Mapping a mathematical expression onto a Montium ALU using GNU Bison

Michèl A.J. Rosien, Paul M. Heysters, Gerard J.M. Smit
Department of Computer Science,
University of Twente, Enschede, the Netherlands
email: rosien@cs.utwente.nl

Abstract— **The Montium processing tile [1], [4] contains a number of complex ALUs which can perform many different operations in many different ways. In the Chameleon tool flow [2], it is necessary to automatically determine whether a certain mathematical expression can be mapped onto an ALU and to automatically generate an ALU configuration for this expression. This paper describes how the parser generator GNU Bison [5] is used to determine whether a mapping is possible and how** *Generalized LR Parsing* **[6] is used to cope with ambiguities and to generate all possible mappings of a specific expression onto an ALU.**

I. Introduction

In the CHAMELEON project we are designing a heterogeneous reconfigurable System-On-a-Chip (SOC). This SOC contains a general-purpose processor (e.g. ARM core), a bit-level reconfigurable part (e.g. embedded FPGA) and several word-level reconfigurable parts (e.g. Montium tiles). We believe that heterogeneous reconfigurable architectures are needed in future 3G/4G terminals. This paper presents one of the developed tools for the Montium tile processor. We believe that the used techniques can also be used for other hardware platforms.

II. The Montium ALU

The Montium processing tile [1], [4] contains a number of complex Arithmetic Logic Units (ALUs) which can perform many different operations in many different ways. To give an indication of the complexity of the ALU, the expression $max(x+y, z)-q+y$ can be executed in 1 clock cycle on a single ALU in fixed-point mode (in 20 different ways). The main reason for having ALUs of this complexity is to minimize the amount of communication between operations because communication is expensive. The scaling of semiconductor technology causes the energy consumption of wires to become increasingly dominant over computation. Moreover, communication is in essence overhead as it does not contribute to the computations of an algorithm. Large energy savings can be obtained by reducing the energy wasted for communication. An obese ALU can compute a complex operation without external communication. In general, this is know as locality of reference.

Figure 1 shows a simplified overview of a single Montium ALU. It has been simplified because showing all the details would take up too much space. The ALU has four 16-bit inputs and each input has

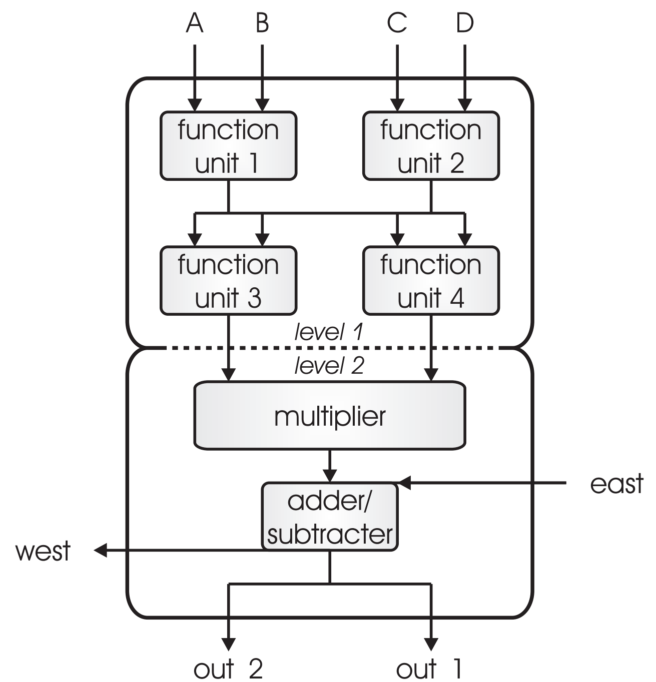

Fig. 1. ALU Overview

a private input register file that can store up to four operands. The input register file cannot be bypassed, i.e. an operand is always read from an input register. Input registers can be written by various sources via a flexible interconnect [1], [4]. An ALU has two 16-bit outputs, which are also connected to the interconnect. The ALU is entirely combinational and consequentially there are no pipeline registers within the ALU. Neighbouring ALUs can also communicate directly on level 2 (see figure 1). The West-output of an ALU con-

© PROGRESS/STW 2004, ISBN 90-73461-41-3

nects to the East-input of the ALU neighbouring on the left. The East-West connection does not introduce a delay or pipeline, as it is not registered. The ALU has 41 control signals. So in theory there are 2^{41} possible configurations. In practice however, many of these configurations will lead to the same functionality. Level 1 contains 4 function units. Each function unit can perform a variety of arithmetic and logic operations: addition, subtraction, saturated addition, saturated subtraction, negate, bitwise AND, bitwise OR, bitwise XOR, shift left/right, bitwise NOT, minimum, maximum and absolute value. It can also generate a number of constants: 0, 1, -1 and -2. Level 2 contains a multiply-add and butterfly structure. Arithmetic can be done in integer mode and in fixed-point mode.

III. Mapping expressions onto an ALU

A specific mathematical expression can often be mapped onto the ALU in many different ways. The expression $x+y+z+q$, for example, can be mapped in 48 ways where x, y, z and q can be one of the inputs A, B, C, D or EAST. Since a program running on the Montium can only have a fixed number of ALU configurations (4 in our current prototype), it is important that the configurations are chosen carefully. Energy efficiency can also be taken into account; because the Montium has to switch between ALU configurations, it is desirable that the number of configuration signals that are changed is kept to a minimum. For this reasons it is very convenient to be able to quickly generate all possible ALU configurations for a specific expression.

One part of the Chameleon tool flow is to divide complex expressions into smaller sub-expressions, called clusters [3], which can be mapped onto the ALUs. These complex expressions could, for example, be expressed in some kind of high-level language like C. In this clustering process it is also often the case that one mapping is better than some other mapping for the reasons described above. Therefore it is important to be able to quickly determine whether a certain sub-expression can be mapped onto an ALU, which means it is a valid cluster. All this should be fully automated so the clustering algorithm can be fully automated as well.

IV. Automated mapping using GNU Bison

A GNU Bison [5] grammar is used to describe the functionality of a Montium ALU. If an input string, a mathematical expression, is accepted by the grammar, it can be mapped onto the ALU. As an example, Figure 2 shows the level 2 multiply-accumulate structure in more detail. The multiply part of this

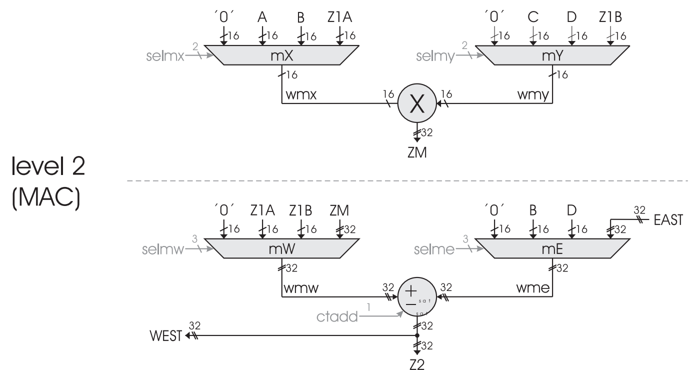

Fig. 2. Level 2: Multiply-Accumulate

structure can be described in bison with the grammar rules shown in figure 3. The quoted strings are ter-

```
ZM : "0"
   | "(" ZM_RIGHT "*" ZM_LEFT ")"
   | "(" ZM_LEFT "*" ZM_RIGHT ")"
;
ZM_LEFT : "0"
        | A
        | B
        | Z1A
;
ZM_RIGHT : "0"
         | C
         | D
         | Z1B
;
```

Fig. 3. Simplified Bison grammar of the multiply structure of the Montium ALU

minals and the other strings are non-terminals. The non-terminals are defined in the rest of the grammar which is not shown. This grammar assumes the input string is a fully parenthesized mathematical expression i.e. with a uniquely specified order of operations. Every expression can easily be transformed into this format and this is done before it is fed to the parser.

A. Ambiguity

Since an expression can often be mapped in a number of different ways, the complete grammar will obviously contain a lot of ambiguities. This can be solved with the Generalized LR Parsing algorithm [6]. Gen-

eralized LR is a parsing algorithm that maintains multiple LR parsing states in parallel. This allows the grammar to be ambiguous and the parser can explore all possible matchings (in GNU Bison 1.875, released jan 2003, this is done using a `%merge` declaration in the grammar). Thus using this technique all possible mappings of an expression onto an ALU can be found.

V. Conclusions

Using the technique described above, an algorithm has been created that can find all possible mappings of a specific mathematical expression onto a Montium ALU. Figure 4 shows a screenshot of the GUI version of the algorithm/tool in action. It shows a possible mapping of the expression $max(x+y,z)-q+y$. Figure

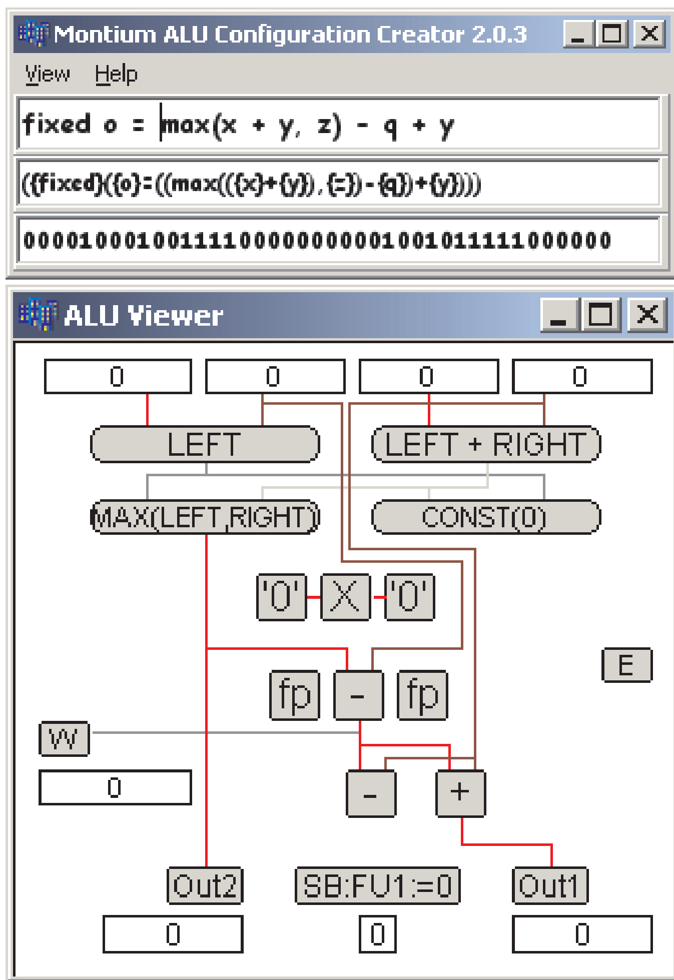

Fig. 4. ALU Mapping GUI

5 shows the ALU configurations of all the mappings the algorithm has found, 20 in this case. Note that signals which are not needed for a mapping are *don't care* as indicated by the hyphens in figure 5. In reality, this window also shows the mapping of variables in the expression to the inputs of the ALU. But the window had to be resized to be able to fit on the page and still be readable. This explains why it seems there are duplicate solutions i.e. the second and third solution. Those solutions differ only in the mapping of variables to inputs.

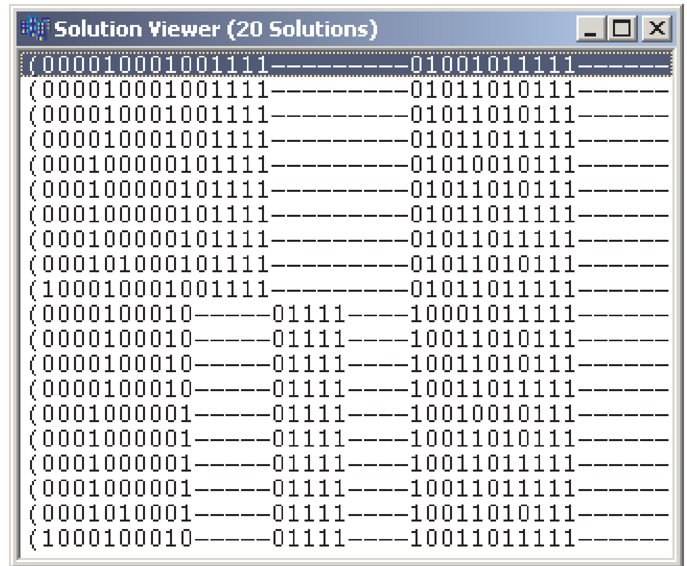

Fig. 5. ALU Mapping Solutions

A. Disadvantages

Because this approach is basically a brute force approach, it doesn't scale well. Adding more inputs to the ALU or trying to map an expression on two ALUs at the same time will drastically increase computation time because the number of permutations of the inputs and outputs will increase dramatically. The mapping of variables to inputs/outputs could be forced to be fixed to reduce the computation time increase and to be able to handle much larger ALUs or multiple connected ALUs.

B. Advantages

Changing the functionality of the ALU is very easy. Adding a different connection or changing the functionality of a function unit, for example, can easily be done by changing the grammar file of Bison. So this approach is very flexible. Also restrictions to which mappings are legal can easily be added. Possible restrictions could be to disallow/force certain inputs to be used, or to disable the butterfly structure, etc. Despite that the brute force approach doesn't scale well, the algorithm performs well for the Montium ALU, generating all mappings for the most difficult expressions in about a second.

References

[1] P.M. Heysters, G.J.M. Smit and E. Molenkamp, "A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems". In *The Journal of Supercomputing*, vol. 26 no. 3 pp. 283-308, Kluwer Academic Publishers, Boston, U.S.A., November 2003, ISSN 0920-8542

[2] G.J.M. Smit, M.A.J. Rosien, Y. Guo, P.M. Heysters, "Overview of the tool-flow for the Montium Processor Tile": In *Proceedings of the international conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, pages 45-51, June 2004, ISBN: 1-932415-42-4

[3] Y. Guo, G.J.M. Smit, P.M. Heysters and H. Broersma, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System". In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pp.199-208, San Diego, USA, June 2003, ISBN 1-58113-647-1.

[4] P.M. Heysters, G.J.M. Smit, B. Molenkamp and G.K. Rauwerda, "Flexibility of the Montium Word-Level Reconfigurable Processing Tile", In *Proceedings of the 4th PROGRESS symposium on Embedded Systems*, pp. 102-108, Nieuwegein, The Netherlands, October 2003, ISBN 90-73461-37-5.

[5] `http://www.delorie.com/gnu/docs/bison/bison_1.html`

[6] `http://www.delorie.com/gnu/docs/bison/bison_90.html`