, where Δ_w denotes the average execution time in nano-seconds and w denotes the word length in bits.

DIVIDER TYPE	AVERAGE EXECUTION TIME
radix-two	$\Delta_w = 11.7\,w$
self-aligning	$\Delta_w = 7.5\,w$
direct-aligning	$\Delta_w = 4.0\,w$
hybrid-aligning	$\Delta_w = 2.7\,w$

Table 2: Performance formulas.

4 Conclusions

We have seen that mapping complex algorithms to performance functions, which can be processed by parallel programs, is a suitable way for obtaining satisfying and also plausible results, whenever conventional methods are not applicable. As one possible solution to the posed problem, we have introduced the MPI methodology as an appropriate tool for developing such programs, which are able to distribute computation over many machines. In this context, we have also experienced that invoking the MPI library provides a very convenient way of extending an existing program to a parallel program.

A MPI-Based Source Code

A.1 File global.h

```
/*****************************************************************************/
/*                                                                           */
/*     Project: MPI Program For Testing All Xilinx Dividers                  */
/*   Copyright: Copyright (c) 2004. All Rights Reserved                      */
/*    Compiler: Microsoft Visual C++ .NET - Target Win32                     */
/*     Company: Dept of ComSci, University of Salzburg                       */
/*      Author: Rainer Trummer <rtrummer@cosy.sbg.ac.at>                     */
/*        Date: October 28, 2004                                             */
/*                                                                           */
/*****************************************************************************/

#ifndef GLOBAL_H
#define GLOBAL_H

#include <stdio.h>
#include <stdlib.h>
#include "int128.h"

// Word length associated with dividers
//
#define WORD_LEN 128

// CPU size associated with data types
//
#if RAND_MAX > 32767
  #define CPU_SIZE 64
#else
  #define CPU_SIZE 32
#endif

// Control of verification routines
//
//#define VERIFY
//#define RANDOM_FUNCTION
//#define RADIX_TWO
//#define SELF_ALIGNING
//#define DIRECT_ALIGNING
//#define HYBRID_ALIGNING

typedef unsigned char  byte;
typedef unsigned short hword;
typedef unsigned int   uint;
typedef unsigned long  word;

#ifndef _WIN32
  typedef unsigned long long dword;
#else
  typedef unsigned __int64 dword;
#endif

#if   WORD_LEN == 16
  typedef hword  intx;

#elif WORD_LEN == 32
  typedef word   intx;

#elif WORD_LEN == 64
  typedef dword  intx;

#elif WORD_LEN == 128
  typedef int128 intx;
#endif

extern int    verification    ( void );
extern intx   randomize       ( void );
extern int    radix_two       ( intx, intx );
extern int    self_aligning   ( intx, intx );
```

```
extern int    direct_aligning ( intx, intx );
extern int    hybrid_aligning ( intx, intx );
extern char * to_str          ( dword );

#endif // !GLOBAL_H
```

A.2 File main.cpp

```
/****************************************************************************/
/*                                                                          */
/*     Project: MPI Program For Testing All Xilinx Dividers                 */
/*   Copyright: Copyright (c) 2004. All Rights Reserved                     */
/*    Compiler: Microsoft Visual C++ .NET - Target Win32                    */
/*     Company: Dept of ComSci, University of Salzburg                      */
/*      Author: Rainer Trummer <rtrummer@cosy.sbg.ac.at>                    */
/*        Date: October 28, 2004                                            */
/*                                                                          */
/****************************************************************************/

#include "global.h"

#ifndef VERIFY

#include <string.h>
#include <time.h>
#include <mpi.h>

#ifdef _WIN32
  #define SEPARATOR '\\'
#else
  #define SEPARATOR '/'
#endif

const int NUM_VALUES = 5;

struct TASK_DATA
{
    dword value[NUM_VALUES];

    void clear( void )
    {
        for( int i = 0; i < NUM_VALUES; ++i )
        {
            value[i] = 0;
        }
    }
};

#endif // !VERIFY


int main( int argc, char **argv )
{
#ifndef VERIFY

    const char *VALUE_NAME[] = {"RT", "SA", "DA", "HA", "DC"};
    char       *p, fname[256];
    int        size, rank, i, k;
    intx       a, b;
    time_t     duration, interval, stop;
    FILE       *file;
    TASK_DATA  *data;
    MPI_Status status;

    if( MPI_Init( &argc, &argv ) != MPI_SUCCESS )
    {
        printf( "\nMPI initialization failed!\n\n" );
        return( 1 );
    }
```

```
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

if( rank == 0 )
{
    if( argc < 2 )
    {
        printf( "\nNo duration (min) specified!\n\n" );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    if( argc < 3 )
    {
        printf( "\nNo interval (min) specified!\n\n" );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    sscanf( argv[1], "%lu", &duration );
    sscanf( argv[2], "%lu", &interval );

    if( duration < interval )
    {
        printf( "\nDuration is less than interval!\n\n" );
        MPI_Abort( MPI_COMM_WORLD, 1 );
    }

    duration *= 60;
    interval *= 60;
}

if( (data = new TASK_DATA[rank ? 1 : size + 1]) == NULL )
{
    printf( "\nMemory allocation failed!\n\n" );
    MPI_Abort( MPI_COMM_WORLD, 1 );
}

data->clear( );
srand( rank );

if( rank == 0 )
{
    strcpy( fname, argv[0] );

    if( (p = strrchr( fname, SEPARATOR )) == NULL )
    {
        p = fname;
    }
    else
    {
        ++p;
    }

    sprintf( p, "report_%d.txt", WORD_LEN );

    if( (file = fopen( fname, "wt" )) == NULL )
    {
        file = stdout;
    }

    time( &stop );
    fprintf( file, "\n*** %d-Bit Results ***\n\n%s", WORD_LEN, ctime( &stop ) );
    fclose( file );
}

for( ; ; )
{
    MPI_Barrier( MPI_COMM_WORLD );
    MPI_Bcast( &interval, 1, MPI_UNSIGNED_LONG, 0, MPI_COMM_WORLD );

    if( !interval )
    {
```

```
        break;
    }

    time( &stop );
    stop += interval;

    do
    {
        a = randomize( );
        b = randomize( );

        if( ++a < ++b ) // additional increment to avoid value 0
        {
            data->value[0] +=       radix_two( b, a );
            data->value[1] +=   self_aligning( b, a );
            data->value[2] += direct_aligning( b, a );
            data->value[3] += hybrid_aligning( b, a );
        }
        else
        {
            data->value[0] +=       radix_two( a, b );
            data->value[1] +=   self_aligning( a, b );
            data->value[2] += direct_aligning( a, b );
            data->value[3] += hybrid_aligning( a, b );
        }

        ++(data->value[4]);
    }
    while( time( NULL ) < stop );

    if( rank )
    {
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_BYTE, 0, NUM_VALUES,
                      MPI_BOTTOM, 0, MPI_BYTE, 0, NUM_VALUES, MPI_COMM_WORLD, &status );

        MPI_Send( data->value, NUM_VALUES, MPI_LONG_LONG_INT, 0, rank, MPI_COMM_WORLD );
    }
    else
    {
        for( k = 1; k < size; ++k )
        {
            MPI_Sendrecv( MPI_BOTTOM, 0, MPI_BYTE, k, NUM_VALUES,
                          MPI_BOTTOM, 0, MPI_BYTE, k, NUM_VALUES, MPI_COMM_WORLD, &status );

            MPI_Recv( data[k].value, NUM_VALUES, MPI_LONG_LONG_INT, k, k, MPI_COMM_WORLD, &status );
        }

        data[size].clear( );

        for( k = 0; k < size; ++k )
        {
            for( i = 0; i < NUM_VALUES; ++i )
            {
                data[size].value[i] += data[k].value[i];
            }
        }

        if( (file = fopen( fname, "at" )) == NULL )
        {
            file = stdout;
        }

        fprintf( file, "\n" );

        for( i = 0; i < NUM_VALUES; ++i )
        {
#if CPU_SIZE == 32
            fprintf( file, "Sum %s: %s\n", VALUE_NAME[i], to_str( data[size].value[i] ) );
#else
            fprintf( file, "Sum %s: %lu\n", VALUE_NAME[i], data[size].value[i] );
#endif
```

```
        }

        k = NUM_VALUES - 1;
        fprintf( file, "\n" );

#if CPU_SIZE == 32
        printf( "\nDivisions: %s\n\n", to_str( data[size].value[k] ) );
#else
        printf( "\nDivisions: %lu\n\n", data[size].value[k] );
#endif

        for( i = 0; i < k; ++i )
        {
            data[size].value[i] /= data[size].value[k];
#if CPU_SIZE == 32
            fprintf( file, "Average %s: %s\n", VALUE_NAME[i], to_str( data[size].value[i] ) );
            printf( "Average %s: %s\n", VALUE_NAME[i], to_str( data[size].value[i] ) );
#else
            fprintf( file, "Average %s: %lu\n", VALUE_NAME[i], data[size].value[i] );
            printf( "Average %s: %lu\n", VALUE_NAME[i], data[size].value[i] );
#endif
        }

        fprintf( file, "\n%s", ctime( &stop ) );
        fclose( file );

        if( (duration -= interval) < interval )
        {
            interval = 0;
        }
    }
}

MPI_Finalize( );

delete [] data;
return( 0 );

#else // !VERIFY

    return verification( );

#endif // !VERIFY
}
```

B Program Reports

B.1 File report_16.txt

```
Tue Nov 23 20:51:56 2004

Sum RT: 1382728216111
Sum SA: 957197321114
Sum DA: 392343909934
Sum HA: 321066317797
Sum DC: 37678798524

Average RT: 36
Average SA: 25
Average DA: 10
Average HA: 8

Tue Nov 23 21:51:56 2004

Sum RT: 2844333006192
Sum SA: 1968998550454
Sum DA: 807067396607
Sum HA: 660446405678
Sum DC: 77506914005

Average RT: 36
Average SA: 25
Average DA: 10
Average HA: 8

Tue Nov 23 22:51:56 2004

Sum RT: 4305668505572
Sum SA: 2980613160784
Sum DA: 1221716403451
Sum HA: 999765390175
Sum DC: 117327718266

Average RT: 36
Average SA: 25
Average DA: 10
Average HA: 8

Tue Nov 23 23:51:57 2004

Sum RT: 5770160994475
Sum SA: 3994416488251
Sum DA: 1637259797346
Sum HA: 1339816617795
Sum DC: 157234494830

Average RT: 36
Average SA: 25
Average DA: 10
Average HA: 8

Wed Nov 24 00:51:57 2004

Sum RT: 7193347472402
Sum SA: 4979620828287
Sum DA: 2041085722728
Sum HA: 1670278149585
Sum DC: 196015727400
```

```
Average RT: 36
Average SA: 25
Average DA: 10
Average HA: 8

Wed Nov 24 01:51:57 2004
```

B.2 File report_32.txt

```
Wed Nov 24 01:53:29 2004

Sum RT: 1612409508735     Sum RT: 3262008543434     Sum RT: 4889481940805     Sum RT: 6541694221790
Sum SA: 1084530102328     Sum SA: 2194074431631     Sum SA: 3288732827422     Sum SA: 4400030120361
Sum DA: 486440913899      Sum DA: 984104022780      Sum DA: 1475086132371     Sum DA: 1973531330367
Sum HA: 357044056748      Sum HA: 722324076321      Sum HA: 1082700682164     Sum HA: 1448554839207
Sum DC: 21843932273       Sum DC: 44191648792       Sum DC: 66239596925       Sum DC: 88622727285

Average RT: 73            Average RT: 73            Average RT: 73            Average RT: 73
Average SA: 49            Average SA: 49            Average SA: 49            Average SA: 49
Average DA: 22            Average DA: 22            Average DA: 22            Average DA: 22
Average HA: 16            Average HA: 16            Average HA: 16            Average HA: 16

Wed Nov 24 02:53:29 2004  Wed Nov 24 03:53:30 2004  Wed Nov 24 04:53:31 2004  Wed Nov 24 05:53:32 2004

Sum RT: 8199912312776
Sum SA: 5515371763185
Sum DA: 2473789634148
Sum HA: 1815739772478
Sum DC: 111087201570

Average RT: 73
Average SA: 49
Average DA: 22
Average HA: 16

Wed Nov 24 06:53:33 2004
```

B.3 File report_64.txt

```
Wed Nov 24 20:52:34 2004

Sum RT: 1337446848508     Sum RT: 2665825382824     Sum RT: 3991660800970     Sum RT: 5320809160296
Sum SA: 881564575287      Sum SA: 1757139806682     Sum SA: 2631049575821     Sum SA: 3507133483440
Sum DA: 433375554545      Sum DA: 863805971465      Sum DA: 1293415699352     Sum DA: 1724096055992
Sum HA: 302158541726      Sum HA: 602262965087      Sum HA: 901795135682      Sum HA: 1202073871775
Sum DC: 9012489041        Sum DC: 17963880963       Sum DC: 26898104912       Sum DC: 35854651533

Average RT: 148           Average RT: 148           Average RT: 148           Average RT: 148
Average SA: 97            Average SA: 97            Average SA: 97            Average SA: 97
Average DA: 48            Average DA: 48            Average DA: 48            Average DA: 48
Average HA: 33            Average HA: 33            Average HA: 33            Average HA: 33

Wed Nov 24 21:52:34 2004  Wed Nov 24 22:52:34 2004  Wed Nov 24 23:52:35 2004  Thu Nov 25 00:52:36 2004

Sum RT: 6650648051546     Sum RT: 7981366495474     Sum RT: 9314213094464     Sum RT: 10645941668538
Sum SA: 4383673997789     Sum SA: 5260788933850     Sum SA: 6139309078834     Sum SA: 7017085940075
Sum DA: 2154997313101     Sum DA: 2586181412494     Sum DA: 3018058720280     Sum DA: 3449570519562
Sum HA: 1502507042359     Sum HA: 1803137279299     Sum HA: 2104251196337     Sum HA: 2405109679626
Sum DC: 44815915489       Sum DC: 53783032283       Sum DC: 62764562757       Sum DC: 71738532533

Average RT: 148           Average RT: 148           Average RT: 148           Average RT: 148
Average SA: 97            Average SA: 97            Average SA: 97            Average SA: 97
Average DA: 48            Average DA: 48            Average DA: 48            Average DA: 48
Average HA: 33            Average HA: 33            Average HA: 33            Average HA: 33

Thu Nov 25 01:52:37 2004  Thu Nov 25 02:52:37 2004  Thu Nov 25 03:52:38 2004  Thu Nov 25 04:52:38 2004

Sum RT: 11969119390962    Sum RT: 13301454729884
Sum SA: 7889231792254     Sum SA: 8767425284200
```

```
Sum DA: 3878312280886      Sum DA: 4310024380627
Sum HA: 2704037394022      Sum HA: 3005036418421
Sum DC: 80654877157        Sum DC: 89632936721

Average RT: 148            Average RT: 148
Average SA: 97             Average SA: 97
Average DA: 48             Average DA: 48
Average HA: 33             Average HA: 33

Thu Nov 25 05:52:39 2004   Thu Nov 25 06:52:40 2004
```

B.4 File report_128.txt

```
Thu Nov 25 19:53:16 2004

Sum RT: 94158116594        Sum RT: 188834767128       Sum RT: 283484769848       Sum RT: 378519904168
Sum SA: 61334669985        Sum SA: 123001549653       Sum SA: 184655449183       Sum SA: 246555065679
Sum DA: 31934769003        Sum DA: 64044345560        Sum DA: 96145510007        Sum DA: 128376484410
Sum HA: 21756570609        Sum HA: 43632272099        Sum HA: 65502028544        Sum HA: 87460250868
Sum DC: 316285626          Sum DC: 634314795          Sum DC: 952253238          Sum DC: 1271485463

Average RT: 297            Average RT: 297            Average RT: 297            Average RT: 297
Average SA: 193            Average SA: 193            Average SA: 193            Average SA: 193
Average DA: 100            Average DA: 100            Average DA: 100            Average DA: 100
Average HA: 68             Average HA: 68             Average HA: 68             Average HA: 68

Thu Nov 25 20:53:16 2004   Thu Nov 25 21:53:17 2004   Thu Nov 25 22:53:17 2004   Thu Nov 25 23:53:18 2004

Sum RT: 472667102146       Sum RT: 567334881050       Sum RT: 662044904062       Sum RT: 756714150438
Sum SA: 307879874389       Sum SA: 369542155903       Sum SA: 431232916611       Sum SA: 492899748439
Sum DA: 160307237434       Sum DA: 192414426081       Sum DA: 224534499936       Sum DA: 256644104568
Sum HA: 109214037817       Sum HA: 131088258051       Sum HA: 152970982266       Sum HA: 174846610776
Sum DC: 1587737550         Sum DC: 1905746445         Sum DC: 2223887011         Sum DC: 2541889061

Average RT: 297            Average RT: 297            Average RT: 297            Average RT: 297
Average SA: 193            Average SA: 193            Average SA: 193            Average SA: 193
Average DA: 100            Average DA: 100            Average DA: 100            Average DA: 100
Average HA: 68             Average HA: 68             Average HA: 68             Average HA: 68

Fri Nov 26 00:53:18 2004   Fri Nov 26 01:53:19 2004   Fri Nov 26 02:53:19 2004   Fri Nov 26 03:53:19 2004

Sum RT: 850524137828       Sum RT: 944967291390       Sum RT: 1039312499196
Sum SA: 554001362925       Sum SA: 615518935154       Sum SA: 676972559434
Sum DA: 288458935772       Sum DA: 320491010522       Sum DA: 352487793991
Sum HA: 196521752951       Sum HA: 218344632626       Sum HA: 240143349406
Sum DC: 2857013425         Sum DC: 3174253781         Sum DC: 3491168623

Average RT: 297            Average RT: 297            Average RT: 297
Average SA: 193            Average SA: 193            Average SA: 193
Average DA: 100            Average DA: 100            Average DA: 100
Average HA: 68             Average HA: 68             Average HA: 68

Fri Nov 26 04:53:19 2004   Fri Nov 26 05:53:20 2004   Fri Nov 26 06:53:21 2004
```

References

[1] Israel Koren. *Computer Arithmetic Algorithms.* A K Peters, 2nd edition, 2002. ISBN 1-56881-160-8.

[2] Michael D. Ciletti. *Advanced Digital Design with the Verilog HDL.* Prentice Hall, 2003. ISBN 0-13-089161-4.

[3] Ian T. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995. ISBN 0-201-57594-9.

[4] Mi Lu. *Arithmetic and Logic in Computer Systems.* Wiley-Interscience, 2004. ISBN 0-471-46945-9.

[5] Pong P. Chu and Robert E. Jones. *Design Techniques of FPGA Based Random Number Generator.* Research Paper (2004), pages 2–3.