A RECONFIGURABLE DATA FLOW ARCHITECTURE FOR SIGNAL PROCESSING APPLICATIONS

Amitabha Sinha, Dhruba Basu,
Department of Computer Science and Engineering Techno India,
An Engineering College affiliated to West Bengal University of Technology, Calcutta, India

DOI: 10.36724/2664-066X-2021-7-5-2-6

ABSTRACT

This paper aims to devise a data flow model of computation for signal processing applications in which the operational nodes are signal/image processing functions such as Pixsum, Edge, Smooth [12]. These functions are configured during run time from a pool of reconfigurable FPGAS[4][5][6]. Thus because of the data flow model of computation, the signal processing functions execute concurrently. At the same time, these functions by exploiting their inherent spatial parallelism execute at high speed. There is a two fold speed up in the execution of image/signal processing applicationsone at the architecture level wherein a node of the dataflow model executes a digital signal processing (DSP) function rather than a low level machine operation. The second speed up is due to the fact that each DSP function is configured to execute in an FPGA by using maximally the concurrent operations that such a function permits. Another significant benefit that arises from our proposed architecture is that by reconfiguring an FPGA for a DSP function at run time, the reusability of the hardware elements results in reduced cost of operations. In this paper we provide an outline of the data flow architecture and its operational aspects.

KEYWORDS: *Semiconductor device modeling, RF IC Design, CMOS and BiCMOS circuit simulations, circuits for communications.*

The article is reworked from unpublished 2nd IEEE International Conference on Circuits and Systems for Communications (ICCSC) materials.

INTRODUCTION

Intensive and complex computations are required for real time execution of signal/image processing algorithms. Such algorithms exhibit spatial parallelism and are therefore suitable for mapping into array architectures like systolic, SIMD etc. [1], [2], [3]. Speed enhancement can be achieved by using dedicated hardware that exploits the inherent parallelism in the algorithm. Data flow architecture offers a possible way of exploiting the concurrency of computations. Here firing or execution of an instruction/function is asynchronous [7], [8], [9] and depends on the availability of data from one or more processing elements (PE) which would have been fired previously. Thus the execution is very fast. However, the conventional data flow model suffers from two inefficiencies:

1 All the PEs may not be in use at any particular instant and thus the hardware utilization factor becomes low.

2. The PEs are basically ALUs and perform low level operations like Add, Sub, Mul etc. and not the functional level operations like FFT, Cosine Transforms etc..

In this paper we propose a new architecture based on the data flow principle which uses a common pool of RPEs available for the execution of image / signal processing algorithms

There is a two fold speed up in the execution of image/signal processing applications- one at the architecture level wherein a node of the dataflow model [10], [11] executes a digital signal processing (DSP) function rather than a low level machine operation. The second speed up is due to the fact that each DSP function executes in an FPGA and uses maximally the spatial parallelism that such a function possesses.

Each RPE is configured during run time to carry out a specific function as required by the algorithm. On the completion of the execution of the function, the RPE is returned to the common pool and available for being reconfigured into a new function depending on the requirement. Thus the hardware utilization is very high and the same pool of 'n' RPEs can be used again and again to execute different functions. In section II we describe the proposed architecture and the model of the RPEs that is required to support the architecture. Section III gives an overview of the state each PE can have along with the state transition diagram. The functioning of a PE manager which manages these transitions with support from the various blocks in the architecture is also discussed. We conclude in Section IV by highlighting the possible application areas.

PROPOSED ARCHITECTURE

The proposed architecture can be explained with the help of the block diagram of Figure 1. It consists of two main blocks – the Processing Unit (PU) and the Activity Template Storage Unit (ATSU). The Update Unit (UU) and the Fetch Unit (FU) act as interfaces between the PU and the ATSU.

The execution of the application algorithm is carried out by the n RPEs in the PU.

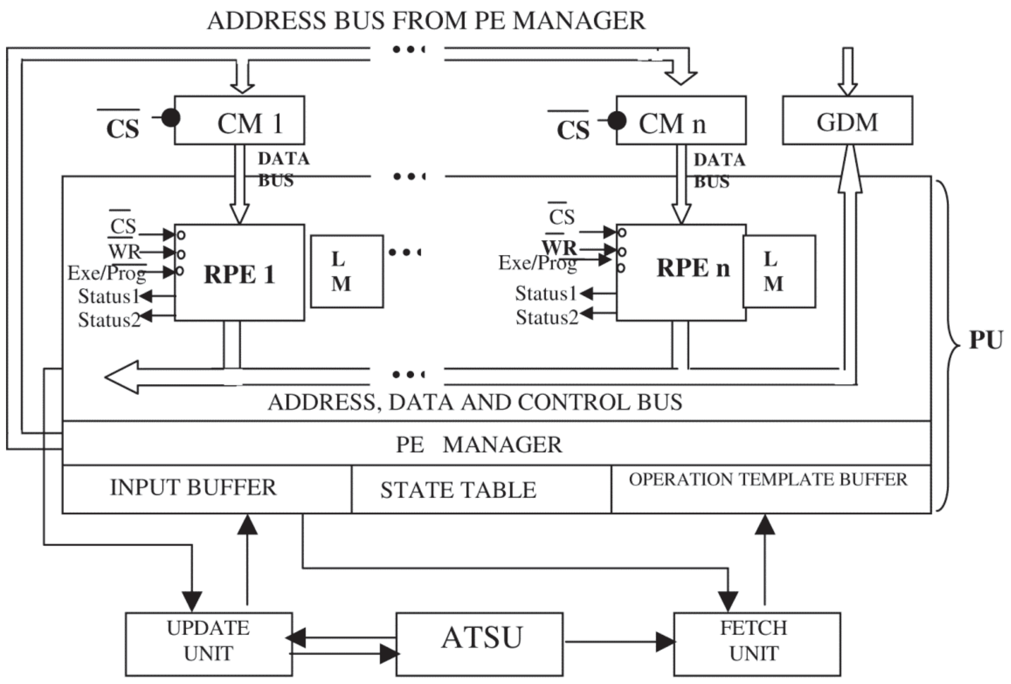


Figure 1

These elements are implemented using Field Programmable Gate Array(s) (FPGA) which are reconfigured for a particular function. This reconfiguration takes place by downloading the required bits-streams from Configuration Memories (CM).

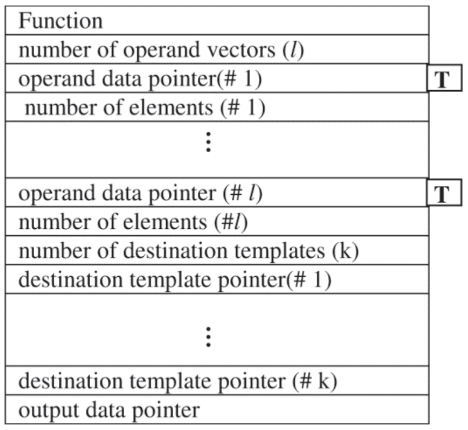
Each RPE has its own local memory (LM) each of which stores all the required reconfiguration function files for the application. The LMs are connected by a global bus to a Global Data Memory (GDM). An RPE is put into execution mode when all the input operand vectors required by the function for which it has been configured are available in the GDM. This availability of input operands is as a result of the completion of execution of its predecessor functions in other RPEs according to the data flow graph. When an RPE finishes execution of a function, it stores the result vector in the GDM, outputs the result packet to the UU for transmission to the ATSU and then makes itself available for possible reconfiguration into another function as required by the data flow model of the application algorithm. The overall management of the RPE s is carried out by the PE manager in the PU. The "result packet" transmitted to the UU has the following structure.

< Result Packet > := < Result Data Pointer > <Size><{Destination Template Address in ATSU} lm >

The ATSU is a memory in which is stored information regarding the various signal/image processing functions in data structures represented by a common template (Fig. 2). In BNF notation, a template can be represented as

< Template in ATSU > := < Function> <{ operand data }1l> <{destination template address}1k > <output data pointer> <no. of elements of output data>

< operand data > := < operand data pointer > < no. of elements of operand data>


Function
number of operand vectors (l)
operand data pointer(# 1)
T
number of elements (# 1)
⋮
operand data pointer (# l)
T
number of elements (#l)
number of destination templates (k)
destination template pointer(# 1)
⋮
destination template pointer (# k)
output data pointer

Each template in the ATSU represents a distinct function in the data flow graph and its first field depicts that function. Next it has a minimum of one field to a maximum of 'l' fields of operand data, each field representing a pointer to an array in the GDM and the number of elements in each array. The template has further one field indicating the number of destination templates 'k' to which its output data should be passed on. The addresses of these 'k' templates in the ATSU are next. The last two fields represent the starting address of the GDM into which the function output vector is written and the number of elements of the output vector respectively.

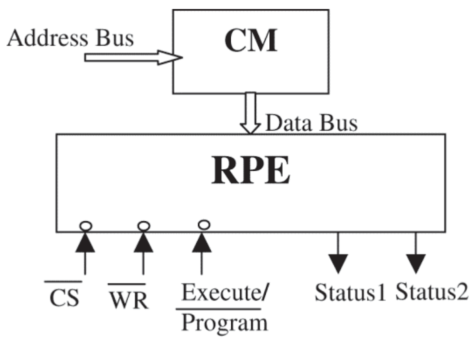


Figure 2. Model of RPE

It is clear from the above that it is the contents of the ATSU that describe the data flow model of the application algorithm to be executed in the PU. To each operand data pointer is appended a tag bit T, which are initially cleared. As and when an input operand becomes available from an executing RPE, the result packet indicates so to the UU which in turn sets the tag bit to 1. When all the tag bits of a particular template become 1, the template is ready for execution and indicates so to the UU. The UU then sends the following packet to an input buffer (IB) in the PU.

<IB Packet> : = <Function> <address of corresponding template in ATSU>.

At any point in time the IB maintains a queue of "ready to execute" functions along with the addresses of the functions' templates in the ATSU. Since the number of PEs is n, we restrict the size of the IB queue also to n as this would not affect the performance in any significant way.

It may be instructive to note that when more than n functions become ready for execution, they are kept pending in the ATSU and are transferred to the IB only when the latter's size falls below n due to the allocation of one or more functions to the PE s which have become free for allocation as determined by the PE manager. The tag bits of a template in the ATSU are cleared as soon as the function represented by the template is moved into the IB.

A state table in the PU maintains the states of the n RPEs i.e. whether these are in the running, idle or reconfiguration mode. The PE manager with the help of this state table and the IB determines whether a particular PE can be set to either execution mode or reconfiguration mode. The model of an RPE is illustrated in Fig 3 and explained below.

The address bus common to all CMs carries the addresses of the locations from which are retrieved a series of configuration bytes via dedicated data buses. The chip select bar (CS) of an RPE determines the RPE which accepts the addresses from the common address bus.

The configuration bytes are then stored internally in the RPE and help reconfigure the RPE to execute a different function. These operations on the RPE are controlled by the PE manager with the help of the signal Execute/Program. The status output signals "Status1" and "Satus2" indicate to the PE manager the status of an RPE as given in Table 1. When a function in the IB has been allotted to an RPE for execution, a request is sent by the IB to the FU to fetch the corresponding template from the ATSU. The FU obtains the template from the ATSU and puts it in the Operation Transfer Block (OTB) for formation of the result packet. Since the number of executing RPEs can be at most n, the size of the OTB should also be n.

FUNCTIONING OF THE PE MANAGER

The PE manager, which manages the n RPEs and their state transitions, consists of essentially two processes, Process 1 and Process 2. Process 1 uses the IB and a state table to manage the state transitions while Process 2 uses the output status lines status 1 and status 2 of the RPEs.

Table I

Status Outputs of an RPE

Status Output Lines		State of the RPE
Status 1	Status 2	
0	0	Idle/Execution complete
0	1	Under configuration
1	0	Configuration complete/ Ready
1	1	Executing

A state table as depicted in Table II maintains a record of the n RPEs, their current states and the functions allotted to them.

Table II

State Table i

RPE	STATE	FUNCTION
1	S_i, i ε {1,2,3,4,5}	f_j, j ε {1,2,…z} when S_i: i ε {2,3,4,5} and f_j is undefined when S_i: i=1. z = number of function templates in ATSU
2	,,	,,
. . .	. . .	. . .
n	,,	,,

Each entry in the state table corresponds to an RPE and is identified by its RPE number. Each RPE can be in one of the five state, S_1, S_2, S_3, S_4 and S_5, where;

S_1 => yet to be configured

S_2 => under configuration

S_3 => configuration complete / ready S4 => executing

S_5 => execution complete.

We propose that the transitions among the states in an RPE is carried out as per the state transition diagram shown in Figure 3.

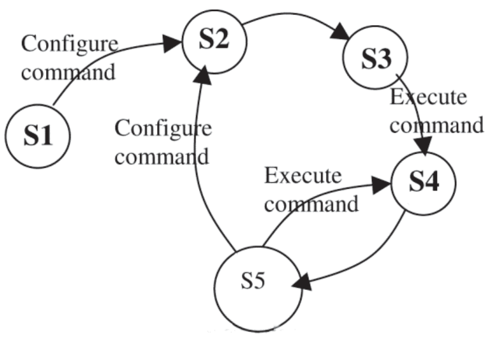


Figure 3. State Transition Diagram of an RPE

Thus initially at start up all RPEs are in S1 and make a transition to S2 as per the requirements of the algorithm. It is assumed that at start up the IB is filled with the functions which exists at the first level in the data flow graph. The corresponding entries in the state table are updated to S1.

The state table also includes the function which is under execution or under configuration in a particular RPE. It is instructive to note that when an RPE is in state 1 (as at initial start up), no function has yet been allotted to it and therefore a state table does not have a defined entry in its function column. The function that can exist in a state table can be any one of the function templates that exist in the ATSU. The IB is a first in first out (FIFO) buffer where the functions get queued in the order of their arrival from the UU and are serviced by the PE manager starting with the function at the queue head and ending with that at the queue tail according to Process 1 shown below.

```
Process 1 :
Do until IB queue tail
1{ take function (h) from queue head;
search state table associatively with $key_1$ = < function(h) >
    < { $S_3$ |$S_5$ } >;
if at least one match exists, do the following for one of the table entries where match occurs

i)      issue "Execute" command to the RPE ;
ii)     update the state entry in the state table for the corresponding RPE to $S_4$;
iii)    delete function (h) from queue and increment queue head pointer;
}11
else
12{ search state table associatively with $key_2$ = < { $S_1$ | $S_5$ } >; if at least one match do the following
121{
    i)      issue "configure" (program) command to the RPE;

    ii)     update the corresponding state table entry to $S_2$;
    iii)    generate signals on the address bus of the CMs corresponding to function (h);
    iv)     activate the CS signal to the corresponding RPE;
    v)      increment queue head pointer;
 }121
else
122{ increment queue head pointer}
    } 122
    } 12
    } 1
```

It can be seen that Process 1 causes the following state transitions in Figure 4, $S_1 \rightarrow S_2$, $S_3 \rightarrow S_4$, $S_5 \rightarrow S_4$, and $S_5 \rightarrow S_2$. The transitions $S_2 \rightarrow S_3$ and $S_4 \rightarrow S_5$ require the examination of the 2 status output lines, status 1 and status 2 for each of the n RPEs and is carried out by Process 2 as follows.

```
    Process 2 : Do for i = 1 …….. n.
    { Read status bits : status 1, status 2,
    switch (status 1i, status 2i)
    { case status 1i= 0, status 2i = 0:
update state table enter to S5;  issue reconfigure command
to RPE; break;
   case status $1_i$ = 1, status $2_i$ = 0:
update state table entry to S2;  issue execute command to
RPE break;
default:
        break
}
```

Theorem 1
Process 1 and Process 2 can run concurrently.

Proof. Process 1 reads the IB and state table and causes the following state transitions $S_1 \rightarrow S_2$, $S_3 \rightarrow S_4$, $S_5 \rightarrow S_2$, $S_5 \rightarrow S_4$ by writing into the state table and issuing either "configure" or "execute" command to the RPEs. Process 2 reads the status pins of the RPEs and causes the following state transitions $S_2 \rightarrow S_3$ and $S_4 \rightarrow S_5$ by writing into the state table. It can be seen that the two processes cause state transitions which are distinct from each other. It is shown below that logically there cannot be any conflict between the two processes when they execute concurrently. Logical conflict between the two processes can arise in either of the following two situations:

1. Two processes attempt to write two different data in the same location at the same time.
2. One process reads from while another process writes into the same location at the same time.

In our design, Process 1 writes into a location indicating states S_1, S_3 or S_5 whereas Process 2 may at the same time write into a location representing states S_2 or S_4. Therefore the two processes can never attempt to write into same location at the same time.

In the case of a simultaneous read and write, a conflict can arise if the write is a multi field write and the outcome of the read is erroneously affected if the read operation completes before the multi field write is completed. But in our proposed design the outcome of the read by Process 1 depends on the reading of one field only viz. the status field and the Process 2 also writes only one field. Therefore there is no logical conflict in the case also.

However it is possible that when Process 2 updates the state table, Process 1 tries to read it. But this problem can be solved by providing a hardware lock mechanism, which prevents such simultaneous access. This can utmost delay the execution of one of the processes for the duration of the locking but cannot cause any logical error.

CONCLUSION

In this paper we have outlined an architecture suitable for executing algorithms which possess parallelism and which need to be executed in high speed real time applications. Emerging areas such as video and image processing for 3G mobile and security applications are likely to benefit by applying our proposed architecture.

REFERENCES

[1]. S.Y.Kung. VLSI Array Processor, PrenticeHall, 1988.

[2]. H.J. Siegal, et al. PASM: partitionable SIMD/MIMD system for image processing and pattern recognition, *IEEE Trans. Computer*., Vol.C30, no. 12, pp. 934-947, Dec 1981.

[3]. A.Sinha, P.C. Jain, V. Mitra. Asynchronous SIMD – A New Architecture For A Class of Image Processing Applications, proc. *Fifth National Conf. On Communication*, pp. 287-294, IIT Kharagpur, India. 1999.

[4]. Hungwen Li and Quentin F.Stout. Reconfigurable SIMD Massively parallel Computers, *Proc. IEEE*, Vol. 79, No.4, April 1991, pp. 429-443.

[5]. John Villasenor and William H.Mangione Smith. Configurable Computing, Scientific American Article, June 1997.

[6]. John Villasenor and Brad Hutchings. The Flexibility of Configurable Computing, *IEEE Signal Processing Magazine*,1998, pp. 67-83.

[7]. J.B. Dennis. Data Flow Supercomputers, Computer, pp. 48-56, Nov., 1980.

[8]. Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yeutsu Kodama, and Toshitsugu Yuba. An Architecture of a Dataflow Single Chip Processor, *In Proc. 16th Annual Int'l Symposium on Computer Architecture*, pp 46-53. ACM, 1989.

[9]. J.B. Dennis. MIT "Stream Data Types for signal processing", Advanced topics in Dataflow Computing and Multithreading (Gao, Bic, Gaudiot eds.) IEEE Computer Society Press, 1995.

[10]. P. Netezki. Exploiting Data Parallelism in Signal Processing on a Dataflow Machine, ACM Annual Int'l Symposium on Computer Architecture, pp. 262-272, 1989.

[11]. Ben Lee and A.R. Hurson. Dataflow architectures and multithreading, IEEE Computer, pp. 27-28, Aug 1994.

[12]. Edward R. Dougherty & Charles R. Giardina. Matrix Structured Image Processing, Prentice Hall, 1987.