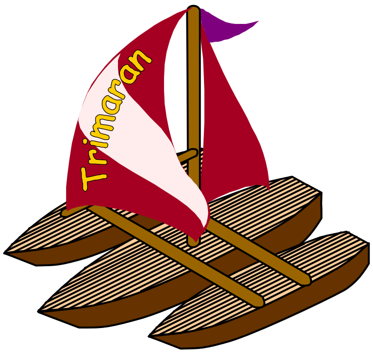


The Elcor Intermediate Representation

1. Introduction

The Trimaran back-end (Elcor) uses the Elcor Intermediate Representation (**The Elcor IR**) to represent a program unit. A program unit consists of a graph of operations connected by edges. This operation graph represents both, a traditional control flow graph and a data flow graph. The edges between operations model various kinds of control flow, data and memory dependences. The Elcor IR provides the necessary infrastructure to build, manipulate and traverse this graph.

In addition, it provides mechanisms to represent:

- The data section in a program unit, e.g. global symbols, arrays, literal pools, etc.
- Predicated execution. Execution of operations can be guarded by predicate operands. This is used to model predicated architectures such as the **HPL-PD**.
- Hierarchical non-overlapping region structures (a tree). Such regions are used to set scope for program analysis and for optimizations such as instruction scheduling, register allocation. A region structure is defined over the operation graph. The root of the tree is the program unit, e.g. a procedure. The leaf nodes of the region are operations.
- EPIC related information. The IR has mechanisms to represent scheduling and machine resource usage information explicitly inside an operation.
- Expanded virtual registers (EVRs). EVRs allow multiple values from a sequence of assignments to be live at the same time. This is particularly useful in the dependence analysis of iterative loops.

This document gives an introduction to using this rich infrastructure. The structure of the Elcor IR and its programming interface have been described in the sections that follow, with the aid of diagrams. Links to files in Trimaran's source code have been provided wherever necessary.

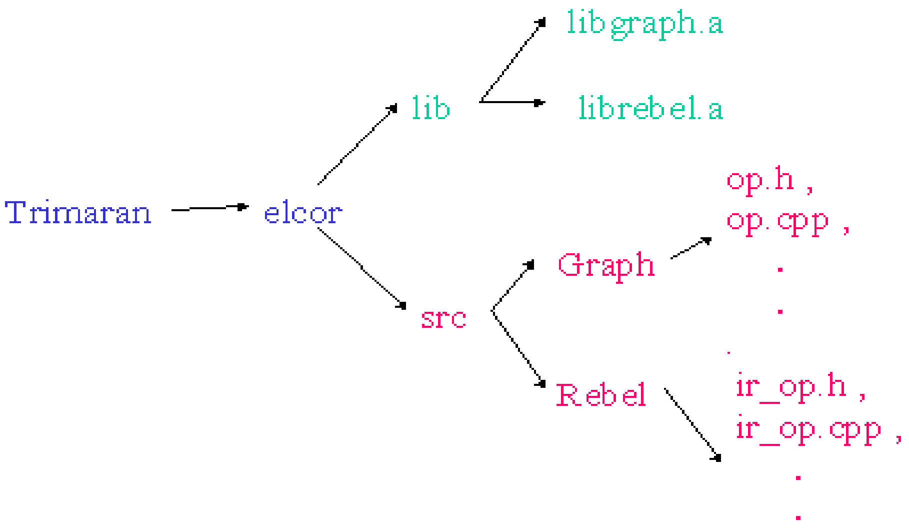
2. Internal and Textual Representation

The internal representation of the Elcor IR consists of a set of C++ objects. All optimization modules in the Elcor IR use the interface provided by these objects to carry out optimizations. Optimizations are simply IR to IR transformations. The simulator also uses this interface to generate out executable object code. The class interface to these objects in described in more detail in subsequent sections.

The Elcor IR also has a textual representation, known as **Rebel**. A reader procedure is provided that reads Rebel and constructs the corresponding internal program representation. A writer procedure is provided for generating Rebel from the internal representation. Rebel is also described in more detail later in this document.

3. The Elcor IR source tree structure

The **trimaran/elcor/src/Graph/** directory contains source code that implements the C++ class interface provided by Elcor IR. The .h files contain the interface and .cpp files contain the implementation



The **trimaran/elcor/src/Rebel/** directory contains source code that implements the Rebel reader and writer functions.

trimaran/elcor/lib has libraries that contain the object code of the various components of elcor. Add-on modules to elcor need to link **libgraph.a** to be able to use the Elcor IR. **librebel.a** is to be linked if an add-on module reads or writes a Rebel file.

The above source code also uses several data structures such as hash tables, linked lists, etc. These are available in the library **trimaran/elcor/lib/libtools.a**. Their implementation is present in **trimaran/elcor/src/Tools**.

4. The Internal Representation

Objects of the **Op** (i.e. operation) class, the **Operand** class, the **Edge** class, the **Region** class, the **Compound_region** class, and the **Attribute** class form the main components of any graph in Elcor IR. The contents of these classes will be described in this section.

4.1 The Op class

An object of the Op class represents one operation in the graph. The operation can be a machine operation i.e. an operation that gets executed by the host architecture (or simulator) or it can be a compiler operation, also known as a pseudo operation. Such operations are not part of the program execution stream but often are inserted by the compiler to hold internal information in an operation form for convenience. CONTROL_MERGE (used to denote a merge of two control paths), DEFINE (used to assign some value to an internal compiler variable) are two examples of compiler operations.

An Elcor operation is of the form :

dest1, … , destm = opcode (src1, … , srcn) if p

dest1, … , destm represent the destination operands,
src1, …, srcn represent the source operands,
opcode is the machine opcode,
p is the predicate operand.

The Op class represents this operation and provides access to operands present in it. Operands are simply objects of the Operand class (described in section **4.2**).

Following are some other facilities provided by the Op class. There are methods to:

- Find the number of operands of each type.
- Query various operation latencies, such as the flow dependence latency and the anti dependence latency.
- Add or remove incoming and outgoing edges. Edges are objects of the Edge class (described in section **4.3**).
- Set the scheduling information of an operation. This is particularly useful for EPIC architectures.

At a conceptual level, the Op class can also be considered a region (introduced in section **1**). The Region class represents such a region and will be described in section **4.6**. Since the Op class is derived from Region, an operation inherits a region's functionality.

The C++ interface of the Op class is present in **trimaran/elcor/src/Graph/op.h**. Its implementation is present in **trimaran/elcor/src/Graph/op.cpp**.

Iterators over the Op class are defined in **trimaran/elcor/src/Graph/iterators.h**. Its implementation is present in **trimaran/elcor/src/Graph/iterators.cpp**. The iterators allow one to walk through the contents of an operation.

4.2 The Operand Class

An object of the Operand class represents an operand in an Elcor operation. An operand sits at specific ports in an operation. A port defines the exact position of an operand within the operation.

An operand can be any of the following type :

- **A Register.**
  A register can be either assigned or unassigned. An assigned register is one that has been bound to a machine (physical) register. It is unassigned (or virtual) otherwise. A register is bound to a Register File. A Register File is an aggregate of registers of a kind, e.g. an integer register file, a floating point register file, etc. A register file can further be either be static (containing static registers) or rotating (containing rotating registers). The HPL-PD architecture specification explains rotating register files in detail.

- **A Macro Register.**
  Macro registers are registers reserved by the compiler or the run-time system. Parameter passing registers, stack pointer, frame pointer, loop counter, epilogue stage counter etc. are a few examples of macro registers.

- **Memory registers.**
  Memory registers are used to encode memory dependence edges. For example, if a load operation follows a store operation, the store operation can define a memory register (at one of its destination port). The load operation can then use the same register (at one of its source ports). When memory dependence edges are drawn, the use of this register is detected creating a memory dependence edge between the two operations.

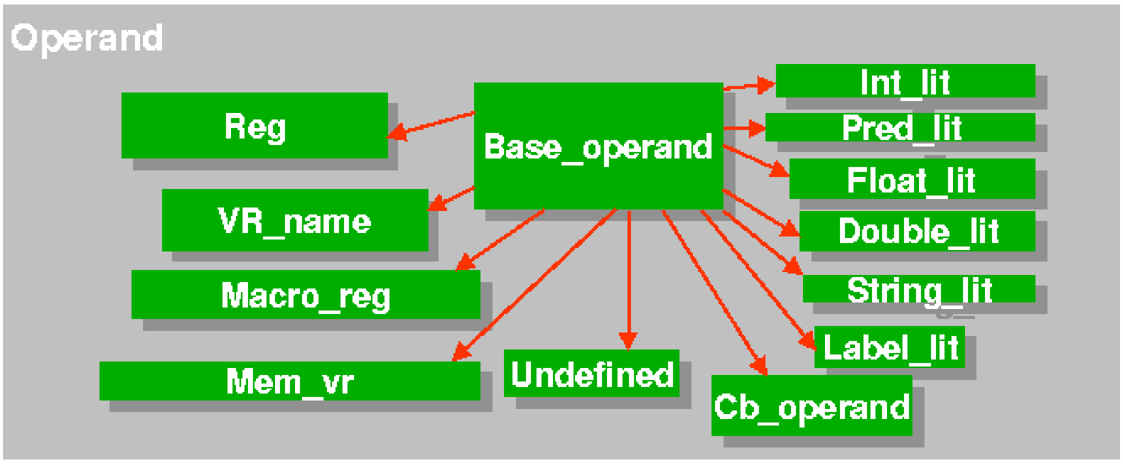
- **Register names.**
  Expanded virtual registers (introduced in Section 1) can be re-assigned to support the DSA (Dynamic Single Assignment) form.

- **Local branch targets.**
  These are just region IDs that appear as branch targets.

- **Literals.**
  Can either be integers, floating point numbers, double numbers, predicate literals (either true or false), strings, labels(such as a global variable name, procedure name, etc.).

- **Undefined.**
  This is just a place holder.

4.3 The Operand Class Hierarchy



All the operand types described above are derived from the Base_operand class. Operand class is a wrapper for all operands and contains Base_operand. The functions of the Operand class are to:

- Provide Boolean methods for testing the class (operand) type.
- Provide access methods to class (operand) specific fields.
- Provide comparison operators for comparing two operands.

The interface to the Operand class is present in **trimaran/elcor/src/Graph/operand.h**. The implementation of the class is present in **trimaran/elcor/src/Graph/operand.cpp**.

4.4 The Edge Class

The Edge class represents an edge in the IR graph. An edge in the graph models dependence constraints between operations. Edges can represent :

- Control dependences. The edge represents a sequential control flow.

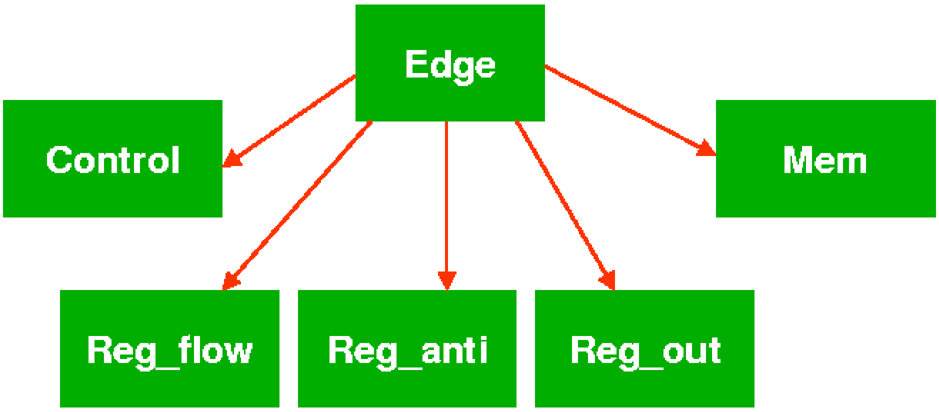
- Flow, anti and output dependences on registers i.e. data dependences.

- Flow, anti and output memory dependences classified as "certain" (when there is always a memory dependency) or "maybe" (when there may be a memory dependency).

An edge has two operations (pointers) on its ends ; the source operation and the destination operation.

An edge also contains more detailed reason for dependence represented in terms of the source and destination operand ports. The class also provides functions to set and query different latencies.

4.5 The Edge Class Hierarchy



Edge is an abstract base class. Other types of edges are derived classes of this class.

The interface to the Edge class is present in **trimaran/elcor/src/Graph/edge.h**. The implementation is present in **trimaran/elcor/src/Graph/edge.cpp**.

Iterators over the Op class to iterate through the edges in an operation are provided. The interface is present in **trimaran/elcor/src/Graph/iterators.h**.

Its implementation is present in **trimaran/elcor/src/Graph/iterators.cpp**.

As an example, the class Op_inedges iterator can be used to iterate through the incoming edges in an operation. The class Op_outedges can be used to iterate through the outgoing edges in an operation.

4.6 Regions

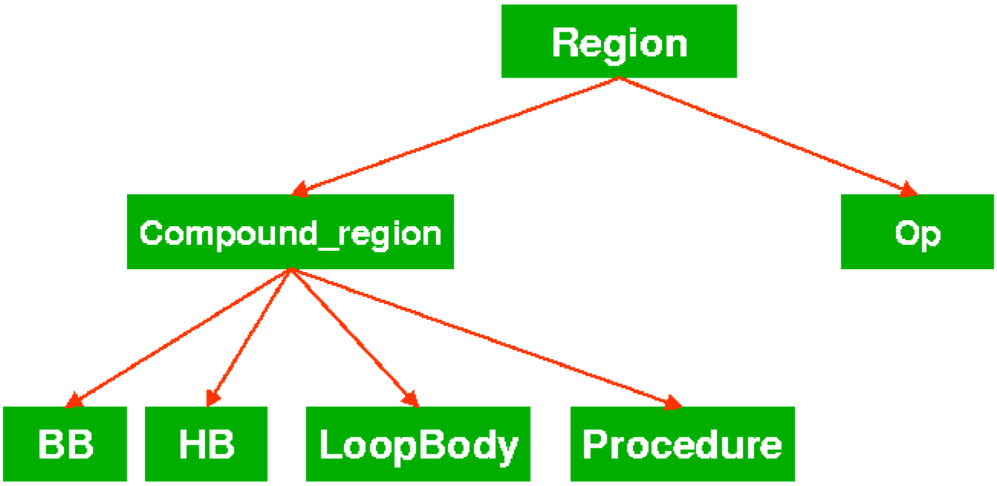
A Region in Elcor is a hierarchical non-overlapping region structures (a tree). Such regions are used to set scope for program analysis and for optimizations such as instruction scheduling, register allocation. A region structure is defined over the operation graph (tree). The root of the tree is the program unit, e.g. a procedure. The leaf nodes of the Region(tree) are operations.

A Region is defined by :

- Operations contained in the region.
- Set of control flow edges that enter or exit the region.
- Set of entry and exit operations (mostly redundant).
- All entry operations are CONTROL_MERGE operation.
- All exit operations are branch operations.
- There is a DUMMY_BRANCH pseudo operation if region exit is fall-through.

A Compound Region is all of the above except that it can contain other regions (recursively).

4.7 Region Class Hierarchy

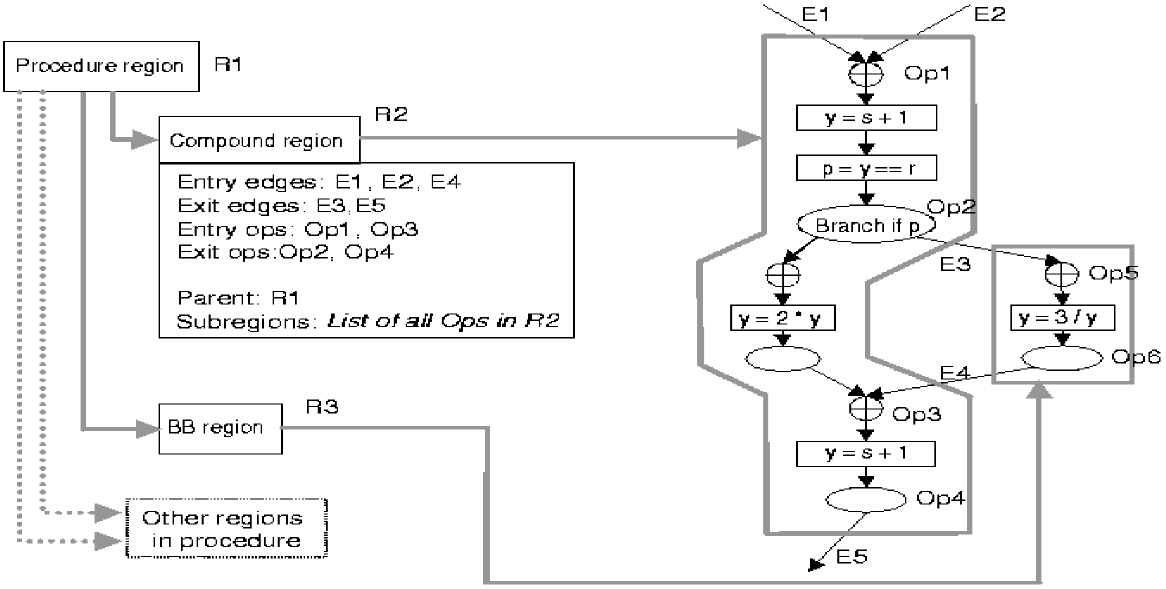


Region class is an abstract base class. A Compound regions is a region and can contain other regions in the region tree. Currently only regions shown in the above figure have been implemented. Basicblock(BB) is a single entry, single exit Compound Region with operations in it. Hyperblock(HB) is a single entry, multiple exit Compound Region with operations in it. Hyperblocks are constructed when aggressive instruction scheduling needs to be done. A LoopBody is a collection of other compound regions suitable for loop optimizations. A Procedure is the outermost compound region that encloses all other regions. It corresponds to the C procedure in the original source code. For full generality, an operation (Op class) is also defined as a region (inherits Region) but is not a compound region.

4.8 Region representation

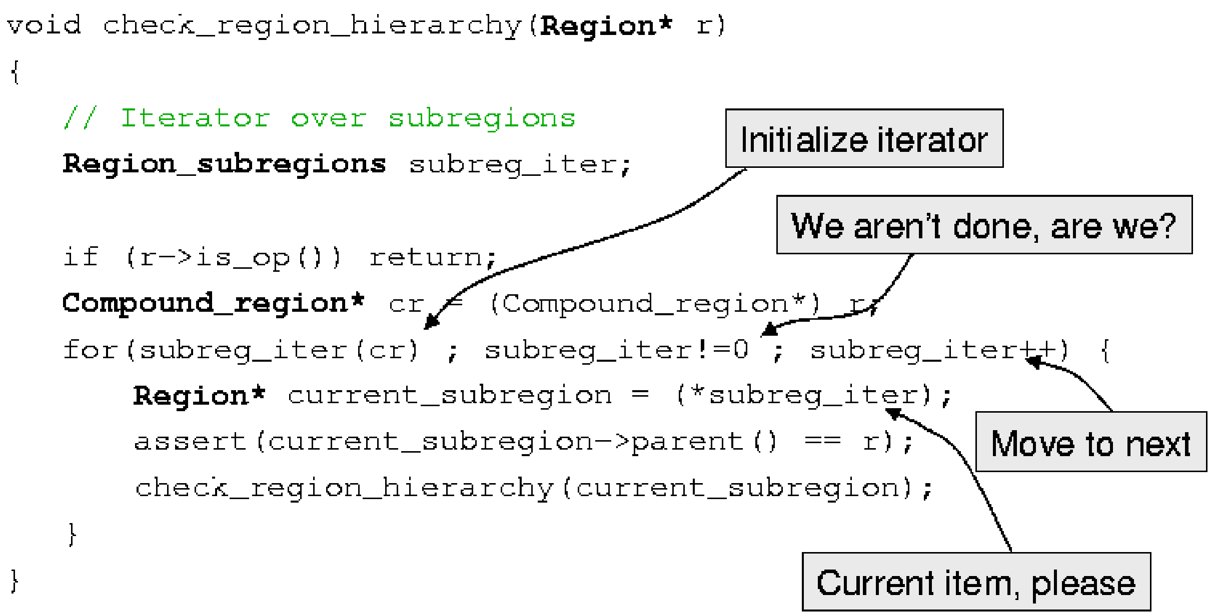
There is no explicit representation of control flow between compound regions since edges in the IR graph connect operations and not Compound regions. But since a Region is defined in terms of other operations/regions it contains , and in terms of the set of edges that enter and exit the region, control flow between compound regions can easily be deduced.

Following figure depicts regions formed from a control flow graph.



4.9 Using Iterators

Iterators are provided to iterate through regions in the graph. Following shows a sample code in C++ used to iterate through regions in a graph recursively.



```
void check_region_hierarchy(Region* r)
{
   // Iterator over subregions
   Region_subregions subreg_iter;

   if (r->is_op()) return;
   Compound_region* cr = (Compound_region*) r;
   for(subreg_iter(cr) ; subreg_iter!=0 ; subreg_iter++) {
       Region* current_subregion = (*subreg_iter);
       assert(current_subregion->parent() == r);
       check_region_hierarchy(current_subregion);
   }
}
```

The interface to Iterators is present in **trimaran/elcor/src/Graph/iterators.h**. The implementation is present in **trimaran/elcor/src/Graph/iterators.cpp**.

4.10 Attributes

The intermediate representation allows attributes (annotations) on Regions and Edges. Such attributes can be used for module specific purposes to hold module specific information.

There are several attributes that are currently used by Elcor. The interface to the attribute classes are found in the files below.

Trimaran/elcor/src/Graph/attribute_types.h
Trimaran/elcor/src/Graph/edge_attributes.h
Trimaran/elcor/src/Graph/op_attributes.h
Trimaran/elcor/src/Graph/attributes.h
Trimaran/elcor/src/Graph/mdes_attributes.h
Trimaran/elcor/src/Graph/region_attributes.h

Implementations of each of the attributes are present in the corresponding **.cpp** files.

5. Rebel

Rebel is the ASCII representation of the IR. It is human-readable. Can be parsed by a recursive descent parser. It has the same structure and elements as the data structures of IR region based and is sufficiently powerful to express program properties at various stages of compilation i.e.before / after scheduling, before / after register allocation.

5.1 The Rebel Reader / Writer

For reading Rebel, an input procedure is provided for each component type in Elcor IR. For e.g. **Region *region(IR_instream&)** parses compound regions and is implemented in **trimaran/elcor/src/Rebel/ir_region.cpp**, **Op* op(IR_instream&)** parses an operation and is implemented in **trimaran/elcor/src/Rebel/ir_op**.cpp, **Edge* edge(IR_instream&)** parses an edge and is implemented in **trimaran/elcor/src/Rebel/ir_edge.cpp**, etc.

IR_instream is a stream that provides an input and output interface to a Rebel file. The functions either return a pointer to the object, if it's of the appropriate type, or **NULL**.

The reader is implemented as a top down recursive decent parser. The main driver routine, which reads the first lexical token and dispatches the appropriate reader procedure, is **El_Input_Token ir_read(IR_instream&)**.

Similarly for writing Rebel , a procedure is provided for each component type in Elcor IR. The writer code for the procedure is implemented in the same file as its corresponding reader procedure.

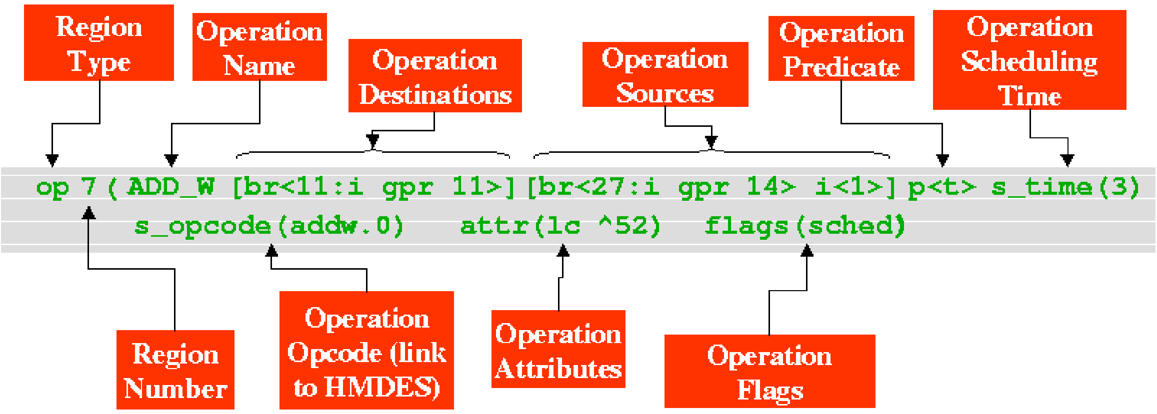
For writing out a top-level object (i.e. a procedure) along with dictionaries of all edges and attributes, there is the top-level procedure **ir_write(IR_outstream& out, Region* r)**.

The entire implementation of the Rebel reader and writer is present in Rebel directory of the elcor source.

5.2 Examples of Rebel

5.2.1 Operation

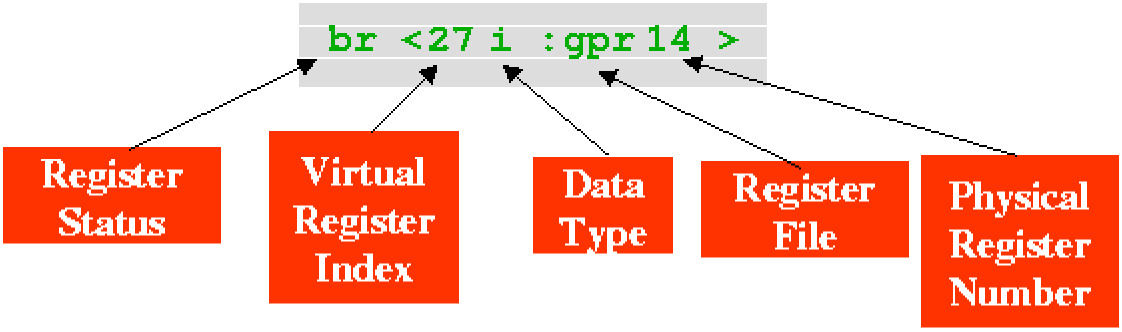
Following is an example of an Elcor operation in rebel format. It resembles the assembly language of a processor in its form except for certain additional fields like the s_time (scheduling time), s_opcode (scheduling opcode), attr (the attribute) and flags.



```
op 7 ( ADD_W [br<11:i gpr 11>] [br<27:i gpr 14> i<1>] p<t> s_time(3)
    s_opcode(addw.0)    attr(lc ^52)   flags(sched)
```

5.2.2 Operands

Following is an example of an operand representation in Rebel. The example shows a bound register (br) i.e. a physical register has been allocated to it. The 27 shows the original virtual register number i.e. its number before it was bound. 14 is the physical register number in the Register file gpr (models the HPL-PD processor).



5.2.3 Compound Region

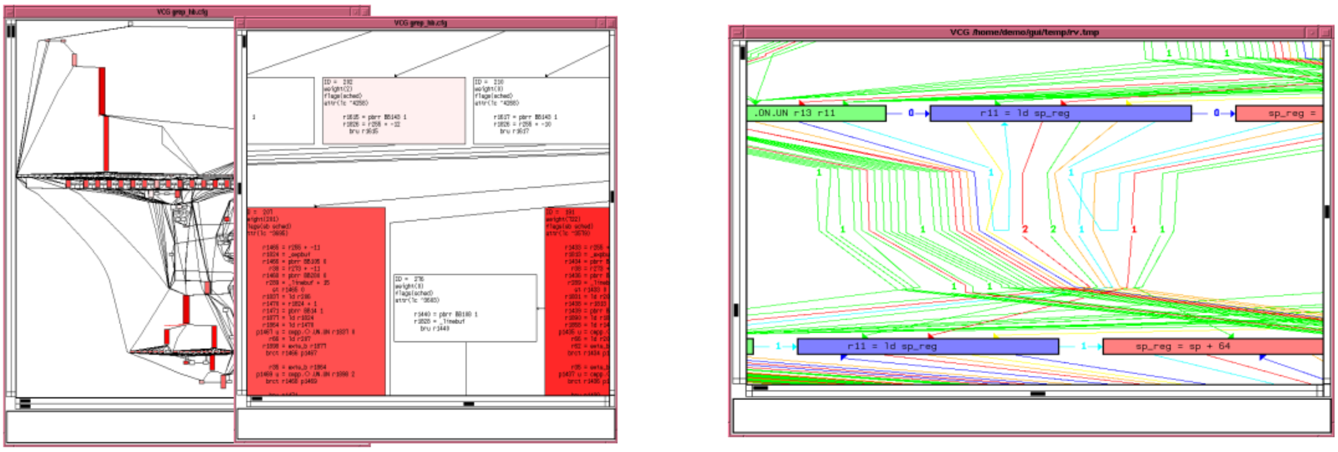
Following shows an example of a Basic block (and hence a compound region) representation in Elcor. bb 1 stands for "basic block with ID 1". The Weight keyword indicates the weight associated with a region. This is typically used to keep the frequency of visits to a region during the program's execution. It can be a guess (done at static time) or deduced from run-time / profile information generated by the simulator. Weight plays an important role in instruction scheduling and register allocation. The entry_ops field shows a list operation Ids where control flow can enter into the region. The exit_ops holds a list of operation Ids from where control flow can exit from the region. The entry_edges field has a list of edges entering the region. exit_edges field has the exit edges. The subregion construct holds all the sub-regions inside a region. Since the example below shows a basic block, subregions holds operations only.

```
bb 1 (
  weight(0)
  entry_ops(44) exit_ops(45)
  entry_edges() exit_edges(ctrl ^7)
  flags(prologue sched)  attr(lc ^32)
  subregions(
    op 44 (C_MERGE [] [] s_time(0)
      s_opcode(control_merge)
      in_edges() flags(sched))
    .
    .
    op 45 (DUMMY_BR [] [] s_time(0)
      s_opcode(dummy_branch)
      out(op-46(0)) flags(sched))
  )
)
```

5.3 The Rebel Viewer

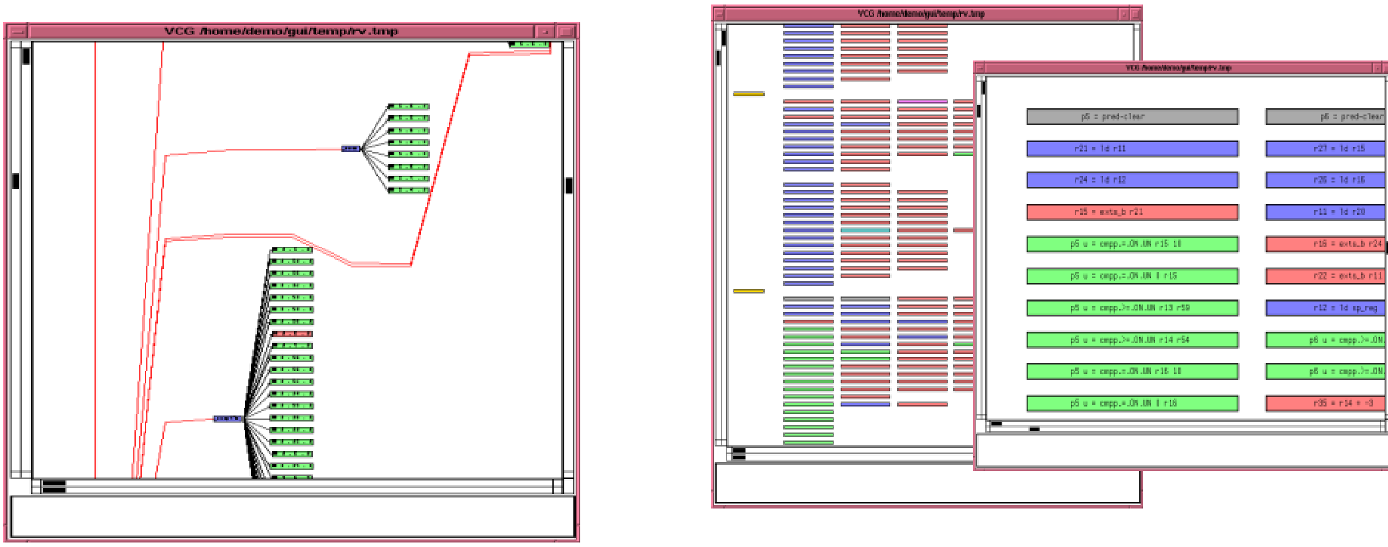
Rebel **V**iewer[1] utilizes the **vcg** utility [2] to display ELCOR intermediate representations in a graphical manner. This is intended as an aid to debug or learn about the intermediate representations of ELCOR and its part of the TRIMARAN [1] distribution. Some of the types of display that can be created are shown Figure 1. The Figurer 1(a) shows the control flow graph at the basic-block (or region) level. Data dependence graph is shown in figurer 1(b), region hierarchy is shown in Figure 1(c) while Figure 1(d) shows a cycle-by-cycle display of the operation schedule.

Figure1: Different Display Outputs Obtained Using RV



(a) CFG

(b) DDG



(c) Region hierarchy

(d) Schedule

RV converts the input rebel file into gdl (graph description language) format which is processed by **xvcg** to create the display. **VCG** (Visualization of Compiler Graphs) is a graph drawing toolkit developed by Georg Sander at Universität des Saarlandes. **VCG** can be downloaded from:
http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html.

Note : RV has been tested using xvcg version 1.3 (Revision: 3.17, Date: 1995/02/08). It is not guaranteed to work under other versions of xvcg. Bug reports may be sent to the standard TRIMARAN bug-reporting e-mail address.

5.3.1 Using RV

Table 1 Using RV

usage:	
>rv [*options*]	; options are described below
-i *input_rebel_file_name*	
-o *output_rebel_file_name*	
-t *ir_view_type*	; ir_view_type∈ { rh | ddg | cfg | stats | sched | names} ; rh = view region hierarchy ; cfg = view control flow graph ; ddg = data-dependence graph ; sched = cycle-by-cycle schedule ; stats = display execution profile as bar chart ; names = dump a list of procedure names and region-ids
-s *scope*	; *scope* ∈ {all | proc | bb } ; all = consider the entire rebel file for processing ; proc = restrict processing to a particular procedure (specified through **-f** switch) ; bb = restrict processing to specified region (specified using **-b** switch)
-f *function_name*	; name of the function to process
-b *region_id*	; number of the region to process; **-f** should be used
-d [1|0]	; to show edge/attribute-dictionaries yes(val=1)/no(val=0)
-k *val*	; *val* ∈ { 0 | 1 | 2} ; 0 : show operation information in short form (only the op-id is displayed) ; 1 : show the complete rebel format for the operation ; 2 : show pseudo-assembly version of each operation

---

[1] Note that the rebel viewer utility is **rv** (in lower case). We however refer to it as **RV** or rv in this document utilizes the

-c [1|0]	; color code according to freq/instr types
-l [1|0]	; compute liveness yes/no

NOTES:

1. When scope is *proc* (*bb*), then -f (-f,-b) options have to be specified.
2. Liveness (-l) can be used only when -t option is *cfg*.
3. Scope cannot be *bb* when type (-t) is *rh*.
4. *Stats* will work only when the ir has been instrumented with the execution profile.

5.3.2 Notes on Manipulating the Display

Some short notes on manipulating the display window. Please refer to the VCG documentation for the complete list of capabilities. Key press is followed by the action performed:

- +/- : Zoom in/out.
- m : Show entire graph (scaling appropriately).
- p : Rubber-band selection for zooming in.
- i : Information regarding a selected node will be displayed.
- q : Quit.
- right-click : Will give a menu with many options (including printing to postscript format)

5.3.3 RV Related Files in Trimaran

Table 2 RV Related Files

Sources directory	$TRIMARAN_REL_PATH/gui/rv
rv_resources.h	; resources for various display features are defined here
	; if you wish to change the color/font/layout-styles change the #defines in here
rv.*	; the main function and all related sources of ir-viewer are defined here
	; there is one function for each graph display type
el_args.*	; utility function for parsing program arguments
Makefile.rv	; makefile for generating rv

5.3.4 RV Resources

Resources are parameters which control the ``look-and-feel" of the displayed graph. Currently, these are #defines in the *rv_resources.h* file in the RV source directory. A description of the various resources and the current values is given in the table 3. Note that the user can change these values. However, that would entail recompiling the RV application.

5.3.5 Limitations/Bugs

1. All combinations of options are not legal. Some are caught but some may cause program crashes.

2. Currently, there is no way to check if the input rebel file is a valid version or not. For newer rebel versions, *rv* has to be updated as well.

Table 3 RV Resources

Colors for operation nodes:	
Integer ALU operations	Light red (17)
Compare to predicate operations	Light green (18)
Floating point operations	Light yellow (19)
Prepare to branch operations	Light magneta (20)
Switch operations	Light cyan (21)
Predicate operations	Light grey (15)
Memory (load/store) operations	Light blue (16)
Colors for edge classes in data dependence graph:	
Sequential edges (C0 edges)	Cyan (6)
Control-1 edges	Orange (29)
Data flow edges	Red (2)
Anti dependence edges	Blue (1)
Output dependence edges	Green (3)
Memory dependence edges	Yellow (4)
Default edge color	Black (31)

Note: Color values given in parentheses
Are from the color table in VCG manual.

6 References

[1] React-ILP Group. Trimaran tutorial. December 1997.

[2] George Sander. Graph layout through the VCG tool. Technical Report Sep 26-34, Technical University of Munich, September 26, 1995.