Research Paper

Volume : 1 | Issue : 6 | Nov 2012 • ISSN No 2277 - 8179

Engineering

KEYWORDS : CGRA, power efficiency, performance improvement, floating point operations, Reconfigurable Architecture.

Floating Point Operations in PipeRench CGRA

M. Somasekhar

GURU NANAK INSTITUTE OF TECHNOLOGY , AHIMPATNAM, HYDERABAD, ANDHRA PRADESH

ABSTRACT

This paper introduces PipeRench, Coarse-grained reconfigurable architectures (CGRAs) have drawn increasing attention due to their performance and flexibility. CGRA provide flexible and efficient solution for data-intensive applications. There have been many coarse-grained reconfigurable architectures proposed and/or commercialized. But most of the existing architectures cannot be used for applications that require floating-point operations, since they have only integer units. This paper introduces approache onto CGRAs supporting both integer and floating point arithmetic. Two-dimensional array of integer processing elements in the PipeRench is configured in run-time to perform integer functions as well as floating-point functions. In this paper, we presented PipeRench optimized for floating-point as well as integer computations. we show that the proposed PipeRench enjoys improved speed for floating-point intensive applications by implementing mesh-plus interconnectivity.

i. Introduction

WITH THE INCREASING requirements for more flexibility and higher performance in embedded systems design, reconfigurable computing is becoming more and more

popular. Various coarse-grained reconfigurable architectures (CGRAs) have been proposed in recent years, with different target domains of applications and different tradeoffs between flexibility and performance. PipeRench (also known as Cached Virtual Hardware or CVH) is a run-time reconfigurable Field-Programmable Gate Array (FPGA) which manages a "virtual" pipeline. The fabric of the FPGA contains physical pipeline stages which, though limited in number, can be reconfigured separately at run-time to perform numerous calculations in an application. In this way, the physical pipeline stages, or "stripes," are configured and arranged to form the virtual pipeline which executes the desired task. The logical size of a virtual pipeline is unbounded, and it can run on a compatible architecture of any size. Such hardware virtualization provides the benefits of forward-compatibility and more robust compilation.

The remainder of this paper is organized as follows. Sec-tion II introduces our CGRA supporting both integer-type application domains and floating-point-type application domains. Finally, Section III concludes this paper.

II. Target Architecture

A. Coarse-Grained Reconfigurable Architecture

PipeRench Architecture

The architecture that can currently be targeted by the Cached Virtual Hardware Assembler is a restricted class of the architectures described in the previous section. This section details the architecture in a top-down manner, first discussing the stripe and then the processing elements (PEs) that are contained in the stripe. The discussion then moves to the routing and configuration of PEs, and finally to the memory controllers which handles flow of data between the FPGA fabric and the memory.

1. The Stripe

In a hierarchical sense, the stripe is the largest, most general block of PipeRench. Each stripe performs a computationally significant portion of the application and is analogous to a pipeline stage. Registers between stripes separate each stripe from the rest of the architecture, providing the pipeline capability. Figure 6 shows two stripes along with the local bus which connects each stripe to its successor. Each stripe is made up of ***N*** computational logic blocks, or Processing Elements (PEs). PEs are described in detail in Section 2.

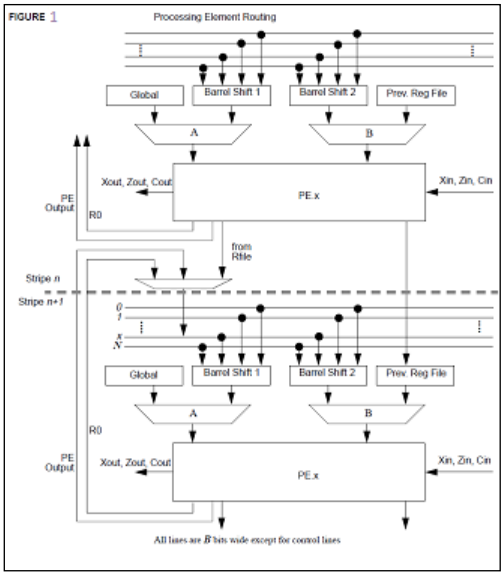
2. The Processing Element

All computation for a PipeRench application is done in the processing elements. Each PE in a stripe can take its inputs from several different locations, and output to various other locations within the fabric. More details follow on the features of a processing element.

2.1 Inputs

A PE takes two ***B***-bit inputs from the local bus, the global bus, or the registered output of the previous stripe. These inputs, A and B, are the values to be operated on within the PE. Operand A can read a value from the local bus, or it can read a value from the global bus. Operand B can read a value from the local bus, or it can read a value directly from the register file of the same PE in the previous stripe.

Other inputs to the PE come from an adjacent PE in the same stripe. These 1-bit values are named Cin (carry-in), Zin (zero in), and Xin (general purpose), and can be chosen from the three inputs to the "Misc. Routing" block, although a limited number of combinations are allowed. One might choose to define Xin to equal the Cout from the previous PE, for example, in order to perform some logical function on that value. Figure 2 shows in detail where the particular inputs come into the PE.



2.2 Outputs

The ***B***-bit functional output of the PE can be saved into the register file and passed on to the next stripe. In this case, the value can be fed directly to the B input of the corresponding PE in the next stripe. The PE output can also be routed, registered or unregistered, to the local bus, as shown in Figure 2. This value can be picked up from the local bus by another PE in the same stripe.

1-bit outputs Cout (carry-out), Zout (zero out), and Xout (general purpose) are routed through the "Misc. Routing" block to the neighboring PE on the left. Cout is the carryout value from the top bit of the carry chain. Zout has a value of 0 if the PE output is zero, and 1 otherwise. The Xout of a PE is hardwired to the Xin of the same PE; this allows the PE to pass a value from the previous PE to the next PE.

2.3 Barrel Shifter

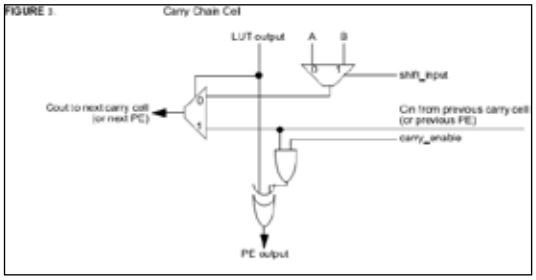
Connecting inputs from the local busses to the PE are barrel shifters for each operand, as seen in Figure 2. Currently, the barrel shifters can only shift left or rotate left, although a shift or rotate right can be accomplished through a combination of the routing resources.

The barrel shifter can take as its input any ***B***-bit window from the total value available to it (***B*N*** bits wide, spanning all ***N*** of the local busses). For example, in an architecture with ***N***=4 and ***B***=4, imagine that the following 16 bit value was placed on the four local busses:
1 0 1 1 1 0 [0 1 0 1] 1 0 1 0 1 1

Any four-bit window on this value can be grabbed by the barrel shifter, and then that value can be shifted or rotated left by any amount. In this case, the value 0101 was grabbed, so if it was shifted left by 2, the resulting input to the PE would be 0100. If it was rotated left by three, we would get 1010 as the PE's input. This flexibility allows a great number of possibilities for manipulating data.

2.4 Ripple-Carry Chain

The ripple-carry chain is intended to be used whenever a PE is performing an addition or subtraction. When the LUT output is combined with its inputs, the ripple-carry chain can output the sum (or difference) of the twoB-bit inputs, A and B. The most significant carry-out result can be passed on to the next PE through Cout, so that addition and sub-traction of values greater thanBbits is supported. The "carry_enable" bit chooses whether the carry chain should pass along only the LUT output or whether it should compute the LUT output XOR'ed with the carry in from the previous PE. In this way, a function such as can be expressed by the PE.



An additional function of the carry chain is realized by activating the "shift_input" bit. It can select which input, A or B, should be shifted through the carry chain as a carry-in for the next carry chain cell

B. Architecture Extension for Floating-Point Operations

A pair of PEs in the PE array is able to compute floating point operations according to its configuration. While a PE in the pair computes the mantissa part of the floating point operations, the other PE handles the exponent part. Since the data path of a PE is 16 bits wide, the floating point operations do not support the single precision IEEE-754 standard, but support only reduced mantissa of 15 bits. However, experiments show that the precision is good enough for hand-held embedded systems .

Since adjacent PEs are paired for floating-point operations, the total number of floating-point units that can be constructed is half the number of integer PEs in the PE array as shown in Fig. 3(a) and (b). The PE array has enough interconnections among the PEs so that it makes use of the interconnections for the data exchange between the PEs required in floating-point operations.

Mapping a floating-point operation on the PE array with integer operations may take many layers of cache. If a kernel consists of a multitude of floating-point operations, then mapping it on the array easily runs out of the cache layers, causing costly fetch of additional context words from the main memory. Instead of using multiple cache layers to perform such a complex operation, we add some control logic to the PEs so that the operation can be completed in multiple cycles
but without requiring multiple cache layers. The control logic can be implemented with a small finite state machine (FSM) that controls the PE's existing datapath for a fixed number of

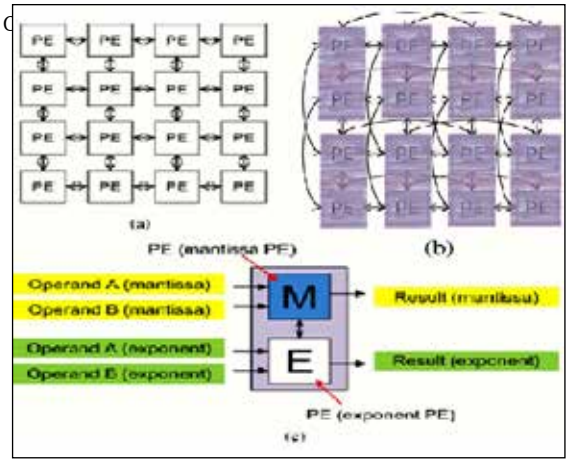


Fig. 3. Diagram of 4×4 PE array: combination of a pair of PEs for floatingpoint operation in the same integer PE array. (a) Integer PE array (4×4). (b) Floating-point PE array (2×4). (c) Pair of integer PEs executes floatingpoint operation

We consider three different interconnect topologies: mesh, mesh-plus, and full. In the mesh-plus connectivity, interconnects of two-hop distance are added to the mesh interconnects. As expected, the more interconnect resources are provided between PEs, the less mapping time is needed, and the less cycles are needed to run the application. It is due to the fact that adding more interconnect resources renders better routability between PEs, and therefore the operations can be easily scheduled without spending much time to find routing paths between the operations. However, adding more interconnect resources may lead to more wires and wider multiplexors, which incurs more area, longer delay, and more power consumption.

TABLE I

According to the Three Different Interconnect Topologies

	Mesh	Mesh-Plus	Full
Latency(cycle) Mappinf Time(s)	4 9	3 <1	3 <1

III. Conclusion

In this paper, we presented PipeRench optimized for floating-point as well as integer computations. we show that the proposed PipeRench enjoys improved speed for floating-point intensive applications.

REFERENCE

[1] Matthew Myers, Kevin Jaget, Srihari Cadambi, Jeffrey Weener, Matthew Moe, Herman Schmit, Seth Copen Goldstein, Dan Bowersox: "PipeRench Manual" in Carnegie Mellon University | [2] Goldstein: "PipeRench: A Reconfigurable Architecture and Compiler". | [3] M. Jo, V. K. P. Arava, H. Yang, and K. Choi, "Implementation of floating-point operations for 3-D graphics on a coarse-grained reconfigurable architecture," in Proc. IEEE-SOCC, Sep. 2007, pp. 127–130. | [4]. Ganghee Lee, Kiyoung Choi, Senior Member, IEEE, and Nikil D. Dutt: "Mapping Multi-Domain Applications onto Coarse-Grained Reconfigurable Architectures". |