Eindhoven University of Technology

MASTER

Mistral2 to FACTS

Arts, B.M.H.

Award date:
1999

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Master's Thesis

Mistral2 to FACTS

B.M.H. Arts

carried out: December 1998 - August 1999 at the Philips Research Laboratories, Eindhoven

supervisors PRLE: ir. B. Mesman/ir. N.G. Busa
prof.dr.ir. J.L. v. Meerbergen

supervisors TUE: dr.ir. C.A.J. v. Eijk
prof.dr.ing. J.A.G. Jess

The Faculty of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of Master's Theses
Preface

Well, finally an end has come to my study period. This thesis finishes exactly six years of reading, learning, sweat, stress, relief, fun, laughing, partying and, generally spoken, enjoyment.

I would like to thank Koen van Eijk for his outstanding supervising and great technical knowledge, both general and at FACTS side. I would like to thank Bart Mesman for being my (almost) daily coach and for his creative and inspiring way of thinking and acting, which helped me a lot. Also I would like to thank Natalino Busa, for helping me with all Mistral2 problems I encountered.

Furthermore I would like to thank my general supervisors, professor Jess and professor v. Meerbergen, who gave me the opportunity to do my work at the Philips Nat.Lab. in Eindhoven.

Last but not least, I would like to thank my girlfriend Manon for being a very lovely and understanding person, even at moments that I did not have much time to spend with her and that I was a bit numb due to all the heavy-minded work. Finally, I owe the trumpet, the Instrument of the Gods :-), many thanks for letting me express musical and mood-controlled thoughts.

greetz,

Bas Arts

--- / |
| (____|____) \`
Abstract

In cooperation with Philips, the company Frontier Design has developed a system that synthesizes a DSP architecture, starting from an algorithmic description of the behaviour of the processor. This synthesis system is called Mistral2.

Code generation is part of the process of generating assembly code for processors. This includes resource assignment, register binding and scheduling. A software infrastructure called FACTS for doing research in this area has been set up at the Eindhoven University of Technology (TUE). FACTS is a code generation tool in which constraint analysis techniques developed at the TUE and the Nat.Lab. play a central role.

To make sure that the techniques applied in FACTS are able to cope with relevant industrial applications, it is necessary to obtain industrially relevant code generation benchmarks for FACTS. Therefore it would be very useful if the Mistral2 frontend could be used to generate these benchmarks.

We have designed a software interface from Mistral2 to FACTS. This interface is capable of reading binary information created by Mistral2 v1.4 rev2 and transferring the necessary information to a textual input format for FACTS.

The interface has been tested using a broad range of Mistral2 examples. On the one hand, the functionality of the interface has been visually verified using small examples. The small examples can be divided in four categories: basic blocks, hierarchical examples, loops and conditional constructs. On the other hand, the robustness of the interface has been verified using larger examples.

The interface is capable of dealing correctly with basic blocks, hierarchical examples, loops and larger examples. The generated output for examples that contain conditional constructs does not contain the right functionality as present in the Mistral2 input description.

Future work includes the improvement of the way the interface deals with conditional constructs. Furthermore, it is very interesting to translate the results of FACTS back to Mistral2. In order to do this, the results of FACTS must be translated to Mistral2 pragmas that can control the Mistral2 scheduler.
Contents

1 Introduction

2 Mistral2, FACTS and Interfacing
   2.1 Mistral2
      2.1.1 Design Flow
      2.1.2 RTL Description
   2.2 FACTS
      2.2.1 Constraint Analysis
   2.3 Interfacing
      2.3.1 Data Conversion
      2.3.2 FACTS Input Information

3 Basic Blocks
   3.1 Mistral2 Concepts
      3.1.1 Theory
      3.1.2 Example
   3.2 FACTS Concepts
      3.2.1 Theory
      3.2.2 Example
   3.3 Interfacing
      3.3.1 Architecture
      3.3.2 RTG Information
      3.3.3 Examples

4 Hierarchy
   4.1 Mistral2 Concepts
      4.1.1 Theory
      4.1.2 Example
   4.2 FACTS Concepts
      4.2.1 Theory
      4.2.2 Example
   4.3 Interfacing
      4.3.1 RTG Information
      4.3.2 Examples
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.2.1</td>
<td>Mistral2 Input Description</td>
<td>60</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Interface Output for 'Blocks1'</td>
<td>61</td>
</tr>
<tr>
<td>B.2.3</td>
<td>Interface Output for 'Blocks2'</td>
<td>63</td>
</tr>
<tr>
<td>B.3</td>
<td>Loops</td>
<td>65</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Mistral2 Input Description</td>
<td>65</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Interface Output for 'Simple Loop'</td>
<td>66</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Interface Output for 'Loop'</td>
<td>68</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Interface Output for 'Loop in Loop'</td>
<td>70</td>
</tr>
<tr>
<td>B.4</td>
<td>Conditional Constructs</td>
<td>72</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Mistral2 Input Description for 'IF'</td>
<td>72</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Interface Output for 'IF'</td>
<td>72</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

One of the aspects of electrical engineering is computer aided design of digital circuits. Due to the increasing complexity of today's digital circuits, high level synthesis more and more plays an important role within their design. High level synthesis consists of a number of steps. One of those steps is code generation.

Code generation is part of the process of generating assembly code for processors. This includes resource assignment, register binding and scheduling. A software infrastructure called FACTS for doing research in this area has been set up at the TUE. FACTS is a code generation tool in which constraint analysis techniques developed at the TUE and the Nat.Lab. play a central role.

In cooperation with Philips, the company Frontier Design has developed a system that synthesizes a DSP architecture, starting from an algorithmic description of the behaviour of the processor. This synthesis system is called Mistral2.

To make sure that the techniques applied in FACTS are able to cope with relevant industrial designs, it is necessary to obtain industrially relevant code generation benchmarks for FACTS. Furthermore, by using these kind of benchmarks, industry can see that research in the area mentioned above is valuable. We use the Mistral2 frontend to generate these benchmarks. This frontend provides a register transfer level (RTL) description of the processor. This RTL description is the input for the code generation part.

Mistral2 is a commercial software product. Therefore, the techniques within FACTS can not be implemented directly into Mistral2. In addition to this, Mistral2 and FACTS use a different representation of information. Our goal is to design a software interface that can be used to transfer information from Mistral2 to FACTS. Transferring data from Mistral2 to FACTS brings along a number of problems, that are conceptually related to Mistral2 and FACTS. The issues that play a role are basic blocks, hierarchy, loops, arrays and conditional constructs. We use a pragmatic approach to solve the information transfer problems. This approach called prototyping is shown in Figure 1.1.

Chapter 2 introduces Mistral2 and FACTS and discusses interfacing in general terms. The problems that are stated above, are thoroughly discussed: basic blocks (Chapter 3), hierarchy (Chapter 4), loops (Chapter 5), arrays (Chapter 6) and conditional constructs (Chapter 7). Chapter 8 shows some examples. Finally, Chapter 9 draws some conclusions.
Design framework of interface from simple examples

Create overview of global problems

Research next problem

Solve this problem

Implement solution in interface

Figure 1.1: Problem Approach
Chapter 2

Mistral2, FACTS and Interfacing

Section 2.1 gives an introduction to Mistral2 and discusses the Mistral2 RTL description. An overview of FACTS is given in Section 2.2. Data conversion and the FACTS input information are discussed in Section 2.3. As stated in Chapter 1, specific issues related to Mistral2 concepts and FACTS concepts, like loops and hierarchy, are discussed in the following chapters.

2.1 Mistral2

Mistral2 is a system that synthesizes a DSP processor architecture. Starting from an algorithmic description of the behaviour of the processor, Mistral2 generates a custom DSP processor containing a data path and a multibranch micro-code based controller. Its library contains standard computational units (such as an ALU, MULT and MAC), memory units (such as RAM and ROM) and I/O interface units. Furthermore, it is possible to create and add self made application specific units (ASU’s).

Figure 2.1 shows the template architecture of a Mistral2 processor. We do not discuss this template architecture.

![Mistral2 Architecture Template](image)

Subsection 2.1.1 shows the design flow of Mistral2. As stated in Chapter 1, the Mistral2 front end provides an RTL description of the processor. The Mistral2 RTL description is discussed in Subsection 2.1.2.
2.1.1 Design Flow

Figure 2.2 shows the global design flow of the Mistral2 system [FD98]. This design flow contains the following steps:

- **Compilation**
  In this step one can compile the algorithmic description. As Figure 2.2 shows, two different input languages can be chosen from: DFL (Data Flow Language) and C. The compilation step translates the input description into a signal flow graph (SFG).

- **Architecture Definition**
  This step creates a description of the processor architecture and its controller. The description can be complete (fixed architecture) or nearly complete. In the latter case, some parameters can still be free, such as the width of the data busses.

- **Algorithm Mapping**
  In this step, the algorithm is mapped onto the architecture defined in the previous step. The result of this step is a list of fundamental operations on the architecture. These operations are called register transfers (RT’s). The set of these operations and their dependencies is called the register transfer graph (RTG).

- **Scheduling & Register Assignment**
  This step maps the operations of the RTG onto the time axis, which is divided into integer machine cycles. The register assignment part computes lifetimes of variables which have to be stored in the register files and defines the size of those register files. This can not be done during algorithm mapping. In case of a completely fixed architecture, this step has to check if the register files are all large enough. If the architecture is not completely fixed, an attempt is made to minimize the register area. Due to the interaction between scheduling and register assignment, these two parts are merged into one step.

- **Microcode Generation & Controller Synthesis**
  Microcode generation transforms the set of scheduled register transfers into a symbolic description of the processor controller. From this description, controller synthesis generates a controller that is capable of sequencing the RT’s.

- **Netlist Generation**
  This step constructs a structural description of the generated processor either in VHDL (VHSIC Hardware Description Language) [PE91] or Verilog [TH96].

Figure 2.2 also contains pragmas. Pragmas are designed to let the user control the Mistral2 compiler upto a certain point. For example, by specifying a pragma, Mistral2 can be forced to map a certain operation onto a certain module, or Mistral2 can be forced to unfold a part of a folded loop.

The Scope box indicates which part of the design flow is interesting for the interface. This will be explained later on.

2.1.2 RTL Description

The RTL description represents the behaviour of the processor. An RT specifies which resources must be activated and which operation is executed. An RTG is a graph that contains RT’s (nodes) and possible data edges and sequence edges between RT’s. In this subsection, only the structure of an
Figure 2.2: Design Flow of Mistral2
RTG and the timing aspects are discussed. Other details, like resource usage of RT's, are discussed in the next chapter.

An RT has input ports and output ports. To each port, a variable is connected. Each port can be connected to more than one data edge and/or sequence edge. Within Mistral2, an RT is always executed in one cycle.

A data edge from $RT1$ to $RT2$ indicates that $RT1$ produces a value that is consumed by $RT2$. A sequence edge from $RT1$ to $RT2$ specifies a minimum distance in (cycle) time between $RT1$ and $RT2$. An example of the structure of an actual Mistral2 RTG is shown graphically in Figure 2.3. This figure shows a sequence edge, indicated by the dotted line, between $RT3$ and $RT4$. The other lines represent the data edges.

![Figure 2.3: A Mistral2 RTG](image)

We can denote a sequence edge from $RT1$ to $RT2$ with a value $d$ as follows:

$$RT1 \xrightarrow{d} RT2$$

(2.1)

This means that $RT2$ can start at the earliest $d$ clock cycles later than $RT1$. The semantics of this relation are

$$s(RT2) \geq s(RT1) + d$$

(2.2)

where $s(RTx)$ is the starting time of $RTx$. Using negative weights, it is possible to fix the number of cycles between two RT's. Let's take a look at the example shown in Figure 2.4. In this example, $RT2$ consumes a value that is produced by $RT1$. However, a sequence edge is present from $RT2$ to $RT1$ with a value $-1$. This means that

$$s(RT1) \geq s(RT2) - 1$$

(2.3)
Figure 2.4: A Sequence Edge with Negative Weights

\[
\text{or: } \quad s(RT2) \leq s(RT1) + 1
\]  \hspace{1cm} (2.4)

This means that \(RT2\) must be executed at most one clock cycle after \(RT1\).

### 2.2 FACTS

Currently, the research tool Facts is mainly focused on code generation. In particular, it can be used to perform scheduling and register binding. Scheduling is the problem of determining the start times of all operations (RT's in the context of Mistral2), while register binding concerns the problem of deciding in which register a certain value is stored. A schedule is called feasible if it does not violate any constraint imposed on it.

#### 2.2.1 Constraint Analysis

A central issue in FACTS is schedule freedom. If the scheduler has little freedom in scheduling the operations, it is easy to find a feasible schedule or to find that no feasible schedule exists. Constraint analysis influences the freedom of the scheduler. We do not go into exact details of constraint analysis, but we give a short description of the way it works.

In FACTS, two types of constraints are analysed [BM99]:

- **Functional Resource and Timing Constraints**
  
  These constraints are imposed by data dependencies, sequence dependencies and functional resource usage.

- **Register Constraints**
  
  These constraints are related to properties of register file models, in connection with variables that have to be stored. These constraints cannot be expressed using regular functional resource or timing constraints, and require dedicated constraint analysis techniques.

The results of constraint analysis are stored in a central data structure called *distance matrix* [BM98]. This matrix has \(N\) rows and \(N\) columns, in which \(N\) stands for number of operations. In the FACTS implementation, the first operation in a data flow graph (DFG) is called the *source*. The last operation in a DFG is called the *sink*. The term 'DFG' will be explained in Chapter 3. The
source and sink are dummy operations that do not use a resource and have a delay of zero cycles. Each element \((n_{\text{row}}, n_{\text{col}})\) of the matrix contains a value \(\min_{\text{dif}}\) which is the minimum difference between the starting times of operations \(n_{\text{row}}\) and \(n_{\text{col}}\). Formally speaking:

\[
s(\text{Op}(n_{\text{col}})) - s(\text{Op}(n_{\text{row}})) \geq \min_{\text{dif}}
\]

This matrix gives us for example the opportunity to extract the values for ASAP (As Soon As Possible) and ALAP (As Late As Possible) scheduling for each operation.

\(\text{distance}_{\text{matrix}}(\text{source}, x)\) contains the minimum distance between \(s(\text{source})\), which is 0, and \(s(\text{Op}(x))\). In other words, it contains the asap start time of \(\text{Op}(x)\). \(\text{distance}_{\text{matrix}}(y, \text{source})\) contains the minimum distance between \(s(\text{Op}(y))\) and \(s(\text{source})\). Summarising:

\[
\text{asap}(\text{Op}(x)) = \text{distance}_{\text{matrix}}(\text{source}, x)
\]

and

\[
\text{alap}(\text{Op}(y)) = -\text{distance}_{\text{matrix}}(y, \text{source})
\]

The distance matrix is continuously updated during constraint analysis. The goal of constraint analysis is to fix the real distance from one operation to another as precisely as possible. Because it can not efficiently be done in an exact way, it is done in a heuristic way. However, the results are more precise than the normal ASAP-ALAP results.

### 2.3 Interfacing

This section discusses interfacing. Subsection 2.3.1 discusses the issue of data conversion. Subsection 2.3.2 discusses the backgrounds of the FACTS input information. Detailed information about the different parts of interfacing and about the semantics of the FACTS input information can be found in Chapters 3 to 7. Finally, a manual about the interface itself can be found in Appendix A.

#### 2.3.1 Data Conversion

Information can be transferred from Mistral2 to FACTS in a limited number of ways. A graphical overview of these possibilities is shown in Figure 2.5. This figure shows the four possibilities that are present (arrows 1 to 4):

- Mistral2 Binary Data Structure \(\xrightarrow{1}\) FACTS Binary Data Structure
- Mistral2 Binary Data Structure \(\xrightarrow{2}\) Textual Intermediate Format
- Mistral2 Textual RTG Information \(\xrightarrow{3}\) FACTS Binary Data Structure
- Mistral2 Textual RTG Information \(\xrightarrow{4}\) Textual Intermediate Format

All possibilities have advantages and disadvantages. Let’s start at the destination side. Generating a textual format is highly preferred over generating a binary data structure. First of all, this textual format can be modified by hand. Therefore new benchmarks can be created as an input for FACTS independent of Mistral2. Using a binary format this would be (almost) impossible. Furthermore, if a binary format is used, the interface is very sensitive to changes. At FACTS side much research and development is still being done, which causes significant, structural changes. In addition to that, an
advantage of using a textual format is that this intermediate format may be used as an input format for other tools. Arrow B indicates that a translation step from the intermediate format to the FACTS binary data structure exists.

At the source side, things are slightly different. First of all, only sporadically small changes are made at the Mistral2 side. Hence the binary data structure will not change significantly. Furthermore, the textual RTG information is generated by a tool within Mistral2, indicated by arrow A. This tool uses the information in the data structure after the 'Scheduler & Register Assignment' step, shown in Figure 2.2. At this point, there are two possibilities. Either we modify this tool in such a way that the generated output of the tool is accepted by FACTS, or we design a parser that reads the RTG text and translates it to the intermediate format. The latter possibility introduces the redundant step of transforming text into text. In addition to that, writing a parser from scratch takes much more effort than modifying an existing tool. Furthermore, it is not sure whether or not the translation tool extracts all information from the Mistral2 binary data structure needed by FACTS. Hence it is much more convenient to generate the intermediate format directly from the Mistral2 binary data structure. For this is done after the 'Scheduler & Register Assignment' step, the latter part of the Mistral2 design flow shown in Figure 2.2 is not part of our scope.

Finally, another issue attracts attention. Let’s look at option 1, which is to converse data from the Mistral2 binary data structure to the FACTS binary data structure. In this case, in order to create new benchmarks for FACTS Mistral2 must be used again. Otherwise the binary format at FACTS side must be modified, which is inconvenient. The fact that Mistral2 is a commercial tool, implies that people
who want to create new examples for FACTS, must have a license for Mistral2. This is another reason
not to choose for option 1.

Although focused on Mistral2 and FACTS, the backgrounds of data conversion can be applied in
a broader range.

FACTS to Mistral2?

Another data conversion issue is the translation of FACTS information back to Mistral2. In order to
do this, the results of FACTS must be translated to Mistral2 pragmas that can control the Mistral2
scheduler. Within this project, we will not consider this reversed translation.

2.3.2 FACTS Input Information

Subsection 2.3.1 has explained why it is smart to let the interface generate a textual format. Regarding
the FACTS input formation, the following issues have to be taken into account:

• Readability
  One should be able to intuitively read the input description. In that way, new benchmarks can
  be created and modified. Furthermore they can be validated by visual inspection.

• 'Independent' of FACTS
  Due to the fact that FACTS is a research vehicle, significant changes are frequently made to its
data structure. Therefore it is useful to design an input format that is (sort of) independent of
FACTS. Otherwise the input format must be changed every time the data structure is changed.
Furthermore, an independent format might be used as an input for other tools.

• Flexibility
  From a FACTS point of view, the input information must not be restricted by the information
provided by Mistral2. By adding information, or by modifying or deleting parts of Mistral2
related input information, a broader range of benchmarks can be transferred to FACTS. On the
other hand, getting rid of too much information can lead to meaningless benchmarks. Further­
more, if we want to put back the results of FACTS into Mistral2, the core information provided
by Mistral2 must be preserved.

• Hierarchy
  Is it useful to have hierarchy in the input format? First of all, hierarchy adds a certain structure
to the input information, which makes it more readable. In addition to that, the Mistral2 RTG
information contains hierarchy too. Hence, by using a hierarchical input format the original
information structure provided by Mistral2 can be maintained. Chapter 4 discusses more details
about hierarchy.

Difference in Architecture

Subsection 2.3.2 has discussed the issue of flexibility. It should be possible to modify the Mistral2
architecture without loosing functional information. First of all, it is possible to manipulate the func­
tional resources. For example, if an arithmetic logical unit (alu) is present that executes an addition
and a subtraction, the alu can be replaced by an adder and a subtracter.
It is also possible not to include any information about multiplexers. They are always coupled to registers of operational units, which in turn are coupled to those operational units. Hence, the usage of the multiplexers is dependent on the usage of their respective registers.

Leaving out register usage and/or register binding information is the next step. However, this will influence the schedule decisions made by FACTS if the register information is a bottleneck for the scheduling of the RT's. In that case, the results obtained by FACTS will not be correct, regarding the actual architecture.

Within this project, we will not consider the consequences of leaving out or changing register information.
Chapter 3

Basic Blocks

The previous chapters have introduced Mistral2 and FACTS, and some issues about interfacing have been discussed. The following chapters discuss specific interfacing problems. This chapter discusses basic blocks. Section 3.1 describes the Mistral2 concepts. The FACTS concepts are described in Section 3.2. Finally, in Section 3.3 a number of interfacing problems are discussed and a few examples are shown.

3.1 Mistral2 Concepts

3.1.1 Theory

The Mistral2 binary data structure contains several hierarchical levels. Each level can contain blocks, that in turn can contain RT's. This is described in Chapter 4. We define a basic block as a set of RT's which do not contain feedback data edges and which occur in the same hierarchical block in the Mistral2 binary data structure. At a more abstract level, we can say that a basic block is a set of RT's which are seen by the control flow as one piece.

Each RTG representation consists of one or more basic blocks. A basic block can be a tiny loop body as well as a representation of a complete processor (without data feedback and hierarchy of course).

3.1.2 Example

Let's start with the following Data Flow Language (DFL) description of a processor:

```plaintext
#define WORD fix<16,15>

func main(in1, in2, in3, in4: WORD) out: WORD =
begin
  tmpl = in1 * in2;
  tmp2 = in3 + in4;
  out = tmpl - tmp2;
end;
```

We will not go into detail about the basics of DFL. We will confine ourselves to saying that this processor has four inputs (in1, in2, in3 and in4), one output (out) and some functionality between the inputs and the output. Furthermore, no loops nor conditional constructs are present in the input
description. In the Mistral2 binary data structure of this example, all RT’s will occur in the same hierarchical block.

Figure 3.1 shows the resulting RTG generated by Mistral2. The four inputs are read from an input buffer. First, inputs in1 and in2 are written in the multiplier registers by respectively RT8 and RT7. Inputs in3 and in4 are written in the alu registers by respectively RT4 and RT3. The Mistral2 multiplier consists of two stages. RT5 writes the result value of the first stage in a pipeline register. RT6 writes the result of the multiply (which is present in the pipeline register) in the first alu register, that is used by RT1. Due to the sequence edge with negative weight from RT6 to RT5, RT6 is executed exactly one cycle after RT5. At the right side of Figure 3.1, RT2 adds the values in the alu registers and writes the result in the second alu register. RT1 subtracts the values in the alu register and writes the result in the output buffer register. Finally, RT0 writes the value of the output buffer register into the output buffer.

![Figure 3.1: A Basic Block RTG](image)

**Resource Conflicts**

Mistral generates a template processor architecture. Which functional resources are actually used by the processor, depends on the algorithmic description. For each type of operation, one resource will be allocated. For example, if the description contains one or more multiplications, one multiplier will be allocated.

In the example described above, two alu operations are present. If no resource pragma is used, Mistral2 generates one alu. This means that RT2 and RT1 can not be scheduled in the same cycle. Otherwise they would have a resource conflict. But this is irrelevant because RT2 and RT1 have a data
dependency between them. So even if $RT2$ and $RT1$ would use different resources (e.g. two different 
alu's), they can not be scheduled in the same cycle.

A real resource conflict exists between the four RT's that read inputs from the input buffer, $RT1$ to $RT4$. Each cycle, only one value can be read from the input buffer. Hence, $RT1$ to $RT4$ will be scheduled in four different cycles.

3.2 FACTS Concepts

3.2.1 Theory

Architecture

In FACTS, two types of resources can be specified, namely functional resources for processing data, and memory resources for storing data. Functional resources are defined by modtypes (module types). Memory resources are defined by memtypes (memory types).

Each module type is characterised by three attributes:

- delay
  The delay defines the number of cycles between the start time of an operation and the time that it is finished.

- data introduction interval
  The dii defines the minimum number of cycles between the start time of an operation and the start time of a next operation on the same functional resource.

- number of instances
  The number of instances defines how many resources of that type are available in the architecture.

It is also possible to add attributes to the definition of a module type. An attribute contains extra information about the module. FACTS first checks whether or not the attribute name is a FACTS attribute name. If not, the attribute has no influence at all. If the attribute name is a FACTS attribute name, FACTS can use the value of the attribute. We do not go into detail about this. An example of a module type definition in the input format is the following:

```
modtype alu {
  delay 2;
  dii 1;
  instances 2;
  attr is_special_type int 1;
}
```

In this example, it takes the alu two cycles to execute an operation. Each cycle, new data may be introduced. The architecture contains two instances of this module type. Finally, the attribute is_special_type of type int is given the value 1.

Each memory type is characterised by two attributes:

- access
  The access method is either random access, fifo access or stack access.
• capacity

The capacity defines the number of values that can be stored simultaneously.

An example of a memory type definition in the input format is the following:

```
memtype register1_alu {
    @access random;
    @capacity 2;
}
```

In this example, the memory type `register1_alu` can be randomly accessed. Furthermore, it can store two values at the same time.

**Data Flow Graph**

A data flow graph (DFG) is a directed graph that models the algorithm to be implemented. Each node represents an operation and each edge represents a value. Considering the DFG as a whole, its latency and its dii can be specified. The latency defines the maximum number of cycles within which all operations of the DFG must be executed. The dii defines the data introduction interval [GM94]. This can be done in the following way:

```
dfg main () () {
    @latency 3;
    @dii 3;
}
```

It is possible specify which input variables the DFG consumes, and which output variables the DFG produces. This can be done in the following way:

```
dfg main (<input_variables>) (<output_variables>) {
    <functionality>
}
```

Within the DFG, operations, resource usage, memory bindings and distances between operations can be specified. An operation consists of a functional resource, zero or more inputs and zero or more outputs. For example:

```
op1: out = alu(in1, in2);
```

Operation `op1` uses the module type `alu`. It consumes the two input values `in1` and `in2`, and produces the output value `out`. Suppose we want to bind the output value `out` to a memory type `reg`. It can be done in this way:

```
op1: out = alu(in1, in2);
@membind(out, reg);
```

Furthermore, a distance between two operations can be specified. Suppose a dummy resource `dummy_resource` is present which produces the input values. Then a distance pragma can be added in the following way:
op0: in1, in2 = dummy_resource();
op1: out = alu(in1, in2);
@membind(out, reg);
@dist(op0, op1) == 1;

Now, op1 is executed exactly 1 cycle after op0.

Furthermore, each DFG contains a source and a sink. The source node is the first operation to be executed, and the sink node is the last operation to be executed. These nodes do not use any resources and have a delay of zero cycles. By specifying a distance between an operation and the source node or the sink node, timing constraints can be expressed. For example:

op1: out = alu(in1, in2);
@dist(source, op1) >= 2;

Now, op1 is executed in cycle 2 or later.

3.2.2 Example

This subsection shows an example of the input format of FACTS.

modtype alu {
    @delay 1;
    @dii 1;
    @instances 1;
}

modtype mux_alu {
    @delay 1;
    @dii 1;
    @instances 1;
}

memtype reg {
    @capacity 1;
    @access random;
}

dfg main (in1, in2) (out) {
    @latency 2;

    RT0: tmp = alu(in1,in2) | mux_alu();
    @membind(tmp, reg);

    RT1: out = alu(tmp);
    @dist(RT0, RT1) >= 1;
}
In this example, a DFG main is present which contains two operations, $RT_0$ and $RT_1$. The architecture contains two module types, $alu$ and $mux_{alu}$, and one memory type, $reg$. The DFG has two input, $in1$ and $in2$, and one output, $out$.

A couple of things attract attention. First of all, $RT_0$ uses two functional resources, separated by a '|' . This means, that the resources $alu$ and $mux_{alu}$ are used by $RT_0$ at the same time. Furthermore, the resource $alu$ consumes two inputs in $RT_0$, although it consumes only one input in $RT_1$. This indicates that the number of inputs (and outputs) a resource consumes (and produces), is not fixed. Finally, the line 
\[
\text{dist}(RT_0, RT_1) \geq 1;
\]
shows that a minimal (or maximal, using $\leq$) distance can also be specified between two RT's, in addition to the fixed distance between RT's shown in Subsection 3.2.1.

3.3 Interfacing

This section explains how the architecture and the RTG information are transformed to the FACTS input format. Those subsections discuss the problems that occur. Finally, some experiments with their results are shown and we evaluate the working of the interface regarding basic blocks.

3.3.1 Architecture

Functional Cores

The operational unit library of Mistral2 contains the following operational units:

- **ALU**
  The arithmetic logical unit is used to perform common arithmetic operations, such as additions and subtractions.

- **ACU**
  The accumulator is used to increment or decrement loop counters and addresses.

- **MULT**
  The multiplier is used to perform multiplications.

- **MUX**
  The multiplexer is used to select inputs for registers of functional resources, or to select inputs directly for functional resources.

- **MAC2**
  The multiply and accumulate unit with two inputs is used to multiply and accumulate three values (e.g. $3 + 4 \times 2$), using only two inputs. This means that the two multiplication values are used before the addition value.

- **MAC3**
  The multiply and accumulate unit with three inputs is used to multiply and accumulate three values using three inputs. In this case, all three values are used at the same time.

- **ASU**
  The application specific unit can be used to define any other operation unit.
For each ALU, ACU and MUX the following FACTS input is generated:

```plaintext
deftype <name_of_unit> {
    @delay 1;
    @dii 1;
    @instances 1;
}
```

The values for `delay` and `dii` are equivalent to the values Mistral2 uses. The Mistral2 values can not be extracted from the binary data structure.

Without specifying any pragma's, Mistral2 will create an architecture containing only one instance of each used module type. If one adds another instance of the same module type, the interface will generate a new module type functionally equivalent to the original module type. The use of these module types has already been decided by Mistral2. Due to the fact that the interface is placed after the 'Scheduling and Register Assignment' step in the Mistral2 design flow, the resource assignment has already been done.

The Mistral2 MULT and MAC3 functional units use two clock cycles to execute an operation, while each clock cycle, a new operation can be started. In order to execute an operation on a MULT or MAC3 functional unit, Mistral2 generates two RT's. However, these RT's use the same functional unit, namely respectively a MULT or a MAC3. This means that FACTS would find a resource constraint between the first RT and the second RT of such an operation. This implies that FACTS can not start a new operation before the previous operation is finished. However, within Mistral2 it is possible to start a new operation on a MULT or a MAC3 each cycle. Therefore, for each MULT and MAC3 the following FACTS input is generated:

```plaintext
deftype <name_of_unit> {
    @delay 1;
    @dii 1;
    @instances 1;
}
```

```plaintext
deftype <name_of_unit>_stage_2 {
    @delay 1;
    @dii 1;
    @instances 1;
}
```

It seems obvious that a more simple solution would be

```plaintext
def_type <name_of_MULT/MAC3> {
    @delay 2;
    @dii 1;
    @instances 1;
}
```

But as explained above, this would imply a resource constraint between the first RT and the second RT of the operation. The changes this implicates in the DFG are explained in Subsection 3.3.2. Using our solution, FACTS thinks the second stage of the operation uses another module type, which is actually not the case.

The functional units MAC2 and ASU are not considered within this project.
Memory Units

The memory unit library contains the following memory units:

- **IPB**
  The input buffer provides inputs for the processor.

- **OPB**
  The output buffer receives outputs produced by the processor.

- **RAM**
  The RAM is used to read data from and to write data in.

- **ROM**
  The ROM is used to read data from.

- **REGFILE**
  The register files (e.g. pipeline registers, normal registers) are used to store data.

- **ROMCTRL**
  The romcontroller provides constant values that can be used as e.g. inputs or as initial values for loop counters.

- **TRIBUFFER**
  The three state buffer is used to pass values (one state) or not (zero state) or is disconnected (high impedance state).

For each register (file), the following FACTS input is generated:

```plaintext
memtype <name_of_unit> { 
  @capacity <capacity>;
  @access random;
}
```

The capacity of a memory unit is present in the Mistral2 binary data structure. Furthermore, the access of memory types is always set to 'random', because that is the only access method that is supported by Mistral2.

The TRIBUFFER is a memory unit that is not considered within this project. This is due to the fact that none of the Mistral2 examples we use and have used, contain a TRIBUFFER.

The IPB and the OPB are special types of memory units. For every operation, FACTS requires the usage of a module type. Reading a value from the IPB, or writing a value to the OPB, does not use a functional core. Furthermore, it takes one cycle to execute such an operation. Taking this into account, the interface generates for each IPB and OPB the following FACTS input:

```plaintext
modtype <name_of_unit> { 
  @delay 1;
  @dii 1;
  @instances 1;
}
```
If an input value from the IPB is used more than once, Mistral2 reads the input again from the IPB. Due to this, using an input value more than once takes more than one clock cycle to read it from the IPB.

The same train of thought holds for the memory types RAM and ROM. Therefore, the interface generates for each RAM and ROM the same output as for an IPB or OPB. The description of a memory type being used as a module type implies changes in the DFG. These are described later on in this section.

The last memory unit, the ROMCTRL, provides constant values. Just like the IPB, OPB, RAM and ROM, providing a constant value does not use a functional core and takes one cycle to execute. Therefore, the interface generates for each ROMCTRL the same as for an IPB, OPB, RAM or ROM. A special case occurs when the ROMCTRL generates one value that is consumed by several RT’s. More details about this can be found in Subsection 3.3.2.

3.3.2 RTG Information

General RT interfacing

From each RT in a basic block RTG, FACTS needs the following information:

- Name of the RT
- Operand variables
- Result variables
- Execution unit that executes the RT

If the RT is present in a hierarchical state or a loop state, FACTS needs additional information. This will be explained later on.

If the RT does not use a MULT or a MAC3, the interface generates the following FACTS input:

<RT_name>: <result_variables> = <execution_unit>({operand_variables});

Note that the ‘=’ is only generated if one or more result variables are present. The order of the operand variables is reversed compared to their order in the input description. For example, if the input description contains the line

\[ x = a - b; \]

the interface will generate (among others) the following:

<RT_name>: \( x = \text{alu}(b, a); \)

This is due to the order of the variables in the Mistral2 binary data structure.

Furthermore, for every RT a node attribute is added containing information about the mode the execution unit is in. For example, this node attribute can tell us whether an alu performs an addition or a subtraction. The interface adds such an attribute in the following way:

@node_attr <RT_name> op_type str "<mode>";

Additionally, the interface inquires the data structure for possible registers in which the result variables are stored and for possible sequence edges. For each result variable that is written to a register file, the interface adds the following line:
@mernbind(<result_variable>, <register>);

For each sequence edge from $RT_x$ to $RT_y$ with a weight $weight$, the interface adds the following line:

@dist($RT_x$, $RT_y$) >= <weight>;

This means, that $RT_y$ starts at least in the same cycle as $RT_x$. For each couple of RT's that write and read a pipeline register, the interface adds the following line:

@dist($RT_x$, $RT_y$) == 1;

This is done, because a Mistral2 pipeline register can store a value for only one clock cycle. Pipeline registers are used in multiplications for example.

If a Mistral2 RTG contains the same result variable in more than one RT, FACTS has to rename that variable because it does not allow a variable to be produced more than once. This is done using the following line:

@edge_rename <result_variable> "";

From this point on, this result variable may be reused as a data dependency or value identifier without causing any conflicts at FACTS side.

### The MULT and the MAC3

As stated before, the MULT and MAC3 require extra attention. Due to the fact that the interface generates an extra module type (see Subsection 3.3.1), the resource information of the second step of the MULT/MAC3 must be modified. In this case, the interface generates the following FACTS input:

<RT_name_1>: <tmp_result> = <name_of_mult/mac>({operand_variables});
@node_attr <RT_name_1> str "<mode>";
@mernbind(<tmp_result1>, <pipelineresister_of_mult/mac>);
@dist(<RT_name_1>, <RT_name_2>) == 1;

<RT_name_2>: <result> = <name_of_mult/mac>_stage_2(<operand_variables>);

Note that the input values for $RT_{name.2}$ are not the same as the input values for $RT_{name.1}$.

### The ROMCTRL

The ROMCTRL can generate one value that is consumed by different RT's. The next FACTS input example illustrates this:

$RT_1$: tmp1 = romctrl(int_5);
$RT_2$: tmp2 = romctrl(int_5);

In this example, the ROMCTRL $romctrl$ produces the variable $int_5$ (which, surprisingly, represents an integer of value 5). This variable is consumed by both $RT_1$ and $RT_2$. In Mistral2, $RT_1$ and $RT_2$ can be scheduled in one cycle (provided that no other constraints are present). Because these RT's all use the module type $romctrl$, FACTS notices a resource conflict between $RT_1$ and $RT_2$ and thus can not schedule them in one cycle. Mistral2 does not consider this kind of resource usage as a resource conflict. This problem does not occur when the ROMCTRL generates different values for different RT's.

The solution to this problem could be to add a certain attribute to a node that uses a ROMCTRL. This attribute should inform FACTS to take the resource for granted and thus not to notice a resource conflict.
Reassignment of Variables (C)

When specifying an input description in C, it is possible to reassign values to a variable. The next Mistral2 input example illustrates this:

```c
int main() {
    int x = 0;
    int y1 = x + 1;
    x = 1;
    int y2 = x + y1;
    return y2;
}
```

In this example, two values are assigned to the variable `x`, namely 0 and 1. For the second assignment, the interface modifies the name of variable `x` to `x_` in order to create a unique name for each variable. Mistral2 schedules the RT's in such an order, that the order of the functions in the input description is maintained in the RT schedule.

3.3.3 Examples

Table 3.1 shows the input descriptions of the three basic block examples 'Function Tree', 'Reassignment' and 'IPB Use' for Mistral2.

Subsection B.1.1 at page 53 also shows these input descriptions. The resulting outputs of the interface are shown respectively in Subsection B.1.2 at page 54, in Subsection B.1.4 at page 58 and in Subsection B.1.3 at page 56.

'Function Tree'

The resulting output is accepted by FACTS. Furthermore, the schedule generated by FACTS is the same schedule as generated by Mistral2.

'Reassignment'

The resulting output is accepted by FACTS. Furthermore, Mistral2 and FACTS both generate the same schedule.

'IPB Use'

The resulting output is accepted by FACTS. Mistral2 and FACTS both generate a schedule of 7 cycles, but the order of the RT's in the FACTS schedule is slightly different from the Mistral2 schedule.
Table 3.1: Basic Block Examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Description</th>
</tr>
</thead>
</table>
| Function Tree| ```
#define WORD fix<16,15>

func main(in1, in2, in3, in4: WORD) out: WORD =
begin
  tmp1 = in2 * in1;
  tmp2 = in4 + in3;
  out = tmp2 - tmp1;
end;
``` |
| Reassignment | ```
int main() {
  int x = 0;
  int y1 = x + 1;
  x = 1;
  int y2 = x + y1;
  return y2;
}
``` |
| IPB Use      | ```
#define WORD fix<16,15>

func main(in1, in2, in3: WORD) out: WORD[2] =
begin
  tmp1 = in1 + in2;
  out[0] = tmp1 * in3;
  out[1] = tmp1 + in3;
end;
``` |
Chapter 4

Hierarchy

This chapter discusses the interfacing problems that are related to the use of hierarchy. Section 4.1 describes the Mistral2 concepts. The FACTS concepts are described in Section 4.2. Finally, Section 4.3 discusses a number of interfacing problems and shows a few examples.

4.1 Mistral2 Concepts

4.1.1 Theory

Within the data structure of a Mistral2 processor, RT's are present in a hierarchical structure. At the highest hierarchical level, Mistral2 provides a timeloop state (next to a possible initialisation state, which is not relevant from our point of view). A timeloop state can contain block states, loop states and normal states. Block states and loop states can contain other block states and loop states, and normal states. A normal state contains primitive states. Every RT is present in a primitive state. From these primitive states, the interface extracts the RT information it needs. An overview of the hierarchical classification of these types of states is shown in Figure 4.1. For the sake of convenience, normal states and primitive states are left out in the RTG figures.

4.1.2 Example

We start with the following DFL description of a processor:

```plaintext
#define WORD fix<16,15>

func main() out: WORD[3] =
begin
    x = 0;
y = 1;
begin
    out[0] = x;
end;
begin
    out[1] = y;
end;
```

24
In order to show some hierarchy in the Mistral2 state structure, we put \texttt{out[0]} and \texttt{out[1]} in extra blocks using the \texttt{begin} and \texttt{end} statements. Figure 4.2 shows the resulting Mistral2 RTG.

The RT's \texttt{RT1}, \texttt{RT3} and \texttt{RT5} read values and write them into the output buffer register. Interesting to see here is that the value for \texttt{out[2]}, which is the addition of the values of variables \texttt{x} and \texttt{y}, has already been computed by the compiler. In other words, no alu is present to execute the addition of \texttt{x} and \texttt{y}. \texttt{RT0}, \texttt{RT2} and \texttt{RT4} write the values of the output buffer register to the output buffer. Furthermore, due to the \texttt{begin} and \texttt{end} statements around \texttt{out[0]} and \texttt{out[1]}, we see that \texttt{RT2} and \texttt{RT4} are placed at a different hierarchical level.

\section{FACTS Concepts}

\subsection{Theory}

In the \textsc{FACTS} input description, hierarchy can be expressed by specifying sub-DFG's. These sub-DFG's add structure to the input description. As stated in Subsection 3.2.1, each DFG has a source and a sink node. In the input description, sub-DFG's occur in their hierarchical higher level DFG('s) as one node. For scheduling, the entire DFG structure is flattened. The nodes referring to sub-DFG's are expanded in their hierarchical higher level DFG('s). These expanded nodes contain a source and a sink node themselves.

A node name can occur in different DFG's. For \textsc{FACTS}, they are different nodes, without any name conflicts. The same holds for variable names. However, if a variable name represents one and
the same variable in different DFG's, it can be transported between those DFG's in the same way as shown in Subsection 3.2.1:

```
dfg sub_dfg (<input_variables>) (<output_variables>) {
   <functionality>
}
```

In this way, `input_variables` can be transported from the hierarchical higher level DFG into `sub_dfg`, and `output_variables` can be transported from `sub_dfg` into the hierarchical higher level DFG.

Sub-DFG's must be described earlier in the input description than their hierarchical higher level DFG's.

### 4.2.2 Example

This subsection shows an example of the use of hierarchy in the input format of FACTS. For the sake of convenience, the architecture description and the non-interesting parts of the DFG descriptions are skipped.

```
dfg func1 (f_in1, f_in2) (f_out) {
   n1: tmp1 = alu(f_in1, 3);
   n2: tmp2 = alu(f_in2, 2);
   n3: f_out = alu(tmp1, tmp2);
}

dfg main () () {
   n1: in1 = ipb();
   n2: in2 = ipb();
   n3: out = func1(in1, in2);
}
```
This example shows two DFG's. The sub-DFG func1 has two inputs, \texttt{f.in1} and \texttt{f.in2}, and one output, \texttt{f.out}. In the main DFG \texttt{main}, two input values are read from the input buffer in nodes \texttt{n1} and \texttt{n2}. Node \texttt{n4} writes the value of the variable \texttt{out} to the output buffer. Node \texttt{n3} shows the use of hierarchy. In this node, the DFG \texttt{func1} is called using the input variables \texttt{in1} and \texttt{in2}. The result variable \texttt{out} contains the result value obtained from \texttt{func1}.

### 4.3 Interfacing

#### 4.3.1 RTG Information

Subsection 4.1.1 has explained the hierarchy in the Mistral2 binary data structure. This section explains how this structure is transformed to the DFG structure in the FACTS input format.

**Access of Mistral2 Hierarchy**

As stated in Subsection 2.3.1, to read the Mistral2 binary data structure we use a tool that reads information after the 'Schedule & Register Assignment'. This means, the tool reads the RT's in scheduled order, regardless of the hierarchical Mistral2 state they are in. As opposed to that, in FACTS the RT's must be placed in their hierarchical states, regardless of the schedule order.

The interface therefore cannot directly write RT information to the FACTS input description. First, the information must be written to the DFG it belongs to. When all RT's have been visited and the right information has been written to the right DFG's, the DFG's are written to the FACTS input description. This is done in order of ascending hierarchical level. That means, the top level DFG('s) will be the last DFG('s) in the FACTS input description.

A node that represents a sub-DFG is initially not present in the FACTS input description. The reason for this is, that in the Mistral2 binary data structure this RT is not present either. Therefore, for every sub-DFG a node is added to its hierarchical higher level DFG. This node occurs after the RT that is scheduled just before the first RT of the hierarchical lower level state.

**Same Variables in Different DFG's**

In the Mistral2 binary data structure, every variable name is unique. That means, if a variable is present in different hierarchical states, it will have one and the same name in every state. Hence, in FACTS that variable name must represent the same variable in every DFG. Therefore, the interface checks which variable names are produced in a certain DFG and consumed in a lower level DFG. These variables are specified as input variables for the lower level DFG. Next to that, the interface checks which variable names are consumed in a certain DFG and produced in a lower level DFG. These variables are specified as output variables for the lower level DFG. Finally, an extra node is added to the higher level DFG, representing the lower level DFG.

#### 4.3.2 Examples

Table 4.1 shows the input descriptions of the two hierarchy examples 'Blocks1' and 'Blocks2' for Mistral2.
Table 4.1: Hierarchy Examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|       | #define WORD fix<16,15>
|       | func main() out: WORD[2] =
|       | begin
|       | out[0] = 2;
|       | begin
|       | out[1] = out[0] - 1;
|       | end;
|       | end; |
| Blocks2 | 
|       | 
|       | #define WORD fix<16,14>
|       | func main() out: WORD[3] =
|       | begin
|       | out[0] = 2;
|       | begin
|       | out[1] = out[0] - 1;
|       | out[2] = out[0] + out[1];
|       | end;
|       | end; |

Subsection B.2.1 at page 60 also shows these input descriptions. The resulting outputs of the interface are shown respectively in Subsection B.2.2 at page 61 and in Subsection B.2.3 at page 63.

'Blocks1'
The resulting output is accepted by FACTS. Mistral2 and FACTS both find a schedule with a length of six cycles. However, the schedules are not equivalent. First of all, Mistral2 schedules the use of a ROMCTRL that produces the same value twice, in one cycle. FACTS schedules it in two cycles.

On the other side, Mistral2 puts all RT's of the sub-DFG within 'Blocks1' later in the schedule than all other RT's from the main DFG. FACTS however does not have that constraint and thus is able (in this case) to schedule the first RT of the sub-DFG, RT8, in the same cycle as one of the last RT's in the main DFG, RT0, simply because there is no data dependency, sequence dependency or resource conflict between those two RT's.

'Blocks2'
The resulting output is accepted by FACTS. The schedule Mistral finds, is one cycle longer than the schedule FACTS finds. The reason for this is the same reason that holds for 'Blocks1': FACTS schedules RT0 and RT13 in one clock cycle, although Mistral2 does not do that.
Chapter 5

Loops

This chapter discusses loops. Section 5.1 describes the Mistral2 concepts. The FACTS concepts are described in Section 5.2. This section also explains some details about software pipelining (also called loop folding). Finally, in Section 5.3 a number of interfacing problems are discussed and a few examples are shown.

5.1 Mistral2 Concepts

5.1.1 Theory

Mistral2 accepts both the DFL language and the C language as input languages. Many variations exist in describing a loop in these languages. For example, in C one can write a "for" loop or a "while" loop in many different forms. However, the loop model in the Mistral2 RTG is basically the same for all kind of loops. This model contains the following:

- loop body functionality
- loop control
  - initialise loop counter
  - initialise exit flag
  - update loop counter
  - update exit flag

The idea of a loop is to execute the loop body functionality as long as a certain condition holds. This condition is expressed in the exit flag. The Mistral2 RTG only contains an RT that increases or decreases the loop counter, and an RT that updates the exit flag. The one or more RT’s that execute the loop body functionality are not connected in any way to those loop control RT’s. In other words, in the RTG no relation exists between the exit condition of the loop and the loop body functionality itself. Whether or not the loop body functionality is executed, is controlled by the micro code of the processor. This micro code actually uses the status of the exit flag in relation to the execution of the loop body functionality.

For the rest of this subsection, the theory is explained as it occurs in loop RTG’s that are generated using the following kind of loop description in the Mistral2 DFL input:
This means that the value for the variable `counter_name` is initially set to `start_value`. As long as `counter_name` has not reached the value `end_value`, the `loop_functionality` is executed and the value of `counter_name` is increased or decreased, depending on which of the two values `start_value` and `end_value` is greater.

The results for "for" loops will basically be the same. When using particular kinds of "while" constructions, an extra conditional construction is present that determines whether or not the loop body functionality must be executed. We will not go deeper into the results of these and other kinds of loop constructions in the Mistral2 DFL or C input.

### Computing the Exit Condition

The loop counter is initially set to `end_value`. In the first cycle of each loop iteration, the counter is decremented. Furthermore, the status of the exit flag is computed using that decremented counter value. In other words, after the first cycle of the loop body it is clear whether or not the next iteration must be started.

A new iteration of a loop can possibly start before the previous iteration ends. This is called loop folding or software pipelining [GG89]. In Mistral2, one can choose whether to use 'Unconstrained Folding' on loops or not. Due to the fact that we want FACTS to (possibly) fold loops, we do not use folding while generating the Mistral2 loop RTG.

### Input Ports and Output Ports

Let's start with the following DFL example:

```c
#define WORD fix<16,15>

func main() out: WORD[10] =
begin
    out[0] = 0;
    (i: 1..9)::
        out[i] = out[i - 1];
end;
```

The goal of showing this example is to show what the structure of a Mistral2 loop looks like, especially regarding the input ports and output ports of its loop state and loop body. We will not explain the functionality of the RT's.

Figure 5.1 graphically shows the structure of this Mistral2 loop example. The loopstate in this example contains six loop state input ports (`sil to si6`), six loop body input ports (`bil to bi6`), six loop body outputports (`bol to bo6`) and five loop state output ports (`sol to so6`).

A difference exists between the loop state and the loop body. All input edges of the loop body come from the input ports of the loop state. Five out of six output edges of the loop body are connected to the five output ports of the loop state. The edge from `bol` represents the transfer of the exit flag status to a control register. This transfer does not use the datapath of the processor, but it uses a status bus. Therefore, this edge is not connected to an output port of the loop state.

The loop state has four feedback edges connected to its input ports and output ports. These ports are respectively `si1`, `si2`, `si3` and `si5`, and `so1`, `so3`, `so4` and `so5`. Feedback edges carry output.
Figure 5.1: Input Ports and Output Ports within a Loop RTG
information from the output port(s) of the loop back to its input port(s). Values of variables that are produced or changed during an iteration, and that are used in a next iteration, are called loop carried dependencies. These dependencies are caused by using certain structures in the input description, such as reassignment of variables within a loop body.

The input edges connected to ports \(si4\) and \(si6\), and the output edge connected to port \(so2\) have no feedback.

### 5.1.2 Example

Let's start with the following DFL description of a processor:

```plaintext
#define WORD fix<16,15>

func main() out: WORD[10] =
begin
  x = 1;
  (i:0..9):
    out[i] = x;
end;
```

This example describes a loop. The processor has no inputs and one array output which has a size of ten. The loop counter \(i\) runs from 0 up to 9. Every iteration, the value of the variable \(x\), which is set to 1, is written to a certain field of the output.

![Figure 5.2: Example of a loop](image)

Figure 5.2: Example of a loop
A graphical representation of the resulting RTG and its hierarchical structure is given in Figure 5.2. This figure shows in fact three separated RTG's. The left RTG, containing \( RT2, RT1 \) and \( LB1 \), takes care of the loop control, together with the isolated RTG, containing only \( LB0 \). The right RTG, containing \( RT4 \) and \( RT3 \), handles the writing of the outputs.

The loop counter \( i \) is initially set to 10 by \( RT2 \) and is put into the acu. Each iteration, this counter is decreased by \( RT1 \), opposite to the increase in the DFL description. Furthermore, the \( creg \) flag is initially set to \( false \) by \( LB0 \). The datapath of a Mistral2 processor does not provide registers to keep flags. Therefore, this control register is necessary.

Each iteration, an acu flag is set to either true or false, depending on whether the result of the decrement of \( i \) equals 0 or not. \( RT4 \) gives variable \( x \) the value 1. This value is only generated once by the ROMCTRL. After that, variable \( x \) is kept in the output buffer register. Each iteration, \( RT3 \) writes the value of \( x \) to the output buffer. Another possibility is that the ROMCTRL would generate a constant value each iteration, but this is not the case.

The RTG does not show any relation between \( RT3 \), which contains the loop body functionality and \( RT1 \) and \( LB1 \), which respectively update the loop counter and pass the exit flag. The only relation between the loop body functionality and the loop control, is the schedule that Mistral2 has generated. Updating the loop counter, passing the flag and writing the value of \( x \) to the output buffer all happens in the body of the loop. In this specific case, one iteration takes one clock cycle.

### 5.2 FACTS Concepts

#### 5.2.1 Theory

**Feedback Data Edges**

For FACTS, the problem with loops concentrates on feedback data edges. These edges result in a cyclic DFG, which cause problems for the scheduler. The example in Figure 5.3 illustrates this.

![Figure 5.3: Which RT to schedule first?](image)

In order to be able to deal with cyclic graphs, FACTS uses the following model. For each feedback data edge, a new node is introduced which is in fact a copy of the node that consumed the feedback data. This is illustrated in Figure 5.4.

The backgrounds of this model are the following. Let's start from a situation as showed in Figure 5.3, in which a data edge exists from \( A \) to \( B \) and a feedback data edge exists from \( B \) to \( A \). Suppose \( A \) is in fact the first operation to be executed. This means that an iteration distance, \( id \), of one exists.
from the first execution of $B$, $B_0$ to the second execution of $A$, $A_1$:

$$B_0 \xrightarrow{\text{id}} A_1 \quad (5.1)$$

And generally, an iteration distance of one exists from $B$ to $A$:

$$B \xrightarrow{\text{id}} A. \quad (5.2)$$

Furthermore, an initiation interval, $II$, exists from $B_0$ to $B_1$, and a negative initiation interval exists from $B_1$ to $B_0$:

$$B_0 \xrightarrow{II} B_1 \quad (5.3)$$

and

$$B_1 \xrightarrow{-II} B_0. \quad (5.4)$$

From this, the following expressions can be deduced:

$$B_1 \xrightarrow{-II} B_0 \xrightarrow{1} A_1 \quad (5.5)$$

hence

$$B_1 \xrightarrow{-II + 1} A_1 \quad (5.6)$$

or generally

$$B \xrightarrow{-II + 1} A. \quad (5.7)$$

This last expression can also written using starttimes $s(X)$:

$$s(A) \geq s(B) - II + 1 \quad (5.8)$$

or

$$s(B) \leq s(A) + II - 1 \quad (5.9)$$

which can also be written as

$$s(B) < s(A) + II \quad (5.10)$$

34
This means that operation \( B \) is not allowed to start later than one initiation interval after operation \( A \).

Now it is possible to update Figure 5.4 to Figure 5.5. The graphs in this figure are equivalent. One can specify the iteration distances (as shown in Figure 5.5) in the FACTS input format using the following pragma:

\[ \text{@dist}(A', A) == (1,0); \]

This line means: \( A' \) must be scheduled in the same clock cycle as \( A \) and one iteration later than \( A \).

This line is equivalent to

\[ \text{@dist}(A', A) == (0, -I); \]

The only problem here is that the value for \( I \) might not be known. Due to the functionality in the current version of FACTS, feedback edges are detected and no distance pragmas regarding iteration distances and initiation intervals have to be provided in the input format. This functionality is explained in the following part, which describes "ENTRY" and "EXIT" nodes.

**ENTRY and EXIT Nodes**

In addition to the scheduling problem, feedback edges also cause other problems. If the feedback edge is carried back to an input port that is also connected to a non-feedback data edge, it is not obvious anymore which value is present at the input port. Furthermore, at the output port it is not obvious whether or not the produced value must be used as a feedback value and as a non-feedback value at the same time.

Figure 5.6 illustrates this.

To cope with this problem, FACTS uses ENTRY and EXIT nodes. An ENTRY node is used to model an RT that consumes feedback data. It has two inputs and one output. Therefore, it can separate the non-feedback value and the feedback value at the input port of the RT. An EXIT node is used to model an RT that produces feedback data. It has one input and two outputs. Therefore, it can separate the non-feedback value and the feedback value at the output port of the RT. The ENTRY and EXIT nodes use a special module type. These modules are modelled in the FACTS input format as follows:

```plaintext
modtype ENTRY_mod {
  @delay 0;
}
```
Figure 5.6: Problems at the Input Port and at the Output Port

@di 0;
@instances 1;
@attr is_entry_type int 1;
}

mod type EXIT_mod {
  @delay 0;
  @di 0;
  @instances 1;
  @attr is_exit_type int 1;
}

The delay and data introduction interval of both module types are zero. This implies that only one
instance of the modules is necessary and the schedule of a loop rtg is not affected in any way. The
attributes is_entry_type and is_exit_type are used by FACTS to detect that these modules are really an
ENTRY module and an EXIT module. Therefore, the names ENTRY_mod and EXIT_mod can be
chosen arbitrarily.

Figure 5.7 shows a graphical representation of an operation, using the ENTRY and EXIT nodes.
The functionality of rtl itself does not matter in this example. In addition to the use of the ENTRY and
EXIT nodes, an attribute is added to let FACTS know that the DFG is a loop, that has been generated
by Mistral. In the FACTS input format, the use of the ENTRY and EXIT nodes and modules looks as
follows:

dfg main (in) (out) {
  @attr loop_type id mistral2;
  rtl_ENTRY: entry_out = ENTRY_mod(in, in_iteration);
  rtl: exit_in = alu(entry_out);
  rtl_EXIT: out, in_iteration = EXIT_mod(exit_in);
}

The attribute that is mentioned above, has recently been added. Therefore, it does not show up in most
of the output files shown in Appendix B.

36
Loop Folding

FACTS is able to minimise the latency and the data introduction interval of a DFG. It is also possible to specify it in the input format, as stated in Subsection 3.2.1. If the latency is greater than the data introduction interval, the DFG can be folded.

Figure 5.8: Loop Folding

Figure 5.8 shows an example of loop folding. The DFG in this figure has a latency of three and a data introduction interval of one. This means that each iteration takes 3 cycles to be finished. Furthermore, each cycle a new iteration can be started.
5.2.2 Example

This subsection shows an example of a loop in the input format of FACTS. For the sake of convenience, the architecture description is skipped.

```dga
dfg loop_i (x, i) () {
  RT3: out = opb_l(x);
  RT1_ENTRY_0: ENTRY_0_out = ENTRY_mod(i, i_iteration);
  RT1: EXIT_0_in, EXIT_1_in = acu_l(ENTRY_0_out);
  RT1_EXIT_0: EXIT_0_out, TEMP = EXIT_mod(EXIT_0_in);
  RT1_EXIT_1: EXIT_1_out, i_iteration = EXIT_mod(EXIT_1_in);
}
```

```dga
dfg main () () {
  RT2: i = romctrl_l();
  @membind(i, reg_l_acu_l);
  RT4: x = romctrl_l();
  @membind(x, reg_l_opb_l);
  RT_loop_i: loop_i(x, i);
}
```

The loop body `loop_i` is placed in a sub-DFG. The DFG `main` calls this sub DFG in operation `RT_loop_i`. `RT1` in the sub-DFG contains one ENTRY node and two EXIT nodes. Note that out of the four variables produced by the EXIT nodes, only one variable (`i_iteration`) is consumed again. Due to the definitions of ENTRY nodes and EXIT nodes, the number of inputs for an ENTRY node and the number of outputs for an EXIT node are fixed. This can imply that certain input variables or output variables are not used. For example, if an RT only produces a feedback value, the RT can still be modelled using an EXIT node. This implies that the non-feedback value of the EXIT node is not used. However, FACTS does not complain about that.

5.3 Interfacing

This section explains how the architecture (5.3.1) and the RTG information (5.3.2) are modified before they are put in the FACTS input format. Finally, some experiments with their results are shown and we evaluate the working of the interface regarding loops (Subsection 5.3.3).

5.3.1 Architecture

After creating the binary data structure of the processor in the memory, the interface extracts the architectural information from the processor. At this point, it is not known whether or not the RTG contains a loop. Therefore, the interface always adds

```dga
modtype ENTRY_mod {
  @delay 0;
  @dii 0;
}
```
5.3.2 RTG Information

For each RT, the interface tries to find out if the RT consumes feedback data and if the RT produces feedback data. For each feedback input variable of an RT, the following RT is added:

\[
\text{<name_of_RT>_ENTRY_<#var>}: \text{<var_name>_ENTRY_<#var>_out} = \text{ENTRY_mod(<var_name>, <var_name>_iteration)};
\]

For each feedback output variable of an RT, the following RT is added:

\[
\text{<name_of_RT>_EXIT_<#var>}: \text{<var_name>_EXIT_<#var>_out, <var_name>_iteration} = \text{EXIT_mod(<var_name>_EXIT_<#var>_in)};
\]

Furthermore, the names of the variables that are consumed and produced by the original RT are modified. This must be done, because the original RT now consumes data that is produced by the added ENTRY module and/or produces data that is consumed by the added EXIT module.

5.3.3 Examples

Table 5.1 shows the input descriptions of the three loop examples 'Simple Loop', 'Loop' and 'Loop in Loop' for Mistral2.

Subsection B.3.1 at page 65 also shows these input descriptions. The resulting outputs of the interface are shown respectively in Subsection B.3.2 at page 66, in Subsection B.3.3 at page 68 and in Subsection B.3.4 at page 70.

'Simple Loop'

The resulting output is accepted by FACTS. We compare the schedules of the loop body. Mistral2 and FACTS both generate a loop body schedule that contains one cycle.

'Loop'

The resulting output is accepted by FACTS. We compare the schedules of the loop body. Mistral2 and FACTS both generate a loop body schedule that contains two cycles.
Table 5.1: Loop Examples

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Loop</td>
<td>#define WORD fix&lt;16,15&gt;</td>
</tr>
<tr>
<td></td>
<td>func main() out: WORD[10] =</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>x = 1;</td>
</tr>
<tr>
<td></td>
<td>(i:0..9):</td>
</tr>
<tr>
<td></td>
<td>out[i] = x;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td>Loop</td>
<td>#define WORD fix&lt;16,15&gt;</td>
</tr>
<tr>
<td></td>
<td>func main() out: WORD[10] =</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>out[0] = 0;</td>
</tr>
<tr>
<td></td>
<td>(i:1..9):</td>
</tr>
<tr>
<td></td>
<td>out[i] = out[i - 1];</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td>Loop in Loop</td>
<td>#define WORD fix&lt;16,15&gt;</td>
</tr>
<tr>
<td></td>
<td>func main() out: WORD[10][10] =</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>(i:0..9):</td>
</tr>
<tr>
<td></td>
<td>(j:0..9):</td>
</tr>
<tr>
<td></td>
<td>out[i][j] = 0;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
</tbody>
</table>

'Loop in Loop'

The resulting output is accepted by FACTS. We compare the schedules of the lowest hierarchical block, which contains the most inner loop body. Mistral2 and FACTS both generate a schedule that contains two cycles.
Chapter 6

Arrays

The previous chapters have discussed specific interfacing problems. This chapter discusses arrays on the basis of examples. They cause no real interfacing problems, but they might introduce unnecessary constraints at FACTS side. Section 6.1 shows the Mistral2 concepts. In Section 6.2, its consequences for FACTS are summarised.

6.1 Mistral2 Concepts

6.1.1 Example 1

Let's take a look at the following example:

```plaintext
#define WORD fix<16,15>

func main(in: WORD[3]) out: WORD[3] =
begin
  out[0] = in[2];
  out[1] = in[1];
  out[2] = in[0];
end;
```

The schedule order of the RT's created by Mistral2 is reversed, compared to the order of the functionality in this example. In other words, in the first cycle `in[0]` is read. After that, in each cycle the input read in the previous cycle is written to the output and the next input is read. Mistral2 does not introduce any sequence edges or other constraints.

6.1.2 Example 2

Let's take a look at the following example:

```plaintext
#define WORD fix<16,15>

func main(in: WORD[3]) out: WORD[3] =
begin
  (i:0..2)::
    out[i] = in[2-i];
end;
```
The loop body of this example contains, among others, the reading of an input and the writing of that same input to an output. Both actions take place within the loop body. Due to the fact that reading an input and writing it to an output cannot be executed in one cycle, Mistral2 needs two cycles for the loop body. A better solution for Mistral2 would be to read the first input before the loop starts, and to write the last output after the loop ends. In that case, the loop body would need only one cycle because the reading of a new input and the writing of a previous input can be executed in one cycle.

Due to the fact that each iteration of the loop uses two cycles, the number of cycles used by the loop body is

$$2 \times \#\text{loops} \quad (6.1)$$

If the solution that is described above is used, two extra cycles are introduced by reading the first input before the loop body starts and writing the last output after the loop body ends. However, each iteration only uses one cycle. Therefore, the number of cycles used by the loop body is

$$2 + \#\text{loops} \quad (6.2)$$

Hence, when the number of iterations increases, the difference between the number of cycles used by the Mistral2 solution and the other solution becomes greater in favour of the latter.

The consequence for FACTS is, that the data dependency, from the first cycle to the second cycle of the loop body in the Mistral2 schedule, forces a latency of two. In the latter case, the latency is only one.

### 6.1.3 Example 3

Let's take a look at the following example:

```c
#define WORD fix<16,15>

func main(in: WORD[3]) out: WORD[3] =
begin
  a = in[2];
  b = in[1];
  c = in[0];
  out[0] = a;
  out[1] = b;
  out[2] = c;
end;
```

The Mistral2 schedule of this example is functionally equivalent to the example described in Subsection 6.1.1. That means that in the first cycle, `in[0]` is read and is assigned to variable `c`. In the next cycles, the next input is read and is assigned to a variable, and the value assigned to the variable in the previous cycle is written to an output.

### 6.1.4 Example 4

Let's take a look at the following example:

```c
#define WORD fix<16,15>
```
func main(out: WORD[3] =
begin
    c[0] = 0;
    c[1] = 1;
    out[0] = c[1];
    c[2] = 2;
    out[1] = c[2];
    out[2] = c[0];
end;

In this example, several values must be written to and read from the ram. First, all values are written to the ram. In the Mistral2 schedule, this is done in the order c[1], c[2], c[0]. Hence, this order is equal to the order of the writing of the outputs in the input description. However, in the Mistral2 schedule the outputs are written according to the order of the assignment of the values to the variables in the input description. In other words, the outputs are written in the order out[2], out[0], out[1], which are related to c[0], c[1], c[2].

6.1.5 Example 5

Let’s take a look at the following example:

```c
#define WORD fix<16,15>

begin
    (i:0..2)::begin
        out[2*i] = in[0];
        out[2*i + 1] = in[1];
    end;
end;
```

The Mistral2 schedule contains a loop body of three cycles. In the first cycle, in[1] is read. In the second cycle, in[0] is read and in[1] is written to an output. Finally, in the third cycle in[0] is written to an output. Mistral2 does not generate any sequence edges between the writing of both outputs in each iteration, and thus does not force FACTS to schedule the writing of one output before the other.

Like the example described in Subsection 6.1.2, it is also possible to obtain a solution that uses less than three cycles in its loop body. The first input can be read before the loop body starts. In the first cycle of the loop body, the first input can be written and the second input can read. In the second cycle of the loop body, the first input can be read for the next iteration and the second input can be written.

If this solution is used, one extra cycle is introduced by reading the first input. However, each iteration only uses two cycles. Therefore, the number of cycles used by the loop body equals

\[ 3 \times \#loops \] (6.3)

for the Mistral2 schedule and

\[ 1 + 2 \times \#loops \] (6.4)

for the schedule of the latter solution.

43
The consequence for FACTS is, that the data dependencies, from the first cycle to the second cycle and from the second cycle to the third cycle of the loop body in the Mistral2 schedule, forces a latency of three. In the latter case, the latency is only two.

6.1.6 Example 6

Let's take a look at the following example:

```c
#define WORD fix<16,15>

func main() out: WORD[4] =
begin
c[0] = 1;
(i:0..2): begin
    c[i + 1] = 1;
    out[i] = c[i];
end;
out[3] = c[3];
end;
```

This example has a loop body of two cycles in the Mistral2 schedule. In the first cycle, the value \( c[i] \) is read from the ram and the value 1 is assigned to \( c[i + 1] \). In the second cycle, the value of \( c[i + 1] \) is written to the ram and the value of \( c[i] \) is written to the output. The only sequence that Mistral2 generates, exists between two RT's in one cycle. The first RT of these two RT's reads the value \( c[i] \) from a certain address in the ram. The second RT of these two RT's updates that address. This update is not allowed to happen before the address is used to read from, and therefore the sequence edge is added.

6.2 Consequences for FACTS

The Mistral2 scheduler does not introduce sequence edges for other purposes than evading the update of addresses that still must be used. However, the schedules that Mistral2 generates for the examples described in Subsections 6.1.2 and 6.1.5 can be slightly changed in order to decrease the number of cycles that is necessary to execute the function.
Chapter 7

Conditional Constructs

This chapter discusses conditional constructs. Section 7.1 describes the Mistral2 concepts. In Section 7.2 a number of interfacing problems are discussed and one example is shown.

7.1 Mistral2 Concepts

7.1.1 Theory

In many processor descriptions, conditional constructs occur. In other words, some functionality of the processor is executed only if a certain condition holds. Whether or not some functionality is executed, is controlled by the micro code of the processor. Within the Mistral2 RTG, all RT's that can possibly be executed are present.

7.1.2 Example

Let's start with the following DFL description of a processor:

```c
#define WORD fix<16,15>

func main(in1, in2: WORD) out: WORD =
begin
  out = if in1 > in2 -> in1
       || in2
     fi;
end;
```

This processor has two inputs, \( in1 \) and \( in2 \), and one output, \( out \). If \( in1 \) is greater than \( in2 \), \( in1 \) is the output; otherwise \( in2 \) is the output. Figure 7.1 shows the resulting RTG generated by Mistral2. The two inputs are read from an input buffer. First, \( RT3 \) and \( RT2 \) write \( in1 \) and \( in2 \) in the alu registers. \( RT1 \) subtracts these two values in order to possibly set the greater-than flag of the alu, \( flagGt \) (alu). This flag is passed to the \( creg \). After that, one of the two inputs is read from the input buffer again and written to the output buffer register by \( RT7 \) or \( RT5 \), depending on the value of \( flagGt \) in the \( creg \). Then this value is written to the output buffer by \( RT6 \) or \( RT4 \). In the RTG, both \( then \) (\( RT7 \) and \( RT6 \)) and \( else \) (\( RT5 \) and \( RT4 \)) functionality is present.

Currently, at FACTS side it is unknown how to deal with conditional constructs.
7.2 Interfacing

7.2.1 RTG Information

The problem with conditional constructs concentrates on how to deal with the condition present in the `creg`. On the one hand, the interface currently does nothing with the condition. On the other hand, at FACTS side it is currently unknown how to deal with conditional constructs. That implies, that the output generated by the interface contains all RT information, which is unrelated to the condition though. In other words, the functionality of the generated output is not correct.

7.2.2 Example

This example is included to show that the interface is at least capable of reading an RTG containing a conditional construct. The input description is the same as shown in Subsection 7.1.2:
#define WORD fix<16,15>

func main(in1, in2: WORD) out: WORD =
begin
    out = if in1 > in2 -> in1
         || in2
    fi;
end;

Subsection B.4.1 at 72 also shows this input description. The resulting output of the interface is shown in Subsection B.4.2.

The resulting output is accepted by FACTS. However, because of the fact that nothing is done with the condition, FACTS schedules the RT’s using all their mutual constraints. This means that RT7 and RT5 are not scheduled in the same clock cycle, because they have a resource constraint on ipb_l. The problem is that, dependent on the condition, only one of those two RT’s will be executed. Therefore, Mistral2 is able to schedule both RT’s in one clock cycle. The same holds for RT6 and RT4, which are considered by FACTS as if they have a resource constraint on opb_l.
Chapter 8

Larger Examples

This chapter discusses some larger examples. These examples contain combinations of the problems that have been explained in the previous chapters. Due to their size, the Mistral2 input descriptions of these examples and the FACTS input descriptions generated by the interface are not shown in this report.

8.1 'IDCT'

The IDCT example is an Inverse 1-D Discrete Cosine Transform. The input description contains no loops nor conditional constructs. The example contains 88 RT's. In the FACTS input description, these RT's are all present in one DFG. This input description is accepted by FACTS. Mistral2 generates a schedule that contains 37 cycles. FACTS however can find a schedule that uses 35 cycles to execute the function.

8.2 'FIR2'

The input description contains an initialise DFG and a main DFG. Both DFG’s contain a loop. For this example, we discuss the results for these two loop bodies.

The loop body within the initialise part contains four RT’s. The input description generated by the interface is accepted by FACTS. Mistral2 and FACTS both find a schedule of two cycles for this loop body.

The loop body within the main part contains 14 RT’s. The input description generated by the interface is accepted by FACTS. Mistral2 and FACTS both find a schedule of five cycles for this loop body.

8.3 'ALG_5.4_1'

The input description contains one loop body. This loop body contains a block DFG. Furthermore, in the input description several variables are reassigned. The loop body itself contains ten RT’s, including the one RT present in the block DFG. The input description generated by the interface is accepted by FACTS. Mistral2 and FACTS both find a schedule of six cycles.
Chapter 9

Conclusions

We have designed a software interface from Mistral2 to FACTS. This interface is capable of reading a binary RTG created by Mistral2 v1.4 rev2 and transferring the necessary information to a textual input format for FACTS.

The interface has been tested using a broad range of Mistral2 examples. On the one hand, the functionality of the interface has been visually verified using small examples. The small examples can be divided in four categories: basic blocks, hierarchical examples, loops and conditional constructs. On the other hand, the robustness of the interface has been verified using larger examples.

• The interface is capable of dealing correctly with basic block examples. The output generated by the interface is accepted by FACTS. The architecture information of the Mistral2 processor is correctly transferred to the FACTS input format. Furthermore, the functionality of the Mistral2 processor is transferred correctly to the FACTS input format.

• The interface is capable of dealing correctly with hierarchical examples. The hierarchy, as present in the Mistral2 binary data structure, is correctly transferred to the FACTS input format. Furthermore, hierarchical problems, like transporting variables between different hierarchical levels, are dealt with correctly.

• The interface is capable of dealing with loops in a right way. It correctly adds extra architecture and functionality information to the FACTS input format regarding the merging and splitting of non-feedback data edges and feedback data edges.

• The current version of the interface is not capable of dealing correctly with conditional constructs. On the one hand, this is due to the fact that at FACTS side it is not clear what to do with conditional constructs. On the other hand, this is due to the fact that the interface is not able to handle the specific problems related to conditional constructs. The generated output is accepted by FACTS, but the functionality as present in the generated output is not equal to the functionality as present in the Mistral2 input description.

The interface is capable of dealing correctly with larger examples, that is, without conditional constructs. The computing time needed by the interface to perform a translation is negligibly small compared to the computing time needed by Mistral2 to generate a binary RTG. The functionality of the generated output has not been verified as correct.

Many larger examples contain conditional constructs. In order to let the current version of the interface generate a functionally correct output, these constructs must be deleted from the Mistral2
input description. This means that at the moment, every input description must be checked for conditional constructs and must be possibly modified. This is not desired for the future. Therefore, both at FACTS side and at the interface side, more clearness is needed about conditional constructs.

Finally, another issue is the translation of FACTS information back to Mistral2. In order to do this, the results of FACTS must be translated to Mistral2 pragmas that can control the Mistral2 scheduler.
Bibliography


[FD98] Frontier Design
MISTRAL2: FUNCTIONAL SPECIFICATION FOR THE RELEASE OF A TOOL
Version 1.6, Leuven
13 May 1998


Appendix A

Interface Usage Manual

The interface has been tested using binary RTG's created with Mistral2 version 1.4 rev2. The interface is not capable of reading binary RTG's created by newer versions.

In this version of the interface, conditional constructs are not supported. That means that the output generated by the interface is not reliable if the Mistral2 input description contains conditional constructs. Therefore, it is better to get rid of all conditional constructs in the Mistral2 input description before generating a binary RTG.

After Mistral2 has generated a binary RTG and the interface has load that RTG, the binary data structure of the RTG is present in the memory. At this point, the interface inquires the data structure for information. Figure A.1 shows this.

![Diagram showing architecture, memory declarations, register declarations, and transfer graph](image)

Figure A.1: Inquire Datastructure
Appendix B

Examples

B.1 Basic Blocks

B.1.1 Mistral2 Input Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function Tree</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>#define WORD fix&lt;16,15&gt;</code></td>
</tr>
<tr>
<td></td>
<td>func main(in1, in2, in3, in4: WORD) out: WORD =</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>tmp1 = in2 * in1;</td>
</tr>
<tr>
<td></td>
<td>tmp2 = in4 + in3;</td>
</tr>
<tr>
<td></td>
<td>out = tmp2 - tmp1;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
<tr>
<td><strong>Reassignment</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>int main() {</td>
</tr>
<tr>
<td></td>
<td>int x = 0;</td>
</tr>
<tr>
<td></td>
<td>int y1 = x + 1;</td>
</tr>
<tr>
<td></td>
<td>x = 1;</td>
</tr>
<tr>
<td></td>
<td>int y2 = x + y1;</td>
</tr>
<tr>
<td></td>
<td>return y2;</td>
</tr>
<tr>
<td><strong>IPB Use</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>#define WORD fix&lt;16,15&gt;</code></td>
</tr>
<tr>
<td></td>
<td>func main(in1, in2, in3: WORD) out: WORD[2] =</td>
</tr>
<tr>
<td></td>
<td>begin</td>
</tr>
<tr>
<td></td>
<td>tmp1 = in1 + in2;</td>
</tr>
<tr>
<td></td>
<td>out[0] = tmp1 * in3;</td>
</tr>
<tr>
<td></td>
<td>out[1] = tmp1 + in3;</td>
</tr>
<tr>
<td></td>
<td>end;</td>
</tr>
</tbody>
</table>
### B.1.2 Interface Output for 'Function Tree'

```plaintext
// special modtypes

type ENTRY_mod {
    delay 0;
    di 0;
    instances 1;
    is_entry_type int 1;
}

type EXIT_mod {
    delay 0;
    di 0;
    instances 1;
    is_exit_type int 1;
}

// modtypes

type alu_l {
    delay 1;
    di 1;
    instances 1;
}

type mult_l {
    delay 1;
    di 1;
    instances 1;
}

type mult_l_stage_2 {
    delay 1;
    di 1;
    instances 1;
}

type ipb_l {
    delay 1;
    di 1;
    instances 1;
}

type opb_l {
    delay 1;
    di 1;
    instances 1;
}

type mux_2_alu_l {
    delay 1;
    di 1;
    instances 1;
}

type mux_2_mult_l {
    delay 1;
    di 1;
    instances 1;
}

//-------------------------------
```
// memtypes (no registers)

// memtypes (registers)
memtype reg_1_opb_l
  @capacity 1;
  @access random;
}

memtype reg_2_mul_t_l
  @capacity 1;
  @access random;
}

memtype reg_1_mul_t_l
  @capacity 1;
  @access random;
}

memtype reg_2_alu_l
  @capacity 2;
  @access random;
}

memtype reg_l_alu_1
  @capacity 1;
  @access random;
}

memtype reg_1_mul_t_1
  @capacity 1;
  @access random;
}

// dfg's

// dfg init

// dfg main

RT5: in2_9 = ipb_l(in2)
  RT5: mux_l_mul_t_l
    @node_attr RTS op_type str "mem_read";
    @membind(in2_9, reg_1_mul_t_l);
  RT4: in1_8 = ipb_l(in1)
    RT4: mLx_2_mul_t_l()
    @ncde_attr RT4 op_type str "mem_read";
    @membind(in1_8, reg_2_mul_t_l);
  RT8: in4_11 = ipb_l(in4)
    RT8: mux_l_alu_l()
    @node_attr RTB op_type str "mem_read";
    @membind(in4_11, reg_l_alu_1);
  RT2: tmp_l = mul_l(in1_8, in2_9);
    @node_attr RT2 op_type str "mpy";
    @membind(tmp_l, preg_1_mul_t_l);
    @dist(RT2, RTJ) = 1;
  RT3: tmp_l3 = mul_l-stage_2(tmp_l)
    RT3: mux_l_alu_l()
    @membind(tmp_l3, reg_2_alu_l);
  RT6: tmp_l6 = alu_l(in1_10, in4_11)
    RT6: mux_l_alu_l()
    @node_attr RT6 op_type str "add";
    @membind(tmp_l6, reg_l_alu_1);
  RT1: out_12 = alu_l(tmp_l3, tmp_l6)
    RT1: mux_l_opb_l()
    @node_attr RT1 op_type str "mem_write";

B.1.3 Interface Output for 'Reassignment'

```
// Interface Output for 'Reassignment'

// special motypes
motype ENTRY_mod { 
  @delay 0;
  @dii 0;
  @instances 1;
  @attr is_entry_type int 1;
}

motype EXIT_mod { 
  @delay 0;
  @dii 0;
  @instances 1;
  @attr is_exit_type int 1;
}

// motypes
motype alu_l { 
  @delay 1;
  @dii 1;
  @instances 1;
}

motype opl_l { 
  @delay 1;
  @dii 1;
  @instances 1;
}

motype romctrl_l { 
  @delay 1;
  @dii 1;
  @instances 1;
}

motype mux_l_opb_l { 
  @delay 1;
  @dii 1;
  @instances 1;
}

motype mux_2_alu_l { 
  @delay 1;
  @dii 1;
  @instances 1;
}

motype mux_l_alu_l { 
  @delay 1;
  @dii 1;
  @instances 1;
}

// memtypes (no registers)

// memtypes (registers)
memtype reg_l_opb_l { 
  @capacity 1;
  @access random;
}

memtype reg_2_alu_l { 
  @capacity 1;
  @access random;
}

memtype reg_l_alu_l { 
  @capacity 1;
  @access random;
}
```

56
dfg init () () {
}

dfg main () () {
  RT3: \_1\_5 = \text{romctrl}\_1\_int\_32\_1\_l | \text{mux}\_2\_alu\_l11;
  @node_attr RT3 op_type str "mem\_read";
  @membind(_l\_5, reg\_2\_alu\_l);

  RT4: x = \text{romctrl}\_1\_int\_32\_0\_l | \text{mux}\_1\_alu\_l1;
  @node_attr RT4 op_type str "mem\_read";
  @membind(x, reg\_1\_alu\_l);
  @dist(RT4, RT5) >= 0;

  RT5: x\_l = \text{romctrl}\_1\_int\_32\_1\_l | \text{mux}\_1\_alu\_l0;
  @node_attr RT5 op_type str "mem\_read";
  @membind(x\_l, reg\_1\_alu\_l);

  RT2: y\_l = \text{alu}\_l\_l\_5, x\_l | \text{mux}\_2\_alu\_l11;
  @node_attr RT2 op_type str "add";
  @membind(y\_l, reg\_2\_alu\_l);
  @dist(RT2, RT3) >= 0;

  RT1: return\_l = \text{alu}\_l\_l\_y\_l, x\_l | \text{mux}\_1\_opb\_l11;
  @node_attr RT1 op_type str "add";
  @membind(return\_l, reg\_1\_opb\_l);

  RT6: return = \text{opb}\_l(return\_l);
  @node_attr RT6 op_type str "mem\_write";
}

57
B.1.4 Interface Output for 'IPB Use'

```
// Interface Output for 'IPB Use'

// RT2C v0.0 (Facts 2) format for design : m3
// generated by Mistral2 : v9.2.3.3 Wed Jun 24 17:29:03 MET DST 1998
// date : Tue Aug 17 15:00:58 1999

// special modtypes
modtype ENTRY_mod {
    @delay 0;
    @di 0;
    @instances 1;
    @attr is_entry_type int 1;
}
modtype EXIT_mod {
    @delay 0;
    @di 0;
    @instances 1;
    @attr is_exit_type int 1;
}

// modtypes
modtype alu_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtype mult_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtype mult_l_stage_2 {
    @delay 1;
    @di 1;
    @instances 1;
}
modtype ipb_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtype pipb_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtype mulpb_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtypemux_2_mult_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtypemux_l_mult_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtypemux_2_alu_l {
    @delay 1;
    @di 1;
    @instances 1;
}
modtypemux_l_alu_l {
    @delay 1;
    @di 1;
    @instances 1;
}
```
## B.2 Hierarchy

### B.2.1 Mistral2 Input Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Description</th>
</tr>
</thead>
</table>
| Blocks1| #define WORD fix<16,15>  
func main() out: WORD[2] =  
begin 
  out[0] = 2;  
begin 
    out[1] = out[0] - 1;  
end;  
end; |
| Blocks2| #define WORD fix<16,14>  
func main() out: WORD[3] =  
begin 
  out[0] = 2;  
begin 
    out[1] = out[0] - 1;  
    out[2] = out[0] + out[1];  
end;  
end; |
B.2.2 Interface Output for 'Blocks1'

```plaintext
// special roodtypes
modtype ENTTY_mod {
    @delay 0;
    @dii 0;
    @instances 1;
    @attr is_entry_type int 1;
}
modtype EXIT_mod {
    @delay 0;
    @dii 1;
    @instances 1;
    @attr is_exit_type int 1;
}

// motypes
modtype alu_l {
    @delay 1;
    @dii 1;
    @instances 1;
}
modtype romctrl_l {
    @delay 1;
    @dii 1;
    @instances 1;
}
modtype ram_l {
    @delay 1;
    @dii 1;
    @instances 1;
}
modtype mux_2_ram_l {
    @delay 1;
    @dii 1;
    @instances 1;
}
modtype ram_2_alu_l {
    @delay 1;
    @dii 1;
    @instances 1;
}
modtype mux_l_alu_l {
    @delay 1;
    @dii 1;
    @instances 1;
}

// memotypes (no registers)

// memotypes (registers)
```

---

61
dfg_init (())

// dfg's

dfg block (out) () {
    RT1: addr_9 = romctrl_l(int_2_0) | max_l_ram_l();
    @node_attr RT1 op_type str "mem_read";
    @membind(addr_9, reg_2_ram_l);
    RT2: out_0_6 = ram_l(addr_9, out) | max_l_alu_l();
    @node_attr RT2 op_type str "mem_read";
    @membind(out_0_6, reg_l_alu_l);
    RT3: _1_7 = romctrl_lfix_16_15_b(00000000000000000000000000000000) | max_l_alu_l();
    @node_attr RT3 op_type str "mem_read";
    @membind(_1_7, reg_l_alu_l);
    RT4: out_l_out_0_1_8 = alu_l(_1_7, out_0_6) | max_l_ram_l();
    @node_attr RT4 op_type str "sub";
    @membind(out_l_out_0_1_8, reg_l_ram_l);
    RT5: addr_lO = romctrl_l(int_2_1) | max_l_ram_l();
    @node_attr RT5 op_type str "mem_read";
    @membind(addr_lO, reg_2_ram_l);
    @edge_rename out "";
    RT6: out = ram_l(addr_lO, out_l_out_0_1_8);  
    @node_attr RT6 op_type str "mem_write";
}

dfg main (())

// dfg's

RT1: out_0_2_3 = romctrl_l(int_2_0) | max_l_ram_l();
    @node_attr RT1 op_type str "mem_read";
    @membind(out_0_2_3, reg_2_ram_l);
    RT2: addr_l1 = romctrl_l(int_2_0) | max_l_ram_l();
    @node_attr RT2 op_type str "mem_read";
    @membind(addr_l1, reg_2_ram_l);
    RT3: out = ram_l(addr_l1, out_0_2_3);  
    @node_attr RT3 op_type str "mem_write";
    RT4: block: block(out);
B.2.3 Interface Output for 'Blocks2'

// Interface Output for 'Blocks2'

// special roodtypes
modtype ENTRY_mod ( 
  @delay 0; 
  @dii 0; 
  @instances 1; 
  @attr is_entry_type int 1; 
)

modtype EXIT_mod ( 
  @delay 0; 
  @dii 0; 
  @instances 1; 
  @attr is_exit_type int 1; 
)

// roodtypes
roodtype alu_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

roodtype romctrl_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

roodtype ram_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

roodtype mux_2_alu_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

roodtype mux_l_ram_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

modtype mux_2_alu_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

modtype mux_l_ram_l ( 
  @delay 1; 
  @dii 1; 
  @instances 1; 
)

// memtypes ( registers)
memtype reg_2_ram_l ( 
  @capacity 1; 
  @access random; 
)

memtype reg_1_ram_l ( 
  @capacity 1; 
  @access random; 
)

memtype reg_2_alu_l ( 
  @capacity 1; 
  @access random; 
)

memtype reg_1_alu_l ( 
  @capacity 1; 
  @access random; 
)

memtype reg_2_2 RAM_l ( 
  @ capacity 1; 
  @access random; 
)

memtype reg_1_2 RAM_l ( 
  @ capacity 1; 
  @access random; 
)

memtype reg_2_1 RAM_l ( 
  @ capacity 1; 
  @access random; 
)

memtype reg_1_1 RAM_l ( 
  @ capacity 1; 
  @access random; 
)
dfg main () { }
### B.3 Loops

#### B.3.1 Mistral2 Input Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Input Description</th>
</tr>
</thead>
</table>
| Simple Loop| ```
#define WORD fix<16,15>  

func main() out: WORD[10] =  
begin  
  x = 1;  
  (i:0..9):  
    out[i] = x;  
end;
``` |
| Loop       | ```
#define WORD fix<16,15>  

func main() out: WORD[10] =  
begin  
  out[0] = 0;  
  (i:1..9):  
    out[i] = out[i - 1];  
end;
``` |
| Loop in Loop| ```
#define WORD fix<16,15>  

func main() out: WORD[10][10] =  
begin  
  (i:0..9):  
    (j:0..9):  
      out[i][j] = 0;  
end;
``` |
B.3.2 Interface Output for 'Simple Loop'

//------------------------------
// RTIC v0.0 (Facts 2) format for Design : ms2

// special modtypes
modtype ENTRY_mod { 
@delay 0;
@delay 0;
@instances 1;
@attr is_entry_type int 1;
}
modtype EXIT_mod { 
@delay 0;
@delay 0;
@instances 1;
@attr is_exit_type int 1;
}

// modtypes
modtype acu_l { 
@delay 1;
@delay 1;
@instances 1;
}
modtype opb_l { 
@delay 1;
@delay 1;
@instances 1;
}
modtype romctrl_l { 
@delay 1;
@delay 1;
@instances 1;
}
modtype mux_l_opb_l { 
@delay 1;
@delay 1;
@instances 1;
}
modtype mux_l_acu_l { 
@delay 1;
@delay 1;
@instances 1;
}

// memtypes (no registers)

// memtypes (registers)
memtype reg_l_opb_l { 
@capacity 1;
@access random;
}
memtype reg_l_acu_l { 
@capacity 1;
@access random;
}
memtype acu_l_FlagEq2 { 
@capacity 1;
@access random;
}

// dfg's
dfg init [] {} {}
dfg loop_i (x, i) // { // FOR (loop_i, 4, 4, 1, i, 0, 9, 13, 15) @node_attr RT3 op_type str 'mem_write'; RT3: out = opb_l(x); i ENTRY_O: i_ENTRY_o_out = ENTRY_mod(i, i_iteration); RT3: i_ENTRY_o_out = ENTRY_mod(i, i_iteration); i ENTRY_L: i_EXIT_l_in = acu_l ENTRY_o_out) | mux_l acu_l(1); @node_attr RT1 op_type str 'decx'; @memhind (i_EXIT_l_in, reg_l acu_l); RT1: i_EXIT_l_out, i_iteration = EXIT_mod(i_EXIT_l_in); RT1: i EXIT_O: i_EXIT_O_out, i_EXIT_l_out = EXIT_mod(i_EXIT_O_in); // FOR_END (loop_i) } dfg main () () // { RT1: i = romctrl_l_int_3_15 | mux_l acu_l(1); @node_attr RT2 op_type str 'mem_read'; @membind {i, reg_l acu_l}; RT2: x = romctrl_l fix_16_15 | membind {x, reg_l acu_l}; @node_attr RT2 op_type str 'mem_read'; @membind (x, reg_l acu_l); RT_loop_i: loop_i (x, i); }
B.3.3 Interface Output for 'Loop'

```plaintext
// -----------------------------------------------
// RTIC v0.5 [Ena2] format for design .sal
// date Fri Aug 13 13:22:07 1999
// -----------------------------------------------
// special modtypes
modtype ENTRY_mod {
    @delay 1;
    @dli 0;
    @instances 1;
    @attr is_entry_type int 1;
}
modtype EXIT_mod {
    @delay 0;
    @dli 0;
    @instances 1;
    @attr is_exit_type int 1;
}

// -----------------------------------------------
// modtypes
modtype acu_l {
    @delay 1;
    @dli 1;
    @instances 1;
}
modtype rornctrl_l {
    @delay 1;
    @dli 1;
    @instances 1;
}
modtype ram_l {
    @delay 1;
    @dli 1;
    @instances 1;
}
modtype mux_2_ram_l {
    @delay 1;
    @dli 1;
    @instances 1;
}
modtype mux_l_rar.~_l {
    @delay 1;
    @dli 1;
    @instances 1;
}
modtype mux_l_acu_l {
    @delay 1;
    @dli 1;
    @instances 1;
}

// -----------------------------------------------
// memtypes (na registers)

// -----------------------------------------------
// memtypes (registers)
memtype reg_2_ram_l {
    @capacity 1;
    @access random;
}
memtype reg_l_ram_l {
    @capacity 1;
    @access random;
}
memtype reg_l_acu_l {
    @capacity 1;
    @access random;
}
memtype acu_l_FlagEql {
    @capacity 1;
    @access random;
}
```

68
// dfg's

dfg init () () {
}
dfg loop_i (addr_18, out, addr_18, i) () {
// FOR(loop_i, 5, 6, 1, i_0_14_22)
RT9_ENTRY_O: addr_lB ENTRY_O_out = ENTRY_mod(addr_l8 ENTRY_O, out ENTRY_O out, out ENTRY_O out) | max l ram l();
@node attr RT9 op type str 'mem_read';
@membind (out ENTRY_O out, reg l ram l); RT9_ENTRY_l: out ENTRY_l_out = ENTRY_mod(out ENTRY_l, out ENTRY_l, out ENTRY_l); RT9: out out = ram_l(addr l8 ENTRY_O out, out ENTRY_l out) I
mux_l ram l();
@node_attr RT9 op type str 'mem_write'; RT9: out out = romctrl_l (_int_S_O_) | mux l ram l();
@node_attr RT9 op type str 'mem_read'; @membind (out_0_0_3, reg l ram l); RTI: addr_21 = romctrl_l (_int_S_O_) | mux_2 ram l();
@node attr RTl op type str 'mem_read'; @membind (addr_21, reg_2 ram l); RTS: i = romctrl_l(_int_5_9_) | mux l acu l();
@node attr RTS op type str 'mem_write';
@membind(i, reg l acu l); RTI: out = romctrl_l(_int_9_5_) | mux l ram l();
@node attr RTI op type str 'mem_read'; @membind(i, reg l acu l); RTI: out = romctrl_l(_int_9_5_) | mux l ram l();
@node attr RTI op type str 'mem_write';
@membind(i, reg l acu l); RT2: out = romctrl_l(_int_5_9_) | max l ram l();
@node attr RT2 op type str 'mem_read'; @membind(out_0_0_3, reg l ram l); RT3: addr_21 = romctrl_l(_int_5_9_) | max l ram l();
@node attr RT3 op type str 'mem_read'; @membind(addr_21, reg_2 ram l); RT3: addr_18, addr_18, i = romctrl_l(_int_5_9_) | max l ram l();
@node attr RT3 op type str 'mem_write';
@membind(addr_18, reg l ram l); RT1: loop_i: loop i addr l8, out, addr_18, i);
B.3.4 Interface Output for 'Loop in Loop'

 '//------------------------------------------------------------------------------
// special roodtypes

modtype ENTRY_mod
    @delay 0;
    @dii 0;
    @instances 1;
    @attr is_entry_type int 1;
}

modtype EXIT_mod
    @delay 0;
    @dii 0;
    @instances 1;
    @attr is_exit_type int 1;
}

//------------------------------------------------------------------------------
// roodtypes

roodtype acu_l
    @delay 1;
    @dii 1;
    @instances 1;

roodtype opb_l
    @delay 1;
    @dii 1;
    @instances 1;

roodtype romctrl_l
    @delay 1;
    @dii 1;
    @instances 1;

roodtype mux_l_opb_l
    @delay 1;
    @dii 1;
    @instances 1;

roodtype mux_l_acu_l
    @delay 1;
    @dii 1;
    @instances 1;

//------------------------------------------------------------------------------
// memtypes (registers)

memtype reg_l_opb_l
    @capacity 1;
    @access random;

memtype reg_l_acu_l
    @capacity 2;
    @access random;

memtype acu_l_FlagEql
    @capacity 1;
    @access random;

//------------------------------------------------------------------------------
// dfg's

dfg init () ()

dfg loop_j () {  
  // FOR loop_j: 6, j, ___, 5, 19, 24, 10)  
  RT7: out_i_j_O_S = romctrl_l[1]_fix14_15_Dec0_0000000000000000 | max_i_mph_l(i);  
  @node_attr RT7 op_type str 'mem_read';  
  @membind(out_i_j_O_S, reg1_mph);  
  RT7 ENTRY_O: j ENTRY_O_out = ENTRY_mod(j, j_iteration);  
  RT7 j_O_l9_24_EXIT_O_in, j_O_l9_24_iteration = EXIT_mod(j_O_l9_24_EXIT_O_in);  
  RT7_EXIT_O: j.EXIT_O_out, j iteration = EXIT_mod(j.EXIT_O_in);  
  RT7: out = opb_l(out_i_j_O_S);  
  @node_attr RT7 op_type str 'mem_write';  
  // FOR_END loop_j  
}

dfg loop_i () {  
  // FOR loop_i: 6, i, ___, 5, 12, 23, 10)  
  RT8: j = romctrl_l[4]_int8_15 | max_i_mph_l(i);  
  @node_attr RT8 op_type str 'mem_read';  
  @membind(j, reg1_mph);  
  RT8 loop_j: loop_j(i);  
  RT8 ENTRY_O: i ENTRY_O_out = ENTRY_mod(i, i_iteration);  
  RT8 i_O_l2_23_EXIT_O_in, i_O_l2_23_iteration = EXIT_mod(i_O_l2_23_EXIT_O_in);  
  RT8_EXIT_O: i.EXIT_O_out, i iteration = EXIT_mod(i.EXIT_O_in);  
  // FOR_END loop_i  
}

dfg main () {  
  RT: i = romctrl_l[1]_int8_15 | max_i_mph_l(i);  
  @node_attr RT op_type str 'mem_read';  
  @membind(i, reg1_mph);  
  RT loop_i: loop_i();  
}  

B.4 Conditional Constructs

B.4.1 Mistral2 Input Description for 'IF'

#define WORD fix<16,15>

func main(in1, in2: WORD) out: WORD =
begin
    out = if in1 > in2 -> in1
         || in2
    fi;
end;

B.4.2 Interface Output for 'IF'

modtype ENTRY_mod {
    @delay 0;
    @dii 0;
    @instances 1;
    #attr is_entry_type int 1;
}

modtype EXIT_mod {
    @delay 0;
    @dii 0;
    @instances 1;
    #attr is_exit_type int 1;
}

modtype alu_l {
    @delay 1;
    @dii 1;
    @instances 1;
}

modtype ipb_l {
    @delay 1;
    @dii 1;
    @instances 1;
}

modtype opb_l {
    @delay 1;
    @dii 1;
    @instances 1;
}

modtype rnux_l_opb_l {
    @delay 1;
    @dii 1;
    @instances 1;
}

modtype mux_2_alu_l {
    @delay 1;
    @dii 1;
    @instances 1;
}

modtype mux_l_alu_l {
    @delay 1;
    @dii 1;
    @instances 1;
}
// memtypes (no registers)

// memtypes (registers)
memtype reg_l_opb_l {
  capacity 1;
  access random;
}
memtype reg_l_alu_l {
  capacity 1;
  access random;
}
memtype reg_2_alu_l {
  capacity 1;
  access random;
}
memtype alu_l_FlagGt {
  capacity 1;
  access random;
}

// dfg's

dfg main () {} {
  dfg init () {} {
    dfg main () {} {
      RT2: in2_9 = ipb_l(in2) | in1_7; @node_attr RT2 op_type str "mem_read";
      @membind(in2_9, reg_2_alu_l);
      RT3: inl_7 = ipb_l(in1) | in2_9; @node_attr RT3 op_type str "mem_read";
      @membind(inl_7, reg_l_alu_l);
      RT4: out_l0 = opb_l(out_l); @node_attr RT4 op_type str "mem_write";
      @edge_rename out "";
      RT5: out = opb_l(out_l0); @node_attr RT5 op_type str "mem_write";
      // Renaming value...
      @edge_rename out "";
      RT6: out = opb_l(out_l0); @node_attr RT6 op_type str "mem_write";
      // Renaming value...
      @edge_rename out "";
      RT7: out_lC = opb_l(out); @node_attr RT7 op_type str "mem_write";
      @edge_rename out "";
      RT8: out = opb_l(out_lC); @node_attr RT8 op_type str "mem_write";
      // Renaming value...
      @edge_rename out "";
      RT9: out = opb_l(out_lC); @node_attr RT9 op_type str "mem_write";
      // Renaming value...
      @edge_rename out "";
    } // edge_rename out ""
  } // edge_rename out ""
} // edge_rename out ""