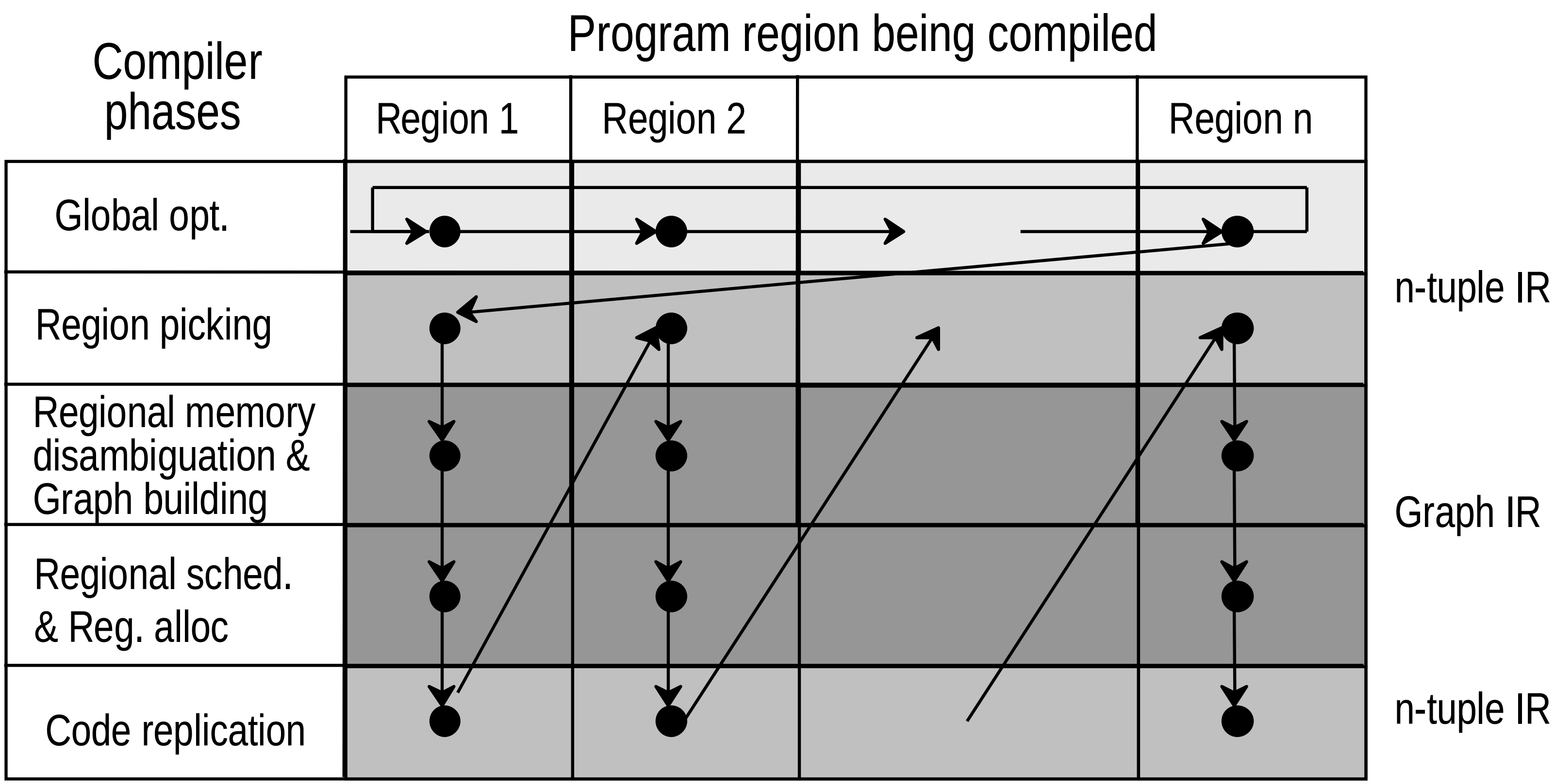
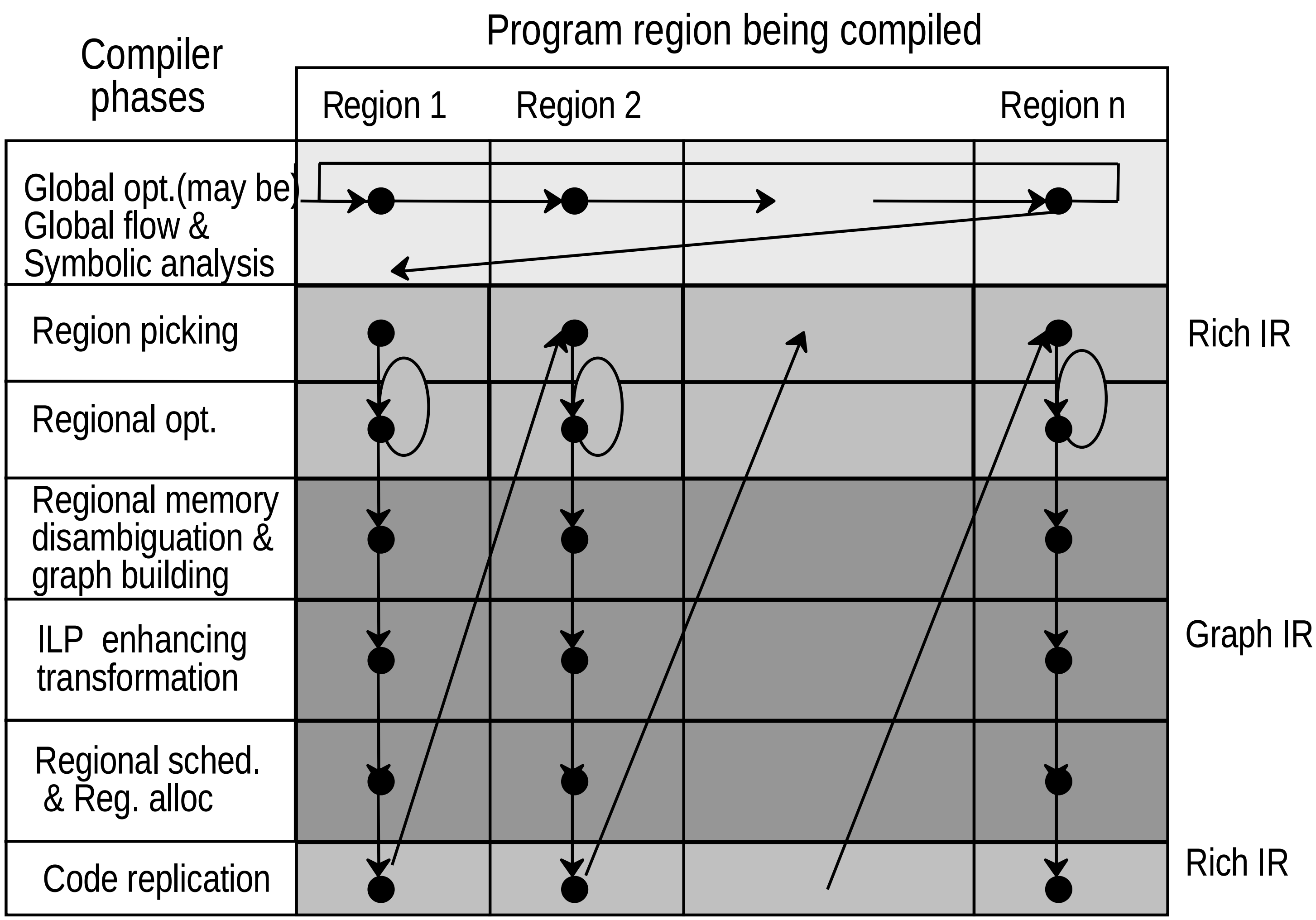
The Elcor Intermediate Representation

Traditional ILP compiler phase diagram



A region-based backend phase diagram



Factors motivating the design

- Global scheduling is key to exploiting ILP
  – We are moving towards bigger and complex regions
- Frequency-based regions have more complex structure than traditional structure-based regions (e.g., intervals, SESE)
  – Even a trace is multiple-entry multiple-exit region
- Many of the ILP enhancing techniques, e.g., height reduction, rely on estimates of height and resource usage (abstract scheduling)
  – Such estimates may be helpful even in earlier phases
- Analysis like memory disambiguation are expensive
  – Need to represent and maintain their results accurately

Factors (cont.)

- Flexibility in phase ordering
  - because we don't fully understand the right phase order
- Flexibility and ability to grow
  - In many cases, we don't fully understand the requirements
  - IR highly optimized for a specific purpose may not be the right one
  - Put general mechanism to support various policies
  - Well defined interfaces to modules and encapsulation
- Uniformity
  - Easy to build software, modify and grow

IR Features

- Multi-state IR
- Provides mechanism for representing
  - Traditional control flow graph
  - Control dependences
  - Data dependences for both registers and memory in various forms
  - Various forms of register usage – single assignment, multiple assignments
  - Expanded virtual registers (EVRs)
  - Predicated execution
- Data section
  - Global symbols, arrays, etc.
- Registers carry values, edges represent dependences
- A uniform, edge-based representation of control flow and data dependences
- Supports threading of data dependences ala dependence flow graphs
- Hierarchical non-overlapping region structure (a tree)

Internal vs. Textual Representation

- Each component of the graph data structure is a C++ object
  - All modules of the Elcor use this IR
  - Optimization are simply IR-to-IR transformations
- There is an ASCII intermediate representation, called Rebel.
  - Phases of Elcor may communicate using 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.

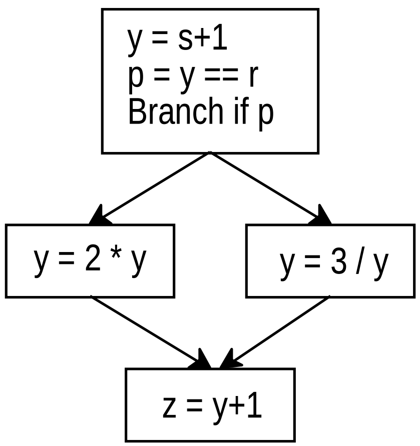
Program Representation

A program unit is represented by

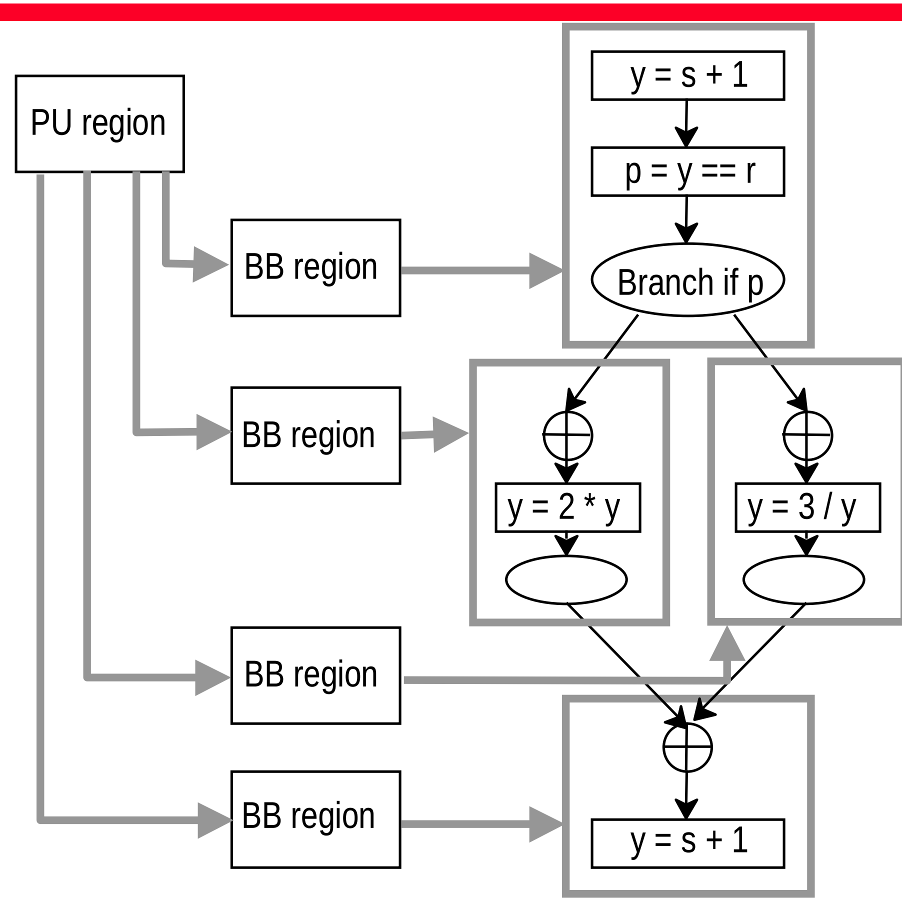
1) A graph of operations connected by edges
   - **Control flow is represented explicitly and at the operation level**

2) A region structure over the operation graph (a tree)
   - **The root of the tree is the program unit, e.g. a procedure**
   - **The leaf nodes of the tree are operations**

An example: Representing traditional CFG



Traditional CFG

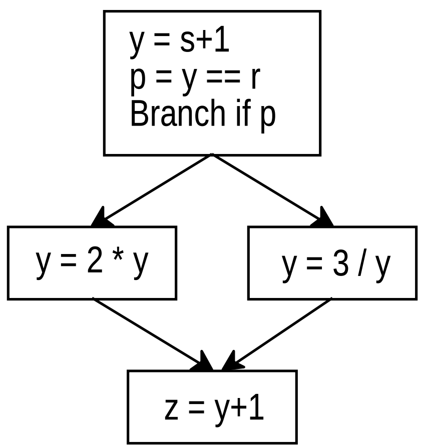


Region Structure

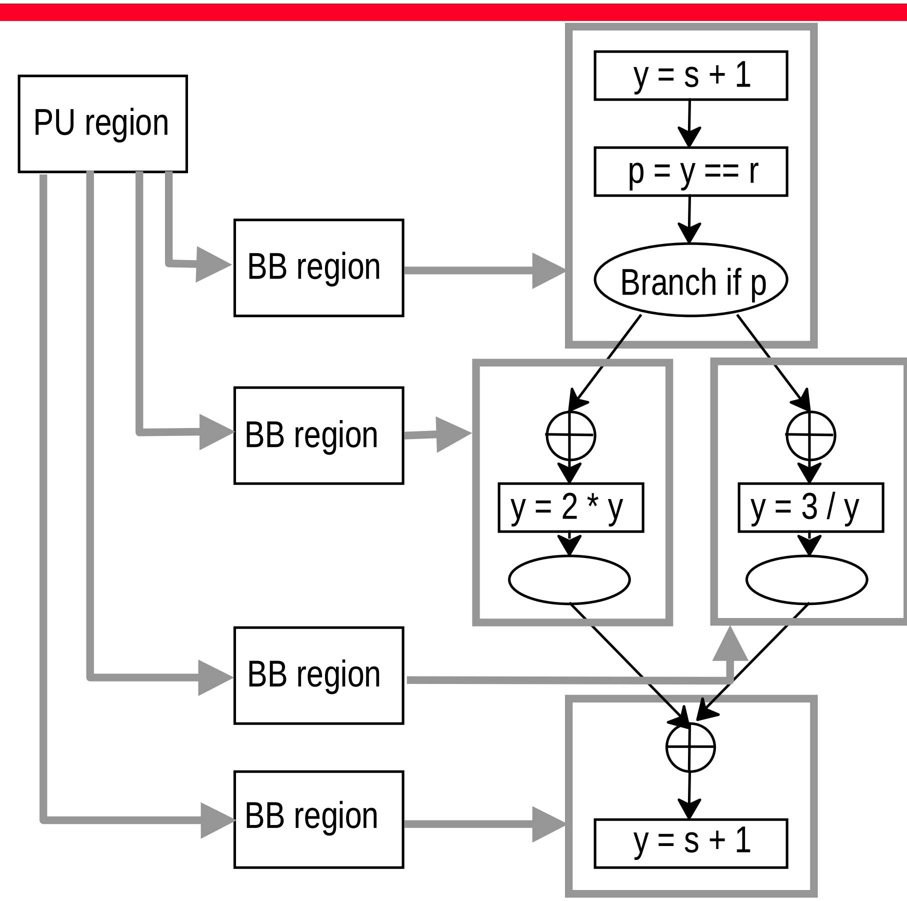
Operation Graph with control flow edges

Representation in Elcor IR

An example: Representing traditional CFG



Traditional CFG

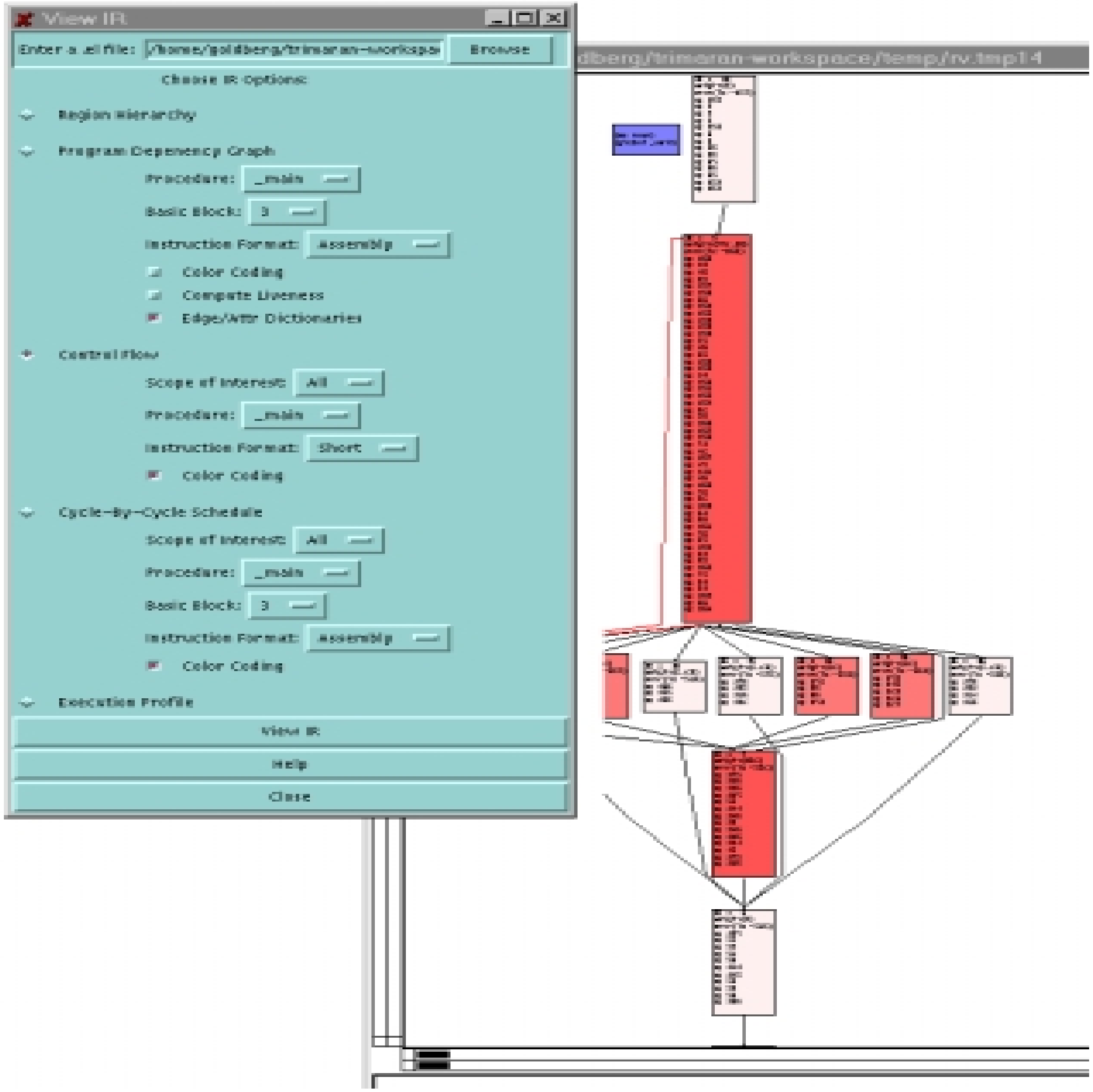


Region Structure

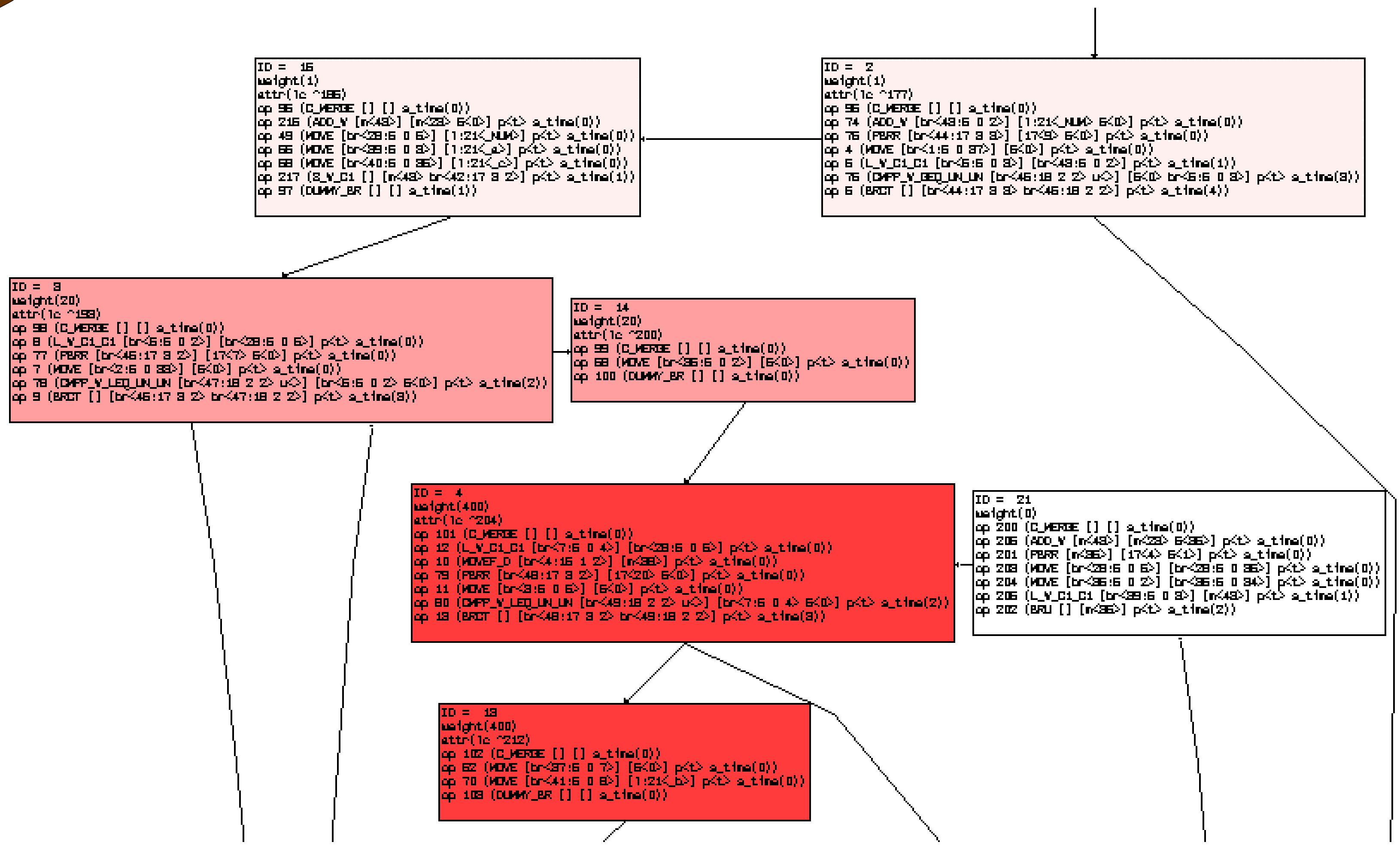
Operation Graph with control flow edges

Representation in Elcor IR

Control Flow Viewer

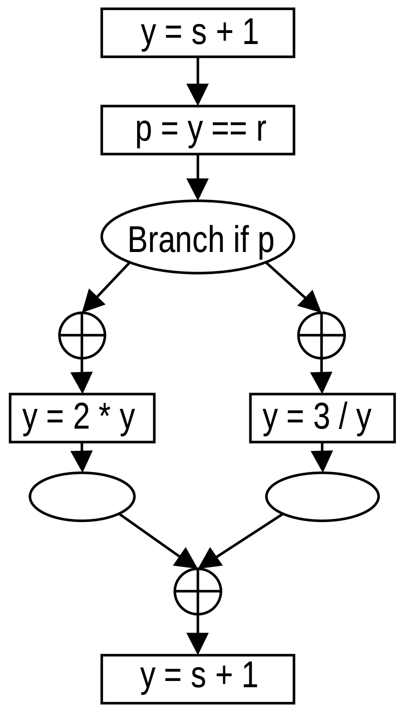


Control Flow Viewer

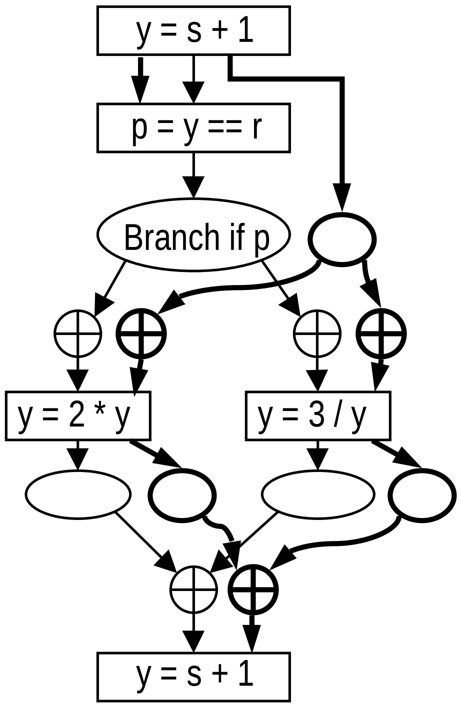


Uniform representation of dependences

- Operation graph can represent both control flow and data dependences

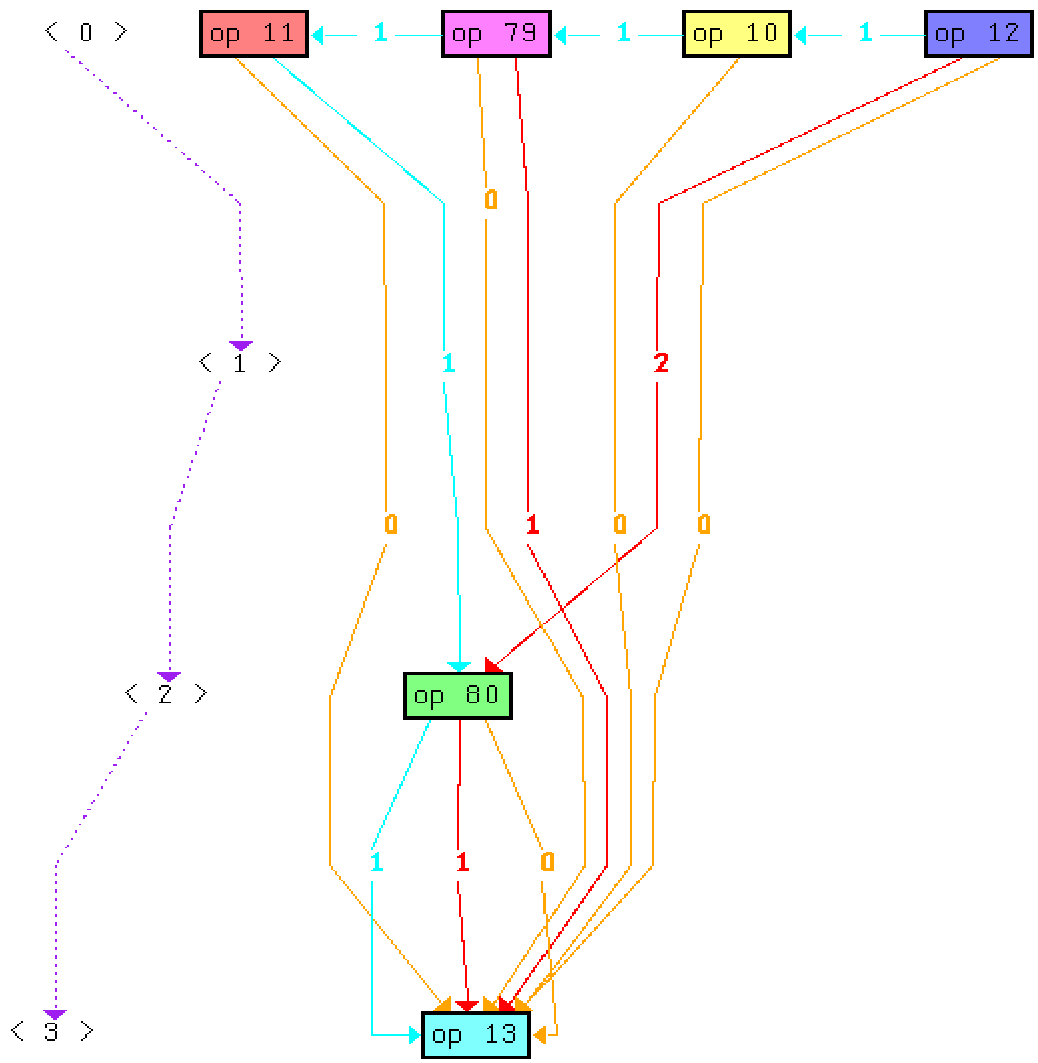


Operation graph with control flow edges



Operation graph with control flow and threaded flow dependence edges for y

Data Dependency Viewer



Operation graph elements

- Op(eration) class
- Operand class
- Edge class

Op class

- Represents an operation
  - Machine operation
  - Compiler operations (e.g.,CONTROL_MERGE, PRED_CLEAR)
- Has source and destination operands including guarding predicate (their number is determined by MDES)

```
dest1, ..., destm = opcode(src1, ..., srcn) if p
```

- May have implicit sources and destinations
  - e.g., parameter passing registers for BRL
- Memory dependence "sources" and "destinations"
  - Memory dependences are encoded as "def" and "use" of special variables

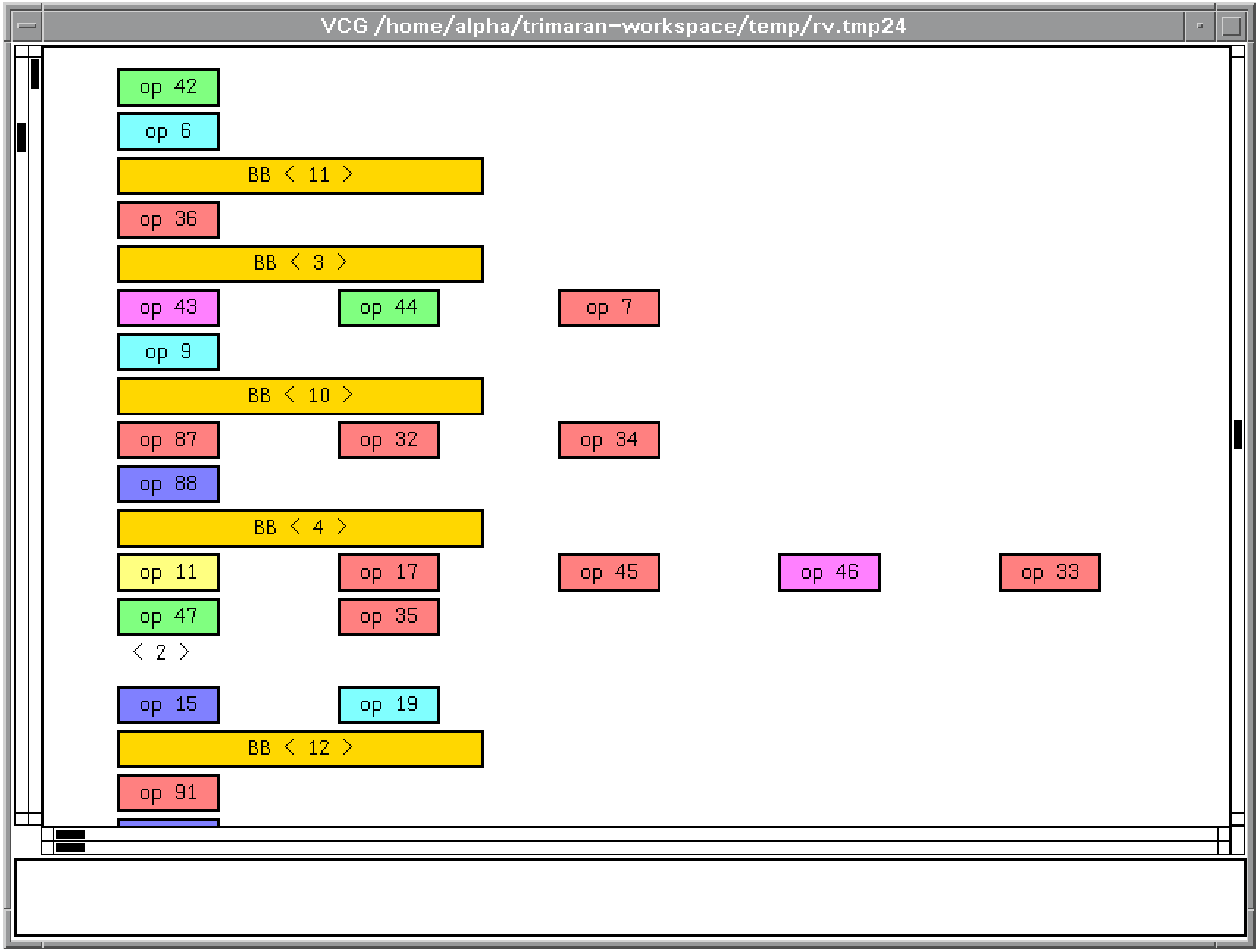
```
<$a> r3 = load (r4)
store(r1, r2) <$a, $b, ...>
```

  - Simplifies dependence graph construction
- Set of input edges and set of output edges
- Schedule time, latency queries for sources/destinations

Operand Class

- Registers
  - Unassigned or assigned
  - Can be unbound or bound to static or rotating register files.
- Macro registers
  - Registers reserved by compiler or runtime system. Parameter passing registers, stack pointer, frame pointer, loop counter, epilogue stage counter etc.
- Memory registers
  - Used to encode memory dependence edges
- Register names
  - Used as operands to REMAP operation for EVR's
- Local branch targets
  - Basic block ID's that appear as branch targets
- Literals
  - Integer, float, double, predicate, string, label
- Undefined

Instruction Schedule Viewer



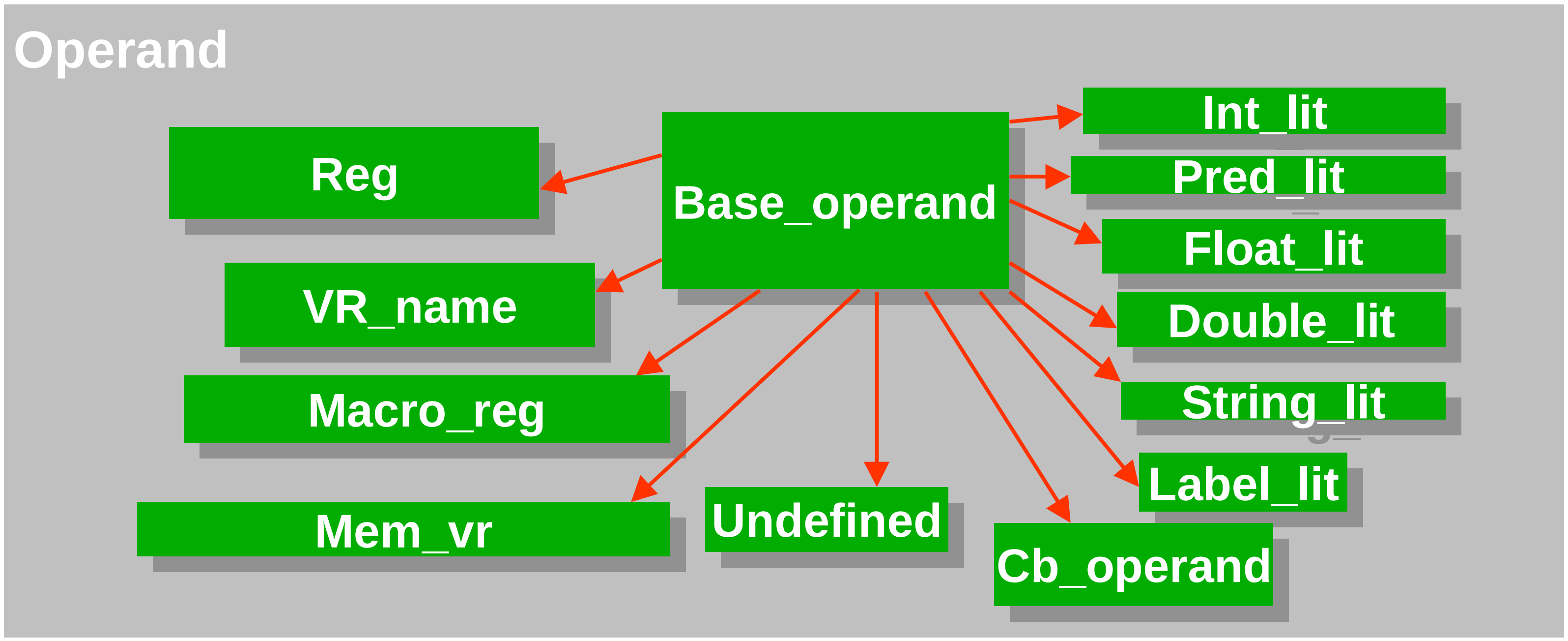
EVR’s

- EVRs allow multiple values from a sequence of assignments to be live at the same time
- An EVR is a linearly ordered set of VRs
  - Elements are referenced using the notation t[0], t[1], etc.
  - A special remap operation to "shift" reference coordinates

```
t = 0;    // t means t[0]

remap(t); // Previous value
          // of t is now t[1]

t = 1;

remap (t);

t = t[1] + t[2] // t = 0 + 1
```

- EVRs allow
  - Accurate representation of value flow across zero or more iterations of a loop
  - Representation of results of analysis and transformation without unrolling or unnecessary copies
    E.g., The use of the value loaded in previous previous iteration as t[2]
  - Representation in dynamic single assignment form to eliminate inter-iteration anti- and output dependences
- Use of EVRs in IR doesn’t imply use of rotating registers in hardware
  - Code can be unrolled at a later stage if rotating registers are not supported

Operand Class Hierarchy



Operand class is a wrapper for all operand types.

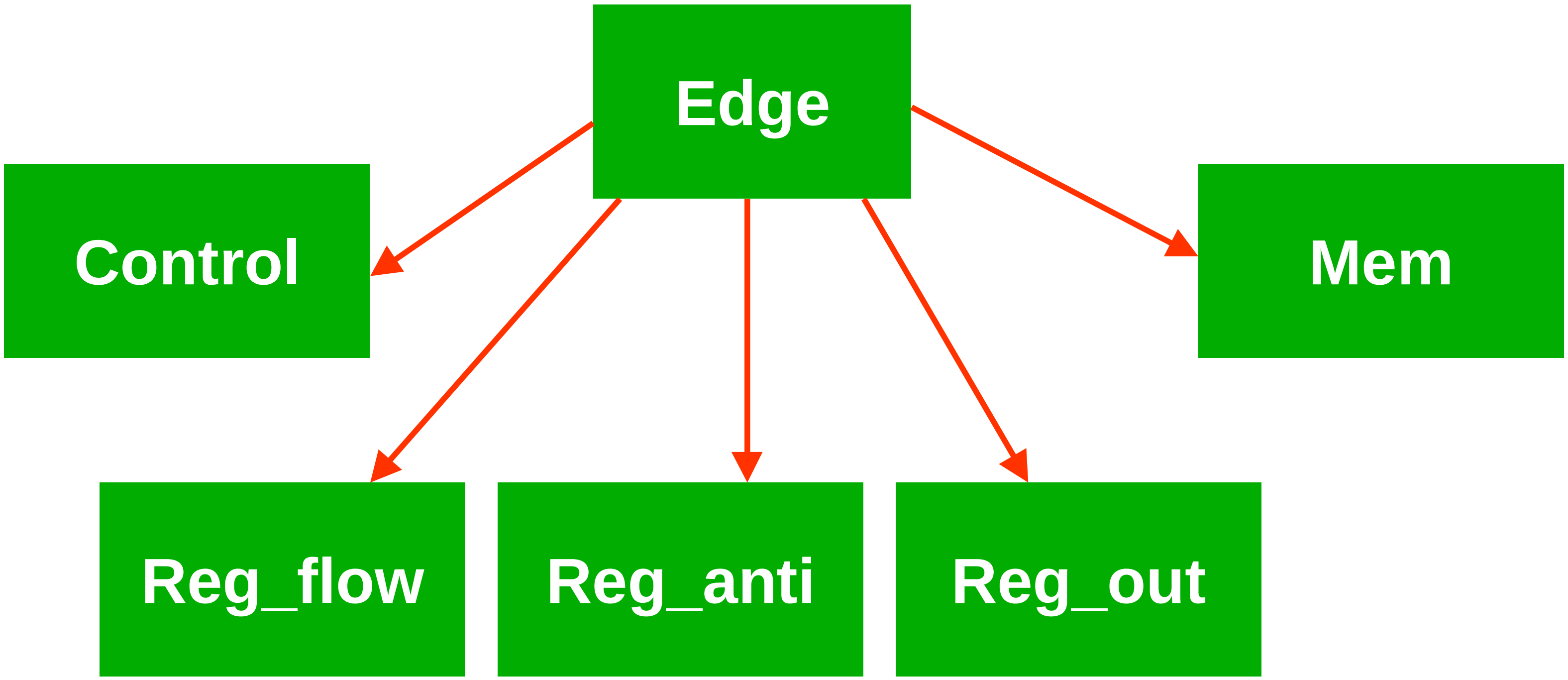
- Provides Boolean methods for class type testing
- Provides access methods to class specific fields
- Provides comparison operators
- Manages symbol table

Edge Class

- Edges Represents dependence constraints between operations
- Edges do not represent value-flow like data flow graphs
- Edge types:
  - Control (sequential control flow, control dependence)
  - Flow, anti and output dependences on registers
  - Flow, anti and output memory dependences classified as "certain" or "maybe"
- An edge has pointers to source and destination ops
- An edge also contains more detailed reason for dependence
  - Represented in terms of "Port" for source and destination operands
    - **e.g., register flow edge from DEST1 of op1 to SRC2 of op2**
- Latency setting and querying functions

Edge Class Hierarchy

- The hierarchy is based on how latency for an edge is computed.

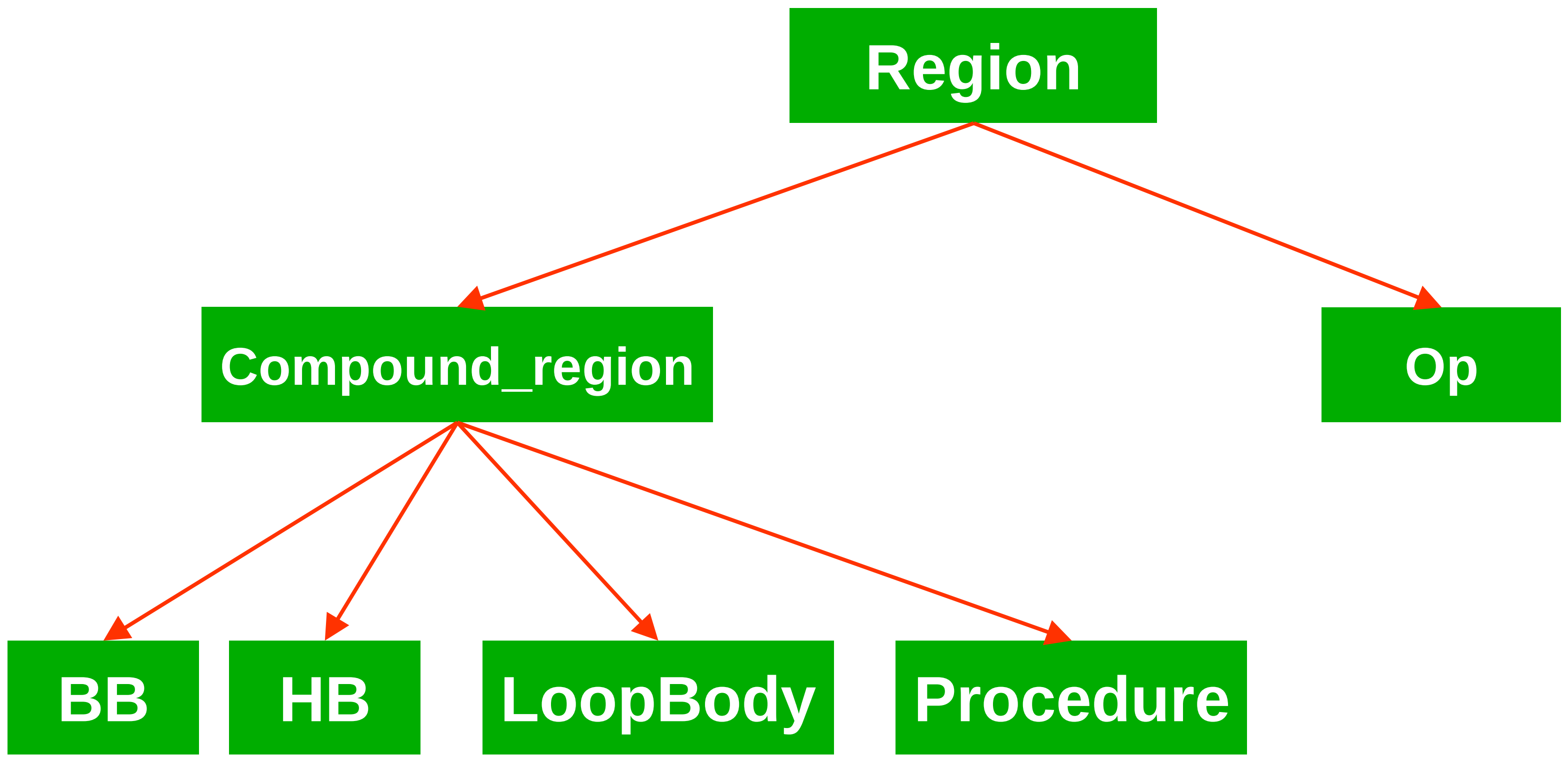


Region Structure

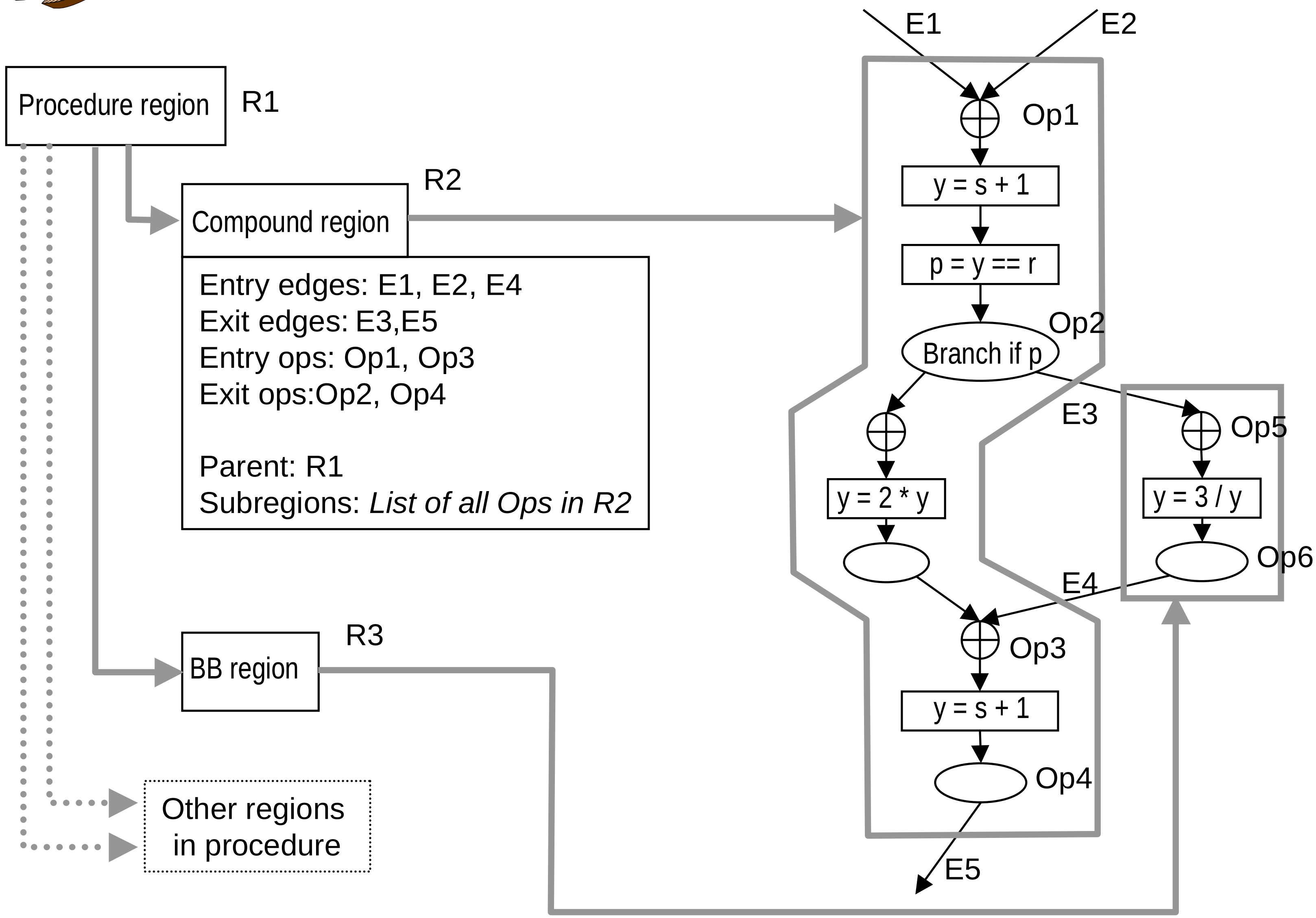
- Region structure over a program is a tree structure
  - Leaves of the tree are Operations
- A region is defined by
  - Operations contained in the region
  - Set of control flow edges that enter or exit the region
  - Set of entry and exit operations (mostly redundant)
- All entry operations are CONTROL_MERGE operation
- All exit operations are branch operations
  - There is a DUMMY_BRANCH operation if region exit is fallthrough
- Regions are used to set scope for analysis, optimization, scheduling, register allocation etc.

Region Class Hierarchy

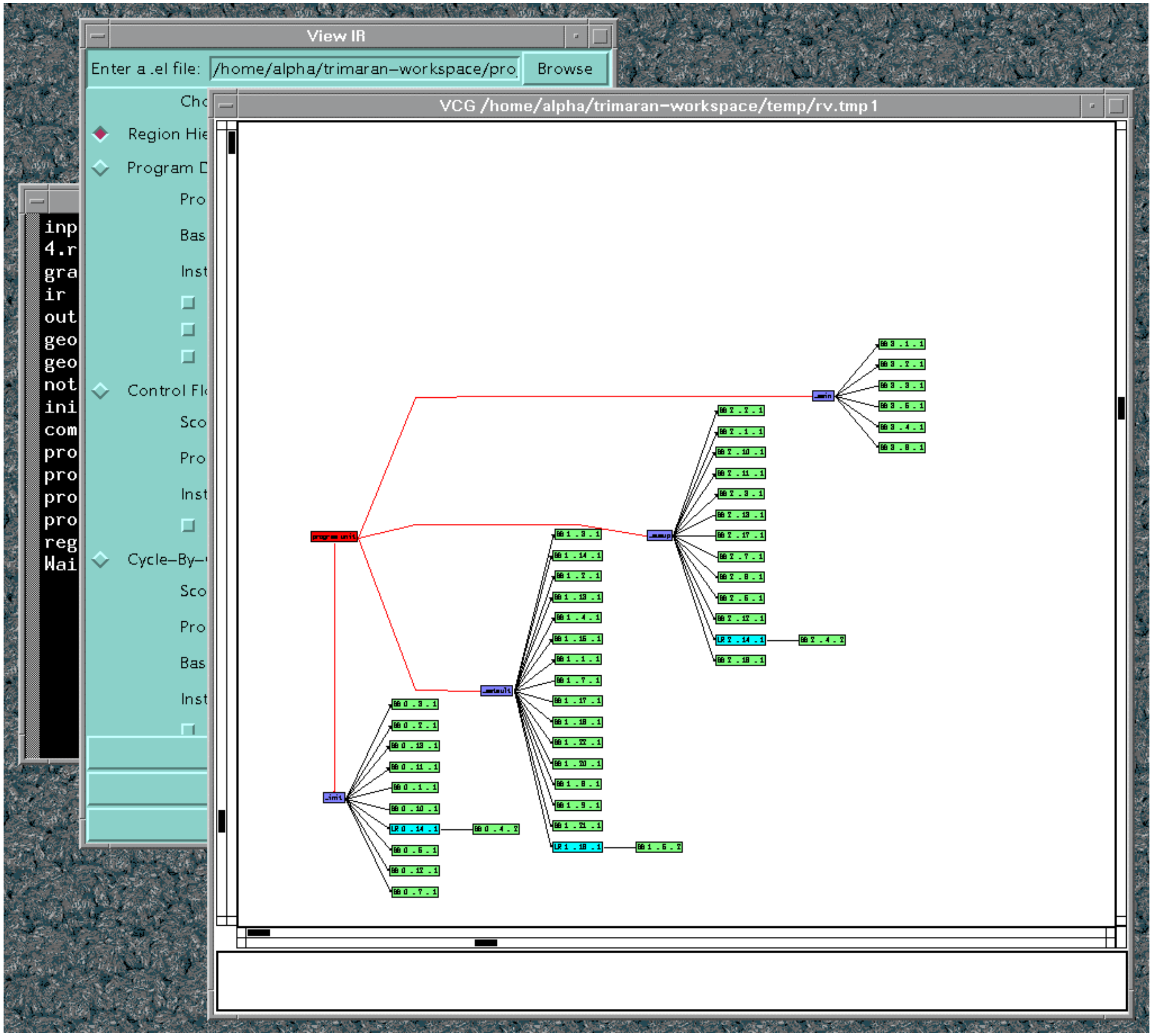
- Region class is an abstract base class.
- Compound regions can contain other regions in the region tree.



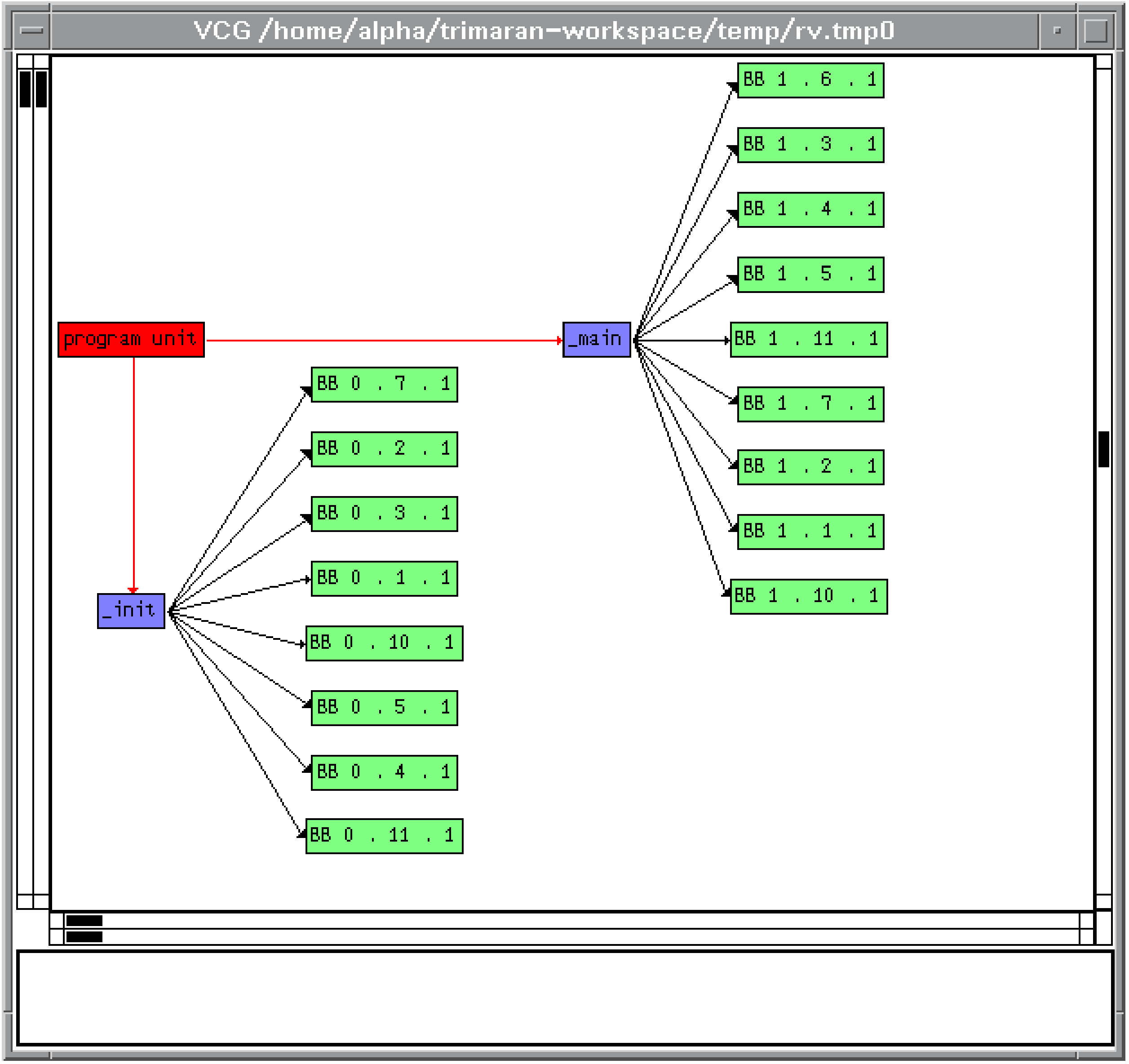
Region Representation



Region Hierarchy Viewer



Region Hierarchy Viewer



Control Flow Between Compound Regions

- There is no explicit representation of control flow between compound regions
  - In a region hierarchy, successor or predecessor of a compound region is not unique.
- If we consider a “cut” of region hierarchy tree, we can construct a control flow graph of the regions in the cut.

  E.g., if in a region hierarchy every operation is subregion of a basicblock, we can find a cut containing only basic blocks therefore a basic block control flow graph can be constructed.

  Every exit edge of a basic block is an entry edge of some basic block

Practical Considerations

- Operation graph has CONTROL_MERGE and DUMMY_BRANCH operations inserted in it to construct a basic-block only covering of the operation graph at all times
- Procedures don’t have entry/exit control flow edges.
- Some important region hierarchies are
  - Single entry/multiple exit compound region with a tiling of basic-blocks and hyperblocks (Region based analysis).
  - Single entry/multiple exit compound region with a tiling of basic-blocks (Control flow transformations).
  - Hyperblock with only operations in it (Acyclic scheduling).
  - LoopBody with only operations in it (Modulo scheduling)

Using the IR iterators

Elcor provides a collection of iterators to walk data structures

```
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);
   }
}
```

Initialize iterator

We aren’t done, are we?

Move to next

Current item, please

Attributes

- The intermediate representation allows annotations on Regions and Edges
  - Used for module specific purposes
  - Used when the information is sparse
- There are two kinds of attributes
  - Heavy weight
    - **Type safe**
    - **Can be represented in ASCII form of IR (can be printed and parsed in)**
    - **If the object it is attached to is deleted the attributes are deleted**
  - Light weight
    - **Stored and retrieved using string keys**
    - **Not type safe**

Rebel

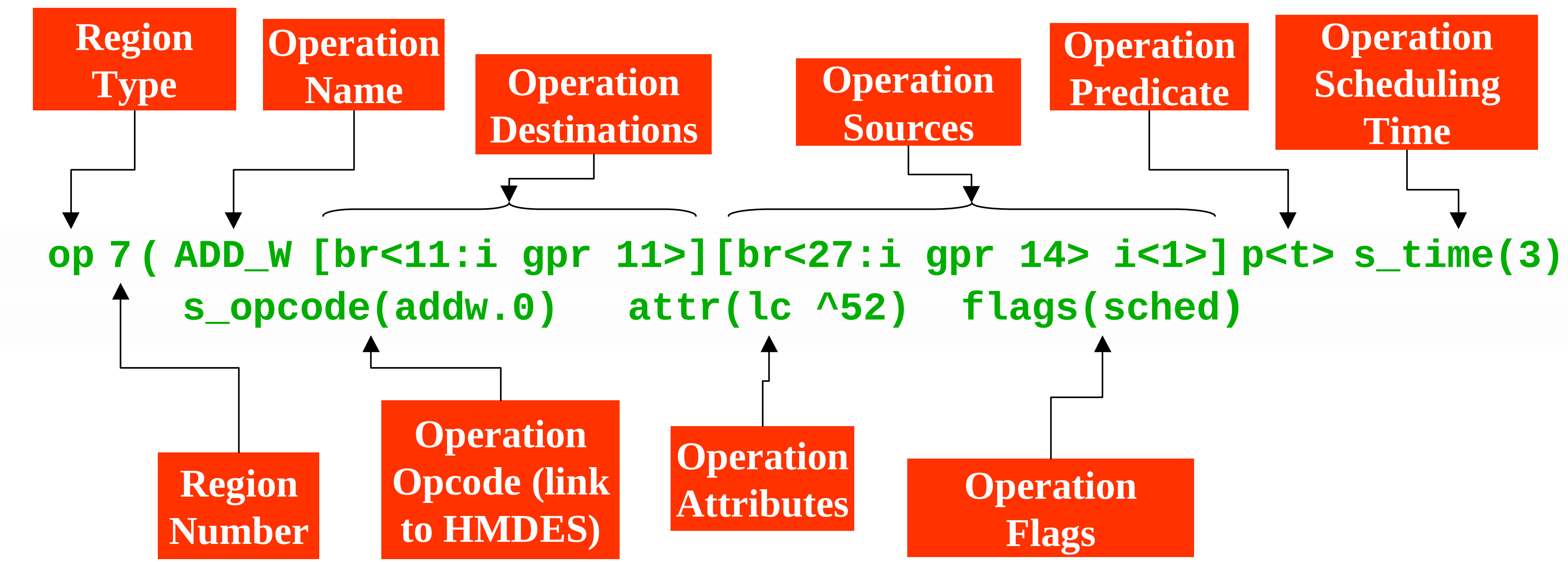
- Rebel is the ASCII representation of the IR
- It is human-readable
  - Can be parsed by a recursive descent parser
- It has the same structure and elements as the data structures of IR
  - region based
  - sufficiently powerful to express program properties at various stages of compilation
    - **before/after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

The Rebel Reader/Writer

- For reading Rebel, an input procedure is provided for each component type:
  - Region *region(IR_instream&);
  - Op *op(IR_instream&);
  - BB*bb(IR_instream&)
  - Edge *edge(IR_instream&);
  - . . .
- These either return a pointer to the object, if it's of the appropriate type, or NULL.
- The main driver routine, which reads the first lexical token and dispatches the appropriate reader procedure, is
  - El_Input_Token ir_read(IR_instream& in)
- For printing 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)

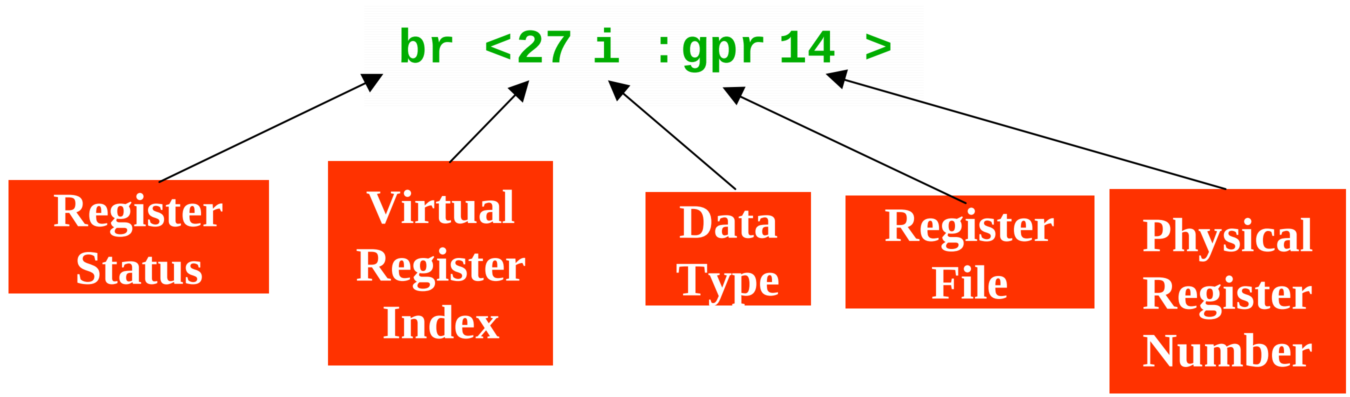
Operation in Rebel

Here is how an operation region looks in Rebel



Operands in Rebel

- A register operand (in an Op region) looks like:



- The register status is r if it's a virtual register or br if it is an allocated register.

Compound Region in Rebel

- Here is a Basic Block region.
  - The other compound regions are similar.

```
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))
  )
)
```

Summary

- Elcor Intermediate Representation is
  - Graph based with explicit representation of dependence and control flow
  - Region based

- There are two forms of the intermediate representation that a researcher can use.
  - Internal representation
    - **C++ object based**
    - **Used by all Elcor modules**
  - Textual representation (Rebel)
    - **Complete program representation**
    - **Easily parsed, readable**