Features of Dataflow Processor Emulator Implementing

E. A. Kuznetsova[1, *] and **A. A. Rybakov**[1, **]

(Submitted by S. S. Submitter)

[1] *Joint Supercomputer Center of the Russian Academy of Sciences – branch of Scientific Research Institute of System Analysis of the Russian Academy of Sciences, Leninsky prospect 32a, Moscow, 119334, Russia*

Received May 01, 2020

Abstract—The development of dataflow processor architecture is a promising direction for improving the performance of computing systems. The dataflow processors has several advantages in contrast with processor with The von Neumann architecture. These advantages are express in explicit parallelism of calculations, since in the presentation of programs for the dataflow processor the sequence of instructions limited solely by the fact of input data's ready state. The concept of using tokens as data storage for instructions allows you to remove many problems associated with data conflicts and simplifies the program logic. The most important element in the implementation of the logic of the dataflow processor is the content-addressable token memory, which should provide storage and quick search for ready-made data for instructions that can be transfer for execution. This article discusses the functionality of the content-addressable token memory and the logic of its operation using the example of a simple dataflow graph.

2010 Mathematical Subject Classification: 68N20,68R10

Keywords and phrases: *Dataflow processor emulator, dataflow graph, token, token state, associative memory, control flow graph, def-use graph.*

1. INTRODUCTION

Введение про архитектуру потокового процессора.

2. PROCESSOR WITH THE VON NEUMANN ARCHITECTURE EMULATION

One of the basic principles of The von Neumann architecture is the principle of program control sequence. According to it, the executable code of the program is stored in memory, and the processor can execute its instructions sequentially, one by one. The next instruction's address submitted to the processor for execution is stored in a special register called instruction pointer. By default, it always shifts to the next instruction in memory. This address can be forcedly changed by special transition instructions, and in this case we can talk about transferring control to another part of the program.

We introduce the concept of a linear section (or the program's base block). The linear section is a sequence on instructions with the next two properties. Firstly, control could only be transfer to the first instruction of a given section. In other words, a linear section has only one input, which is the first section's instruction. Secondly, if control was transfer to the linear section than all instructions of a given section must be done (i.e., there can be no transition instructions inside the linear section). The transition instruction can only be at the end of the linear section. Sometimes, instead of linear sections, it is convenient to view so-called quasilinear sections, which can have arbitrary amount of transition instructions inside for transitions to other linear sections. We will not review quasilinear sections, because any such section can be split into a sequence of ordinary linear sections.

[*] E-mail: mrallis@jscc.com

[**] E-mail: rybakov.aax@gmail.com

An important concept widely used in the theory of optimizing compilers is the control flow graph. A control flow graph is a directed graph whose nodes are linear sections of the program, and arcs indicate the transfer of control between them. The control flow graph is a convenient and visual structure for describing program's behavior and used by optimizing compilers to implement global program optimizations i.e. optimizations that go beyond linear sections.

Consider the logic of the device intermediate program's representation inside one linear section. Consider for this a simple example of finding roots of a quadratic equation. We will use listing in the C programming language to write examples and pseudocode. So, let us have a linear section (program code block) at the input of which we have the variables a, b, c containing the coefficients of the quadratic equation $ax^2 + bx + c = 0$. At the linear section's output it is required to get the values of the roots, which are calculated by the formulas $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/(2a)$.

Listing 1. The code block for calculating the roots of the quadratic equation.

```c
{
    float t = sqrt(b * b - 4.0 * a * c);

    x1 = (-b + t) / (2.0 * a);
    x2 = (-b - t) / (2.0 * a);
}
```

In Listing 1, linear section is delimited in curly brackets and a temporary variable t is declared to understand that this variable is not used outside the linear section. For convenience, we assume that the processor, for which the executable code will be created, supports the instruction for calculating the square root. Thus, the following instructions for working with real numbers will figure in the listing above: MUL – multiplication of two numbers, SUB – subtraction of two numbers, SQRT – square root of number, NEG – unary negative, ADD – addition of two numbers, DIV – division of two numbers.

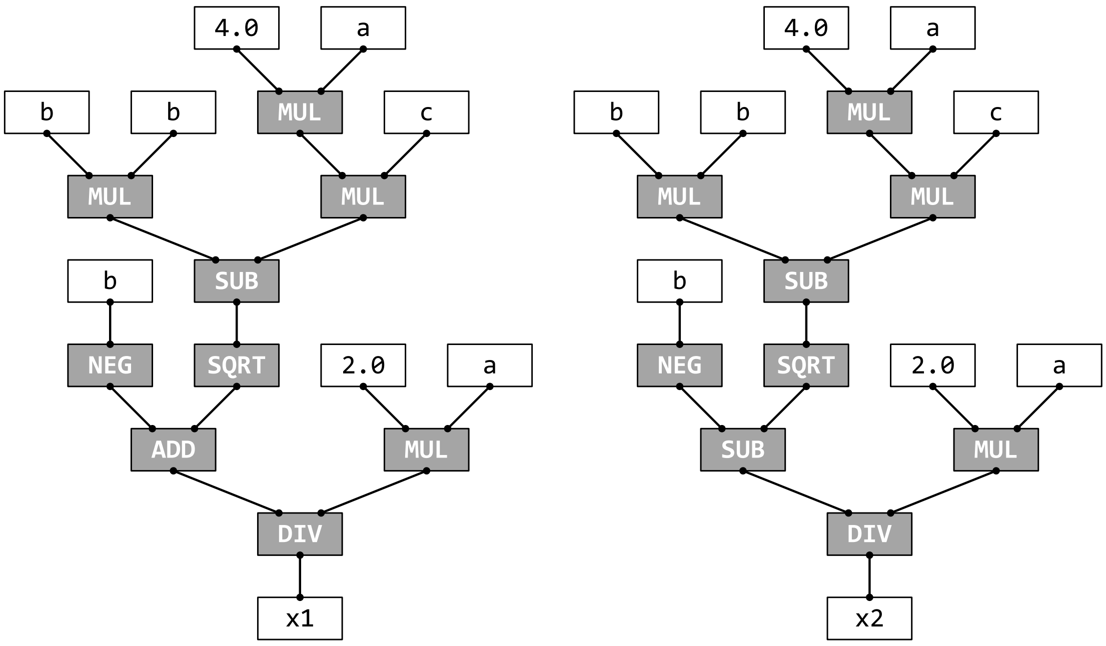

Figure 1. Tree for calculating the expressions of the roots of the quadratic equation.

Figure 1 shows two trees of evaluating the expressions x_1 and x_2. The compiler, of course, will simplify the calculation data by collecting common subexpressions, and the final version of the linear section representation will be close to the next dependency graph shown in Figure 2.

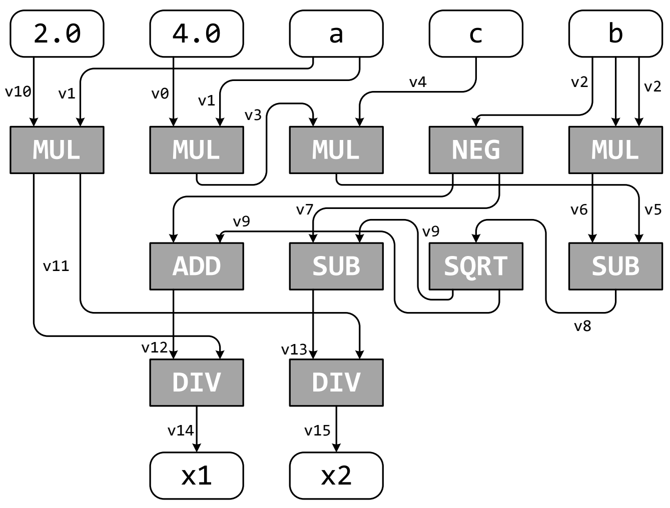

Figure 2. The dependency graph of the linear section in the calculation of the expressions of the roots of the quadratic equation.

Nodes of this directed graph are separate operations (shows dark rectangles), and arcs are use-define chains between data operations (the B command depends on the data from the A command if it consumes the data generated by the A command). It can be other dependency between operations besides data dependency (control dependency, anti-dependency), but these dependency are missing in our simple example.

To form the resulting executable code the presence of a use-define graph is not enough. It is necessary to line up all commands included in the use-define graph in a single sequence, which will subsequently be written to a linear section of memory. In this case, when alignment a sequence of commands, it is necessary to observe the requirement that if the B command depends on the A command according to the data, then it should be later than the A command in the final sequence. This process called the planning of the code[2]. In the result of planning of the code for considering case we will get pseudocode for calculating the roots of the quadratic equation is such view, shown in Listing 2.

Listing 2. Pseudocode for calculating the roots of a quadratic equation.

```
MOV 4.0        ->  v0
MOV   a        ->  v1
MOV   b        ->  v2
MUL  v0,  v1  ->  v3
MOV   c        ->  v4
MUL  v3,  v4  ->  v5
MUL  v2,  v2  ->  v6
NEG  v2        ->  v7
SUB  v6,  v5  ->  v8
SQRT v8        ->  v9
MOV 2.0        -> v10
MUL v10,  v1  -> v11
ADD  v7,  v9  -> v12
SUB  v7,  v9  -> v13
DIV v12, v11  -> v14
DIV v13, v11  -> v15
MOV v14        ->  x1
MOV v15        ->  x2
```

In Listing 2, all instructions except transmission operation operate on virtual registers (v0-v15). The order of the commands is important in this note. At any time after the execution of any instruction, we can talk about some state of the processor which characterized by the contents of the memory, as well as all the registers used.

The main part of processor's emulator is the realization of instruction's semantic, executed by the processor. I.e. that every instruction is considered as function, which transfers calculating machine from one state to another.

3. DATAFLOW PROCESSOR EMULATION

In contrast with processors with The von Neumann architecture, dataflow processors completely orderliness in command execution. I.e. if two anyone instructions can execute in parallel, (this is possible in the absence of dependence between them) then the order of their execution did not determined. We will consider the same example of the linear section, inside which the roots of the quadratic equation are calculated. The program for dataflow processor is the graph just like use-define graph showed at Figure 2. Dataflow graph, which is a program for dataflow processor, is also oriented. Nodes of this graph also are the instructions, but arcs have a completely different meaning. In the case of use-define graph, arcs were fix fact of the dependence between two operations (indicating by which variables, by which resource), in case of dataflow graph the arc is a unique data carrier that names token. Therefore, token is a data structure that storage the information about transferring data and about destination point of this data (the identifier consuming this instruction data as well as the argument number, which this data is for the instruction). Besides, the token contains additional information (the token state) allowing distinguishing instruction, which relate to different function process or to different cycle iteration, but for this example, this is not important, therefore, the description of token state will omitted.

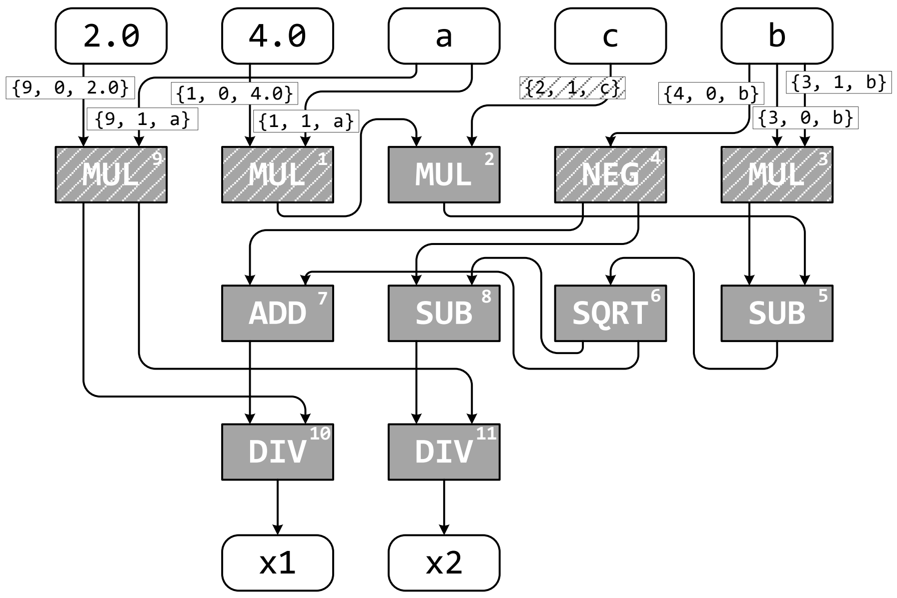

Figure 3. The dataflow graph of a linear section in the calculation of the expressions of the roots of the quadratic equation.

Figure 3 shows us dataflow graph in the initial time when none of the instructions have been completed yet. In this Figure in the arcs, we can see those tokens, which are ready and transfer at relevant instructions. For example, for the instruction 9 MUL two tokens are ready: $\{9, 0, 2.0\}$ (9 – number of instruction, 0 – number of argument, 2.0 – real number), $\{9, 1, a\}$ (9 – number of instruction, 1 – number of argument, a – real number). I.e. both arguments for the instruction 9 MUL are ready and instruction can be execute. After execution tokens, which has arguments, deletes, and at the instructions 10 DIV and 11 DIV along the output arcs will transfer tokens with the results from instruction 9 MUL.

If we will view another instruction 2 MUL we can see that it got only one ready argument (token $\{2, 1, c\}$). First argument will be formed only after execution of instruction 1 MUL.

Thereby, all of the instructions in the presented graph will be processed while needed values will not be received in the tokens at output arcs from the instructions 10 DIV and 11 DIV.

The realization of instructions semantic in the dataflow processor's emulator has no different from the same functional in the traditional processor's emulator.

Central functional element for ensure operability of the dataflow processor is as named content-addressable memory, realization of which is the appointment of this article.

Content-addressable token memory is a repository of tokens that are currently created and are being executed in the program. Wherein, if it turns out that there is all needed arguments at the content-addressable memory for anyone of commands, they immediately deletes for content-addressable memory and forward for the execution in the command. Thereby, at any given time in the content-addressable memory are only those arguments, which are still waiting missing arguments for some instructions. I.e. at conditionally the first moment of time shows at Figure 3, in the content-addressable token memory is only one token $\{2, 1, c\}$, which are still waiting its pair, whereas the commands 9 MUL, 1 MUL, 4 NEG, 3 MUL are already ready for execution and being in the command's buffer together with the sets of their arguments. At the Figure 3 shaded the token, which is in the content-addressable memory and so the commands, which are ready to execute.

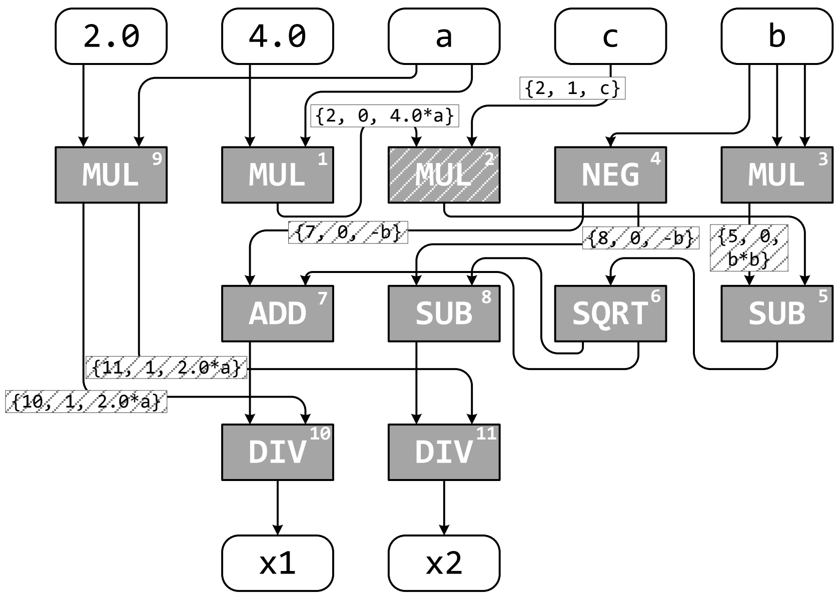

Figure 4. The dataflow graph of the linear section when calculating the expressions of the roots of the quadratic equation before executing the 2 MUL instruction.

After the execution of the instructions 9 MUL, 1 MUL, 4 HEG and 3 MUL the only instruction is ready to execute is 2 MUL. The machine state of this instruction before the execution shows at Figure 4. At this Figure we can see instruction 2 MUL which are ready to execute, and also some tokens caught it the content-addressable memory. Further execution of the program is similar and at the end, we come to the formation of two tokens on the output arcs of operations 10 DIV and 11 DIV, which will be send to their consumer instructions after that.

Listing 3. Implementation of the logic of the contest-addressable token memory.

```
set_token({CId, E, _} = T, Args, D) when (Args > 1) ->
```

```
 2    FullEntries = lists:seq(0, Args - 1, 1),
 3    ET = {E, T},
 4    case dict:find(CId, D) of
 5      error ->
 6        dict:append(CId, ET, D);
 7      {ok, Vs} ->
 8        {Es, Ts} = lists:unzip(lists:sort(Vs ++ [ET])),
 9        case Es of
10          FullEntries ->
11            {Ts, dict:erase(CId, D)};
12          _ ->
13            dict:append(CId, ET, D)
14        end
15    end.
```

At Listing 3 there is the compact logic implementation of content-addressable token memory in the Erlang programming language. All the logics realize with one function *set_token* with three arguments: T – added token (which consists of the instructions identifier CId, number of argument E, and dates), $Args$ – the number of arguments of the instruction for which the token is added, D – dictionary which token content-addressable memory is implemented. The function logic turns to the next: to extract from the dictionary all the tokens, which are associated with the key CId, and check if they close all the arguments of considered instruction (i.e. numbers of the arguments form a sequence $[0, 1, ..Args - 1]$.). If the added token is the las missing argument of the considered instruction, then we need to extract all the tokens form the dictionary and return them back in the form of list for executing the instruction (line 11). Otherwise token should be added in the dictionary tying it to a key CId (lines 6 and 13).

Note that the most simplified implementation is given. It is not considered possibility of using the token state for instructions identify from different function calls or from different cycles iteration (for completeness as a key of dictionary together with instruction identifier CId should also use the token state). In the given implementation, exception handling also is not considered. For example, sequential addition of two token in the content-addressable memory that are the same argument of the same instruction should cause the error. However, despite the incompleteness of implementation, shows here logic allows providing functionality of the content-addressable token memory, which needed for execution of the dataflow processor's program.

4. CONCLUSION

The article discusses the differences in the implementation of an executable program designed for a processor with von Neumann architecture and for a dataflow processor. The differences between the use-define graph and the dataflow graph are shown, which reflect the program logic within the linear section. The necessary element of dataflow processor's realization shown – content-addressable token memory with which is able to storage and quick search the ready-made tokens for the commands, which may transfer for execution. A compact implementation of the content-addressable token memory for instructions with an arbitrary number of iterations is given. That realization can be expanded for any program for the dataflow processor including which use cycles and function calls (including recursive). This approach to organizing of the content-addressable token memory used in the implementation of the vector processor with dataflow architecture emulator, which is developing at the JSCC RAS.

Acknowledgments. The work has been done at the JSCC RAS as part of the state assignment for the topic ... The supercomputer MVS-10P, located at the JSCC RAS, was used for calculations during the research.

REFERENCES

1. S. S. Muchnick, *"Advanced Compiler Design & Implementation."*, Academic Press (1997).
2. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *"Compilers. Principles, Techniques, & Tools."*, Addison-Wesley (2007).
3. J. Armstrong, *"Programming Erlang. Software for a Concurrent World."*, Pragmatic Programmers (2013).