Code-generator of Parallel Assembly Code for Digital Signal Processor

Nikita Bukharenko, Alexey Syschikov
State University of Aerospace Instrumentation
Saint-Petersburg, Russia
massarakh@gmail.com, alexey.syschikov@guap.ru

Abstract

This paper presents some approaches to the creation of a code generator parallel assembly for digital signal processors (DSP). Generator simplifies the creation of heavy programming constructs and provides parallel execution of DSP-core software through the use of pipeline and parallel constructions.

Index Terms: code-generator, digital signal processor, parallel code.

I. Introduction

Digital signal processors are actively applied in modern computing systems. Such systems are used in control systems, processing audio, images, and video. They are widely spread in communication devices and many other areas. For providing a high performance such processors have DSP-cores, which have specialized architecture that provides high performance in limited number of resource-intensive tasks. DSP-core can have several parallel ALU, extended tables and dimensions of registers, a set of memory subsystems, SIMD-extensions etc.

Programming languages such as C/C++ are mainly used for writing and debugging programs for modern processors instead of native assemblers. All written code is translated from the C/C++ languages to a low-level core assembler. Standard languages are oriented on traditional architectures, that's why compilers can't generate code with optimal efficiency using all possibilities of DSP architectures. To gain a well-optimized code it is necessary to write a code directly on the low-level assembly language. It's a very laborious process and to simplify a task of writing a program, it's possible to use a code-generator oriented on a certain processor. Using initially a parallel and structured programming language with a specialized assembler code-generator is possible to gain more efficient code than from translated applications written in C/C++. The task of generation efficient assembly code which actively uses parallelism and other features of DSP-kernel is the purpose of this work.

Usage of parallel assembly generator gives optimized, parallel, high-performance code without a necessity of manual parallelizing assembly code.

II. Solution of the problem

A. Overview

The code generation from high-level representation plays an important role in the computer science. It's difficult to imagine a programming environment without a visual

With the financial support of the Ministry of Education and Science of the Russian Federation

editor, when you move objects on the form and generator writes a code with the desired characteristics in the correct location.

Writing web-based applications, creating software with GUI for desktop computers, developing applications for mobile phones is extremely difficult without graphic editors that make a routine job of generating code.

A low-level assembly code is generated by all compilers, but they work with traditional text-based languages like C/C++. At the same time, a development of solutions in the field of digital signal processing tasks is at the turn of a work of mathematicians, engineers and programmers. Traditional text languages don't fit well for a design and specification of algorithms. Provision of a high level language, preferably visual, to developers, which on the one hand allows designing the algorithm visually, and on the other hand generating an effective low-level code, will allow solving these problems in a natural way, without translating them in an uncomfortable text language.

Among the successes of systems that implement the visual programming approach, it's necessary to note systems such as Simulink, LabView and SCADE. Simulink is based on the Matlab execution environment and performs schemes within the Matlab package, programs written in LabView run in its own standalone and portable runtime environment and only SCADE provides full code generation, but only to the C language.

Among these examples it can be seen that the code generation is not new, in fact, this is the main way to automate the process of obtaining native core code. As it is seen from the examples above, there are no tools, combining the possibility of a high-level programming and generating effective low-level code for specific architectures of DSP-cores. The solution of this task is presented in this paper.

B. Solution of the problem

Most of signal processors have similar architectures, which mean that the developed approach can be applied to a wide class of signal processors. In this paper as an experimental platform we chose the microprocessor MC-24 [1] with a 32-bit central processing unit with RISC-architecture and the high performance co-processor for digital signal processing (DSP). DSP-core supports data processing with fixed and floating point, provides information processing with a variable data format from bit formats to standard data formats with floating-point according to IEEE754 standard.

1) Solution concept

For solving the problem of writing parallel assembly code-generator the following approaches are applied:

- Templating. Assembly instructions are presented like separate text templates;
- Parallelization. Providing a parallel execution of operations on parallel streams (only available for DSP-core);
- Optimization of registers usage. The algorithm of selection and purification of the core registers;
- Optimization of a memory usage. The algorithm of reusing data stored in a memory;

The source code is presented in the form of a scheme at the VPL programming language [2, 3]: it describes objects and their relationships (objects graph), actually in a data-flow-like scheme. In this language model there are number of significant differences from the classic data-flow, but in the borders of this task it's not significant. In the

scheme it is described a computational operation of a component (schema object), its properties, parameters, and links to other objects.

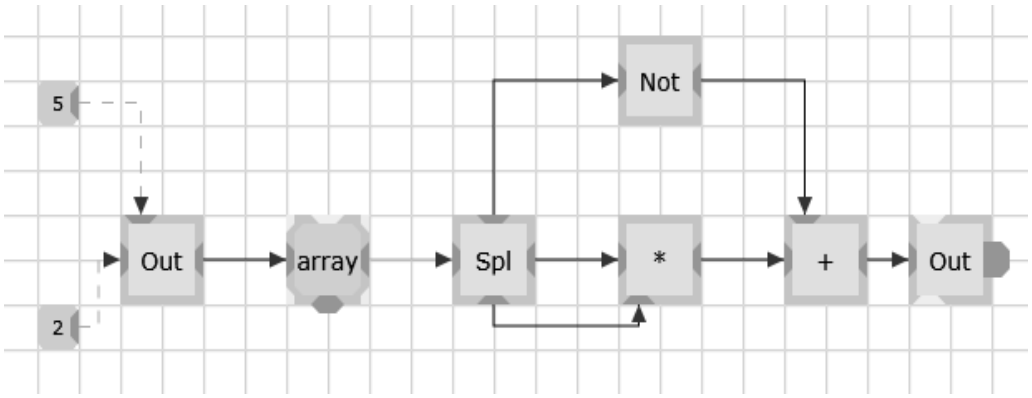

Fig. 1. Sample input scheme program

Example at the Fig.1 includes the following objects:

- grey squares – constants
- spl – “multiplicator” of data
- "*" – object, that multiplies
- "Not" – object thst makes inversion
- “+”– object that specifies computational operation “addition“
- “array” is a representation of memory on a scheme
- “Out” – object that works with memory

Relationships (links) between components are specified with arrows. The scheme is supplied to the code-generator through the intermediate representation that contains all options, links, properties and descriptions of scheme components.

```
<?xml version="1.0" encoding="UTF-8"?>
<program id="p0" comment="" mainSection="i0">
  <group id="i0" subtype="section" comment="">
    <nodes>
      <terminal id="i5" subtype="splitter" comment="Spl" tag="" isPermanent="False" execCost="0">
        <port id="i6" subtype="simple" tag="" link="i31" portNum="1" />
        <port id="i7" subtype="simple" tag="" link="i44" portNum="2" />
        <port id="i8" subtype="simple" tag="" link="i46" portNum="3" />
        <port id="i9" subtype="simple" tag="" link="i75" portNum="4" />
      </terminal>
      <terminal id="i14" subtype="functional" comment="" tag="" isPermanent="False" execCost="0"
        <port id="i15" subtype="simple" tag="" link="i47" portNum="1" />
        <port id="i16" subtype="simple" tag="" link="i51" portNum="2" />
        <port id="i18" subtype="simple" tag="" link="i73" portNum="4" />
      </terminal>
```

Fig. 2. Intermediate representation code of a scheme

At the first stage we parse a scheme and create a linked graph, which contains all the information about objects and their relationships, properties and parameters.

We construct a schedule for all scheme objects in accordance with the time of their execution and data dependences. At the next stage we determine a parallel execution possibility of operations based on available threads and, if necessary, rearrange the schedule.

Each component maps to a correspondent code template. At the next step we collate registers in template with registers from register table. If it is necessary we generate the code of loading data from the memory to registers or unloading the registers to the memory. As a result, on the output there is a working program for the selected core.

To use the code-generator we have to set several parameters:

- Scheme file;
- Template library (for both cores);
- Operation mode of code-generator (RISC or DSP);
- The starting available register;
- The final available register;
- The starting available memory address;
- Flag of parallelism possibility, “1” для permission, “0” for prohibition (it doesn’t matter for RISC-core).

2) Templating

Representation in assembly language terms is formed for all types of scheme elements. Each core has its own instruction set which means that we have to create templates for both of them. Moreover, the chosen processor MC-24 in DSP-core command format depends on the operational units (ALU), on which will be performed an operation. That’s why it’s possible to specify a template for the various operating units within the core.

An example of a subtraction template for RISC-core:

```
entity llsub [risc] is
    sub &regout1,&reginx1,&reginy1
entity llsub end.
```

An example of the same operation template for the first ALU of DSP-core:

```
entity llsub [dsp1] is
    sub &reginx1,&reginy1,&regout1
entity llsub end.
```

Reserved word "entity" is intended to designate a template in a text file. The next word is an identifier that denotes the operation at the program scheme.

There is an accessory to the core of the microprocessor in square brackets and belonging to a group of operands for parallel operation (only for DSP-core). A parallel execution of two computational operations in one VLIW command of DSP-core is possible only if they are executed on different operating units (OU). For example it is impossible to execute operations AND and OR, as they both have to be performed by the same logical unit LU. Computational operations can be divided into two types - OP1 and OP2 depending on their operational executing unit. The belonging to different groups makes it possible to execute two operations in a single command. The first type (OP1) includes operations, performed by the arithmetic and logical unit (AU, LU), the second type (OP2) - operations, performed by the multiplier, shifter MS. In the template the mark "1" denotes the identity of the first type (OP1), the mark "2" respectively to the second type (OP2).

The word "sub" means the assembly operation command, for which a template is made. "&reginx1" indicates that the parameter is the input register corresponding with the tag "x" of the scheme operation port and with the number "1". "&reginy1" means the same thing except for the tag (“y” instead of “x”). The tag is required for proper binding

of two components and two operations. "&regout1" is a parameter for the output register with an empty tag and the number "1".

A template can contain the code of any size. For example, the operation “test of equality” of the two values for RISC-core has the following description:

```
entity llequal [risc] is
    beq &regin1,&regin2,wait0
    li &regout1,0x0
    j loop0
wait0:
    li &regout1,0x1
loop0:
entity llequal end.
```

3) Parallelism

A parallelization is possible if two operations have different types and do not have data dependency. The parallelism of the DSP-core is ensured by the existence of multiple operating units (OU) in a single core. In this example, DSP-core MC-24 has 3 OU. If there are enough general-purpose registers to perform both operations and they can be performed in parallel, then the code parallelization occurs. Otherwise, threading does not occur and commands are executed sequentially. In the DSP-core of MC-24 running of parallel commands is implemented through the usage of VLIW instructions which can contain up to three concurrent commands.

An example of instructions to perform two operations simultaneously:

```
mpss R2,R4,R10 or R8,R6,R12
```

An example of the same instructions to perform two operations sequentially:

```
mpss R2,R4,R10
or R8,R6,R2
```

The command "mpss" belongs to the type OP2 and multiplies two integers in the "short" format. The command "or" designed to perform a logical OR in the format "short" belongs to OP1.

There are enough registers in the first case and the flag of parallelism is set to "1", while in the latter case, there are also enough registers, but the parallel flag is set to the state "0", hence, a parallelism is not realized.

The system of instruction and flexible address modes allow the DSP-core efficiently implement signal processing algorithms. Execution time is minimized by the usage of the software pipeline and high-level instruction implementing several parallel computational operations and transfers.

4) Optimization of registers

Each microprocessor core, RISC and DSP have a set of general-purpose registers. The source data and the results of all operations in the ALU are stored in registers. The code-generator has a configurable number of registers within existing (e.g. registers from the first to the eighth of sixteen 32-bit registers).

The code-generator creates a virtual table of registers with a specified number of cells (let’s say 8, as mentioned above), that means that in the beginning of the code-generator work we have 8 free registers for commands data preparation.

During the processing the code-generator register table is gradually filling and there is a need to free registers. After giving out all necessary records to a component and

generating its code, we clean the input registers and mark them as free in the table of registers. When all the registers in the table are marked as busy, we free any register that is not used now by using a designated algorithm, store its value in memory and mark as free.

It is possible to use various algorithms for freeing register from the register table. In the given code-generator the random selection is implemented.

Each register contains its own serial number, the flag of occupation in the current moment, the flag of possibility of release and the link identifier of connection with another component.

An example of the register table at the beginning of the generator work:

Value:	2	3	4	5
Busy:	False	False	False	False
Link:				
Now:	False	False	False	False

Fig. 3. Table of registers at the initial state

The "Value" indicates the ordinal number of the register in the table, the flag "Busy" shows if the register can be released. If the flag is "False" then the register can be released.

The field "Link" shows what the identification number of a link between the two components. The "Now" is used to display the status of employment register for the currently generating command.

An example of the register table during the processing of the generator:

Value:	2	3	4	5
Busy:	False	True	True	True
Link:		241	274	273
Now:	False	False	True	True

Fig. 4. Table of registers during the work of registers

The "Value" does not change throughout the processing of the generator

The program work is impossible if in the input parameters of the generator the number of registers is less than the number of registers required to perform each operation. In other words if you set in the input parameters only two available registers, but there is some operation that requires three registers, the work of the generator will not be possible.

5) Work with memory

All objects in the scheme are connected by links. Parameters, properties, and object described in the link diagram.

```
<links>
  <link subtype="read-erase" source="i50046" sourcePort="112" target="i50082" targetPort="2"
  <link subtype="read-erase" source="i50058" sourcePort="113" target="i50082" targetPort="6"
  <link subtype="read-erase" source="i50058" sourcePort="114" target="i50097" targetPort="2"
  <link subtype="read-erase" source="i50070" sourcePort="115" target="i50097" targetPort="6"
  <link subtype="read" source="i50001" sourcePort="" target="i50046" targetPort="2" defDataAr
  <link subtype="read" source="i50003" sourcePort="" target="i50058" targetPort="4" defDataAr
  <link subtype="read" source="i50005" sourcePort="" target="i50070" targetPort="6" defDataAr
  <link subtype="read-erase" source="i50097" sourcePort="10" target="i50007" targetPort="6" c
  <link subtype="read-erase" source="i50082" sourcePort="10" target="i50007" targetPort="4" c
  <link subtype="read-erase" source="i50007" sourcePort="10" target="i50043" targetPort="1" c
  <link subtype="read" source="i50126" sourcePort="" target="i50043" targetPort="4" defDataAr
  <link subtype="read-erase" source="i50046" sourcePort="36" target="i50007" targetPort="0" c
```

Fig. 5 An example of the intermediate representation of the program scheme that specifies interconnections.

A related graph obtained after parsing the scheme contains all the necessary information about the data interconnection. Links confronted with the addresses in the internal memory of the processor (either RISC or DSP). In each register in the table of registers recorded the identification number (ID) of link to know for what input or output value it was issued. If we have to save a value contained in the register during the work of the generator, the value is store to the address specified in the link. If we want to get the value, then we just unload it from memory.

For the DSP-core of the MC-24 processor working with memory is possible only by usage the address registers. There are eight address registers (A0-A7). To write a value to a specific address the address must be loaded in the address register:

```
move 0x00000044,A0
```

Then value must be written in the memory at that address:

```
move R2, (A0)
```

For the usage of offsets in memory addressing there is a set of offset registers (I0-I7). To use them we need to load an offset value in the offset register:

```
move 0x00000004,I0
```

Then add to the address at the time of loading in memory:

```
move R2, (A0+I0)
```

The initial memory address of the RISC core is 0x00000000, for DSP-core is 0x80000000. During the development the optimization of address and offset registers is not implemented yet, but of course, it's necessary for increasing the efficiency of the generated code and it is in plans for the further development.

C. Implementation

Using the described approach to the construction of a code-generator, we can get a working program for RISC-core assembly language or for the DSP-core assembly (sequential or parallel assembly) language.

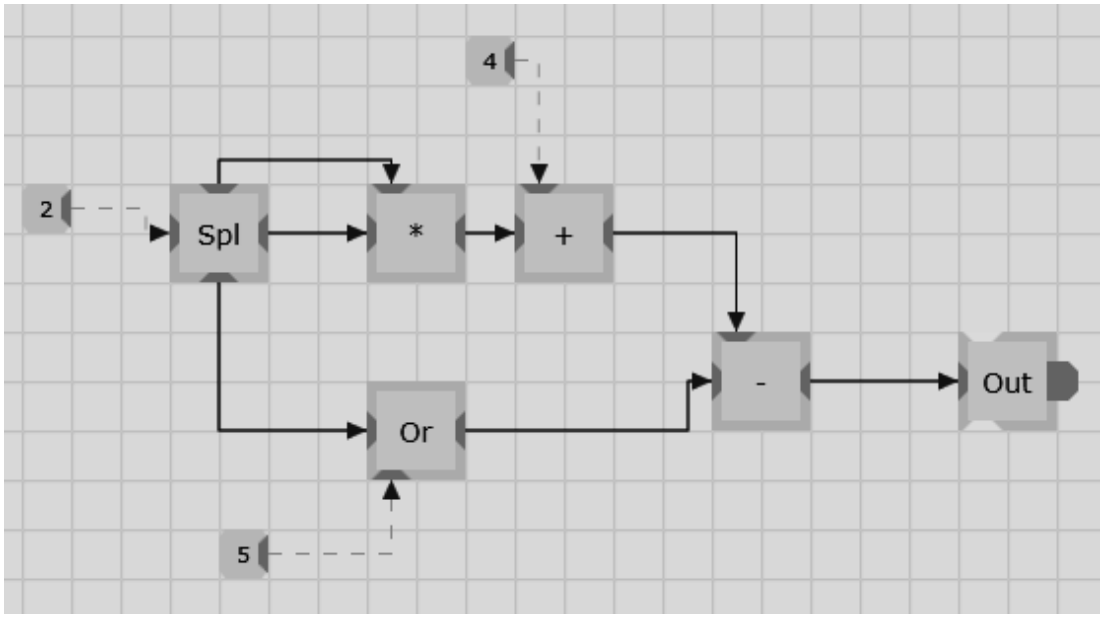

Fig. 6. Program's scheme for code generation

Generated software implementations of the scheme are presented below:

<table>
<tr><td>Program for RISC-core</td><td>Program for DSP-core</td></tr>
<tr><td>
<pre>
li $2,0x2
li $3,0x5
li $4,0x4
sw $2,0x80000014
or $5,$2,$3
mul $3,$2,$2
add $2,$3,$4
sub $3,$5,$2
sw $3,0x80000990
</pre>
</td><td>
<pre>
move 0x2,R2
move 0x5,R4
move 0x4,R6
move 0x00000014,A2
move R2,(A2)
mpss R2,R2,R8   or R2,R4,R10
add R8,R6,R4
sub R10,R4,R6
move 0x00000044,A2
move R6,(A2)
</pre>
</td></tr>
<tr><td colspan="2">Results of execution</td></tr>
<tr><td>8-7=1</td><td>8-7=1</td></tr>
<tr><td>
The result is written into the register $3
<table>
<tr><td>0</td><td>zero</td><td>00000000</td></tr>
<tr><td>3</td><td>v1</td><td>00000001</td></tr>
<tr><td>6</td><td>a2</td><td>00000000</td></tr>
<tr><td>9</td><td>t1</td><td>00000000</td></tr>
<tr><td>12</td><td>t4</td><td>00000000</td></tr>
</table>
Next to the memory by address 0x80000990
<table>
<tr><td>80000990</td><td>01 00 00</td></tr>
<tr><td>800009A1</td><td>00 00 00</td></tr>
</table>
</td><td>
The result is written into the register R6
<table>
<tr><td>R04.L: R05 R04</td><td>0000 0008</td></tr>
<tr><td>R06.L: R07 R06</td><td>0000 0001</td></tr>
</table>
Next to the memory by address 0x00000044
<table>
<tr><td>00000040</td><td>00000000</td></tr>
<tr><td>00000044</td><td>00000001</td></tr>
<tr><td>00000048</td><td>00000000</td></tr>
</table>
</td></tr>
</table>

III. Conclusion

The developed approach allows us to simplify the creation of complex programming constructions by generating parallel assembly code instead of the complicated manual programming of parallel assembly instructions. The target platform used in the development is the MC-24 processor-based platform "Multicore". The system of instructions and flexible address modes of DSP-core ELcore-24™ can effectively implement the signal processing algorithms. Execution time is minimized through the use of the software pipeline and high-level instruction implementing several parallel computational operations and transfers.

We are working on adding functionality to the code-generator to process control constructions such as conditional branching and loops.

References

[1] Signal processor chip 1892ВМ2Я (MC-24). http://multicore.ru/index.php?id=47.

[2] Syschikov A. The parallel programming technology of heterogeneous systems on a chip. Scientific session SUAI: Sb. paper.: B 3 . P. I. Engineering sciences /SUAI. SPb., 2008, p.133-139.

[3] Ivanov V., Sheynin Y, Syschikov A. Programming model for coarse-grained distributed heterogeneous architecture Proceedings of the XI Symposium on the issue of redundancy in information systems /Edited by prof. Kruk E.A. – SPb: SUAI, 2007, p.246-250.