Polymorphic C++ Debugging for System Design

Frederic Doucet, Rajesh Gupta

Technical Report #00-06

Information and Computer Science
University of California
Irvine, CA 92697-3245
http://www.ics.uci.edu/∼iesag

Febuary 11, 2000

Contents

Abstract

SystemC and CynApps libraries makes the use of the C++ language for system design possible. However, designers use it the same way they use HDLs: they model the hardware part of the design at the register transfer level, or wire level. In particular, extensive use of object oriented mechanisms (as in software) is restricted. Usually, this is because these features are difficult to conceptualize, simulate and to synthesize. This paper addresses the use in system design of one of these mechanism for hardware design, polymorphism, and the associated debugging problems. The C++ implementation of polymorphism relies on pointer manipulation. We describe how pointers have been identified as useful, the capability of their usage, and describe how to solve some of the associated debugging problems.

Introduction

Object oriented methodology for hardware design have been proposed in [4, 10] and [11] to increase the level of abstraction to address the growing complexity of implementations. Class libraries as an extension to the C++ language have been proposed recently to address the increasing complexity of system design [7, 1]. Such techniques and languages were applied in successful chip design in [8, 9]. Recent interest in these techniques is due to the fact that modeling hardware in a software centric environment is key to enabling effective system codesign.

With an object oriented language, the system designer can use powerful decomposition techniques for both software and hardware portions of the design. One of these technique is polymorphism, the specialization of operations through object interfaces.

The ways polymorphism is implemented in C++ is by using pointers. There have been an implicit consensus not to use pointers among language and tool designers for digital hardware because of the difficulty to synthesize them. But, work done in [6, 5] shows new pointer synthesis techniques and opens the door to their usage in hardware modeling. The problem we address in this paper is that in presence of pointers, HDL debugging with traditional waveform viewers is not possible due to static binding of variables to their locations.

We have modeled a DLX processor pipeline in the C++ language using the SystemC class libraries and used polymorphism for the instruction functional decomposition instead of modeling at the register transfer level. We show how to trace, view and interpret the wave output of pointers.

In the section 1, we describe the polymorphism mechanism. This is followed by a description of our approach and implementation.

1 On the Usage of Polymorphism

1.1 What is it?

Polymorphism is a key concept in object-oriented system. It is based on the definition of an object interface. This interface has virtual methods, that can be implemented by specialized (derived) objects. It is then possible to substitute the base object by a specialized object which has the same interface and have the same method invocations. This specialized object, can implement or re-implement the methods from the interface for a specialized behavior. This technique is common in software design patterns [2], though not as prevalent among hardware designers.

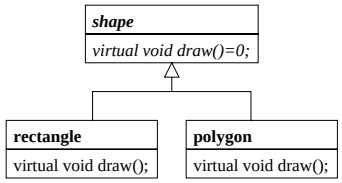

Figure 1: Typical Polymorphism Usage

Let us take a look at a polymorphic design. Several design patterns use polymorphism to call different functionality through a standard interface. A typical usage of this is shown at Figure 1: a class structure for manipulating shapes in a program. Each drawable shape inherits from the attributes of the *shape* base class and implements the *draw()* virtual method for the desired shape. The program manages shapes using pointers to the base class, and calls the interface methods defined at the base class level. These calls are forwarded down the hierarchy to the virtual method with the implementation of the specified shape. For example, it will draw a rectangle for a call forwarded to the *draw()* method of the *rectangle* class, and a polygon for a call forwarded to the *draw()* method of the *polygon* class.

1.2 Why is it useful?

From the software perpective the designer does not have to commit to an implementation until run-time where the objects are dynamically binded through their interfaces. Also, this functional decomposition enable higher level modeling than RTL design. Wire and register details are implicit through the interfaces and the method calls.

1.3 Problems for Hardware

The problem for the usage of polymorphism in hardware description is the debugging of dynamically bound objects through their interfaces. This mechanism is done through the pointer to the base class. During the simulation, the designer will want to view the state of the pointers, and of the addressed data in the time frame. Unfortunately, a pointer and its binding is not traceable by the current SystemC static waveform tracing mechanisms. In the next section, we will take a look at the problems raised by pointer tracing in this context, and describe our approach to address them.

2 Waveform Tracing

One of the mechanism for wave tracing in SystemC is through the Wave Intermediate Format, or WIF. The address of a variable to trace is registered, and watched for a value change. If such an event occurs, the tracer will write the new value and the temporal information of the event in the trace file.

In the context of polymorphic usage, the tracing problem is basically a data location one. The problem with pointers is that the location of the value to watch changes, as objects are being

reallocated and passed around in the program.

The tracer is aware of the changes of the location of the value, but not the changes of state of the value. The pointer is the intermediate between the attribute being traced and the tracer in the sense that it adds one more addressing level. Let us illustrate the problem by looking at Figure 2.

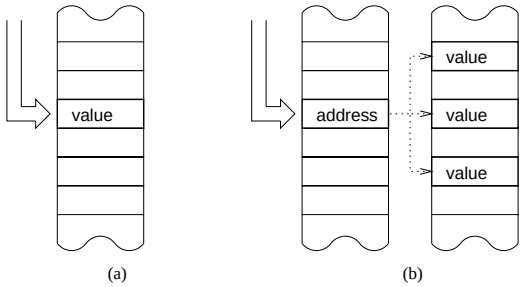

Figure 2: Tracing Example: (a) Attribute (b) Pointer

Figure 2(a) shows the tracing for a normal aggregated attribute. The location of the variable is constant throughout the simulation. The tracer registers its address in the set of variable to trace, and watch for a change of value at this location. If there is a change, the trace file will be updated with the new value and the time of the event. Figure 2(b) illustrates the case of pointer tracing. The tracer registers the address of the pointer and watches for a change at this location. The tracing algorithm will watch the address of the value, and update the tracing file with changes on the address. It will not trace the state of the value, but just the state of its address in the memory.

Now that we have identified the problem, let us look at a tracing algorithm that will enable us to debug our design. Figure 3 presents our algorithm for pointer tracing. The first step is to register the address of the pointer in the set of variable to trace. The value of the pointer will be followed, and if the value is not null, the state of the pointed data will be displayed. This displayed value is saved as the last state of the pointed-to data. The tracing then watches the value of the pointer, and the value of the data to track changes. If there is a change on either of them, the trace file will be updated using this approach.

We have extended the tracing capabilities of the SystemC WIF mechanism to be able to trace pointers in a polymorphic usage pattern. We have specialized the WIF tracing interfaces to add the pointer tracing mechanism.

1: add $pointer$ to $variable\_list$ to watch
2: $old\_pointer = pointer$
3: **while** $simulationruns$
4: **if** $pointer != NULL$
5: **if** $pointer != old\_pointer \parallel *pointer != old\_data$
6: update trace file
7: $old\_data = *pointer$
8: **end while**

Figure 3: Pointer Tracing Algorithm

3 Experimental Results

In this section, we present the design of a DLX processor pipeline [3] using polymorphic decomposition. We also explain the implementation of the pointer tracing applied to polymorphic design debugging, and illustrate the results.

3.1 DLX Pipeline

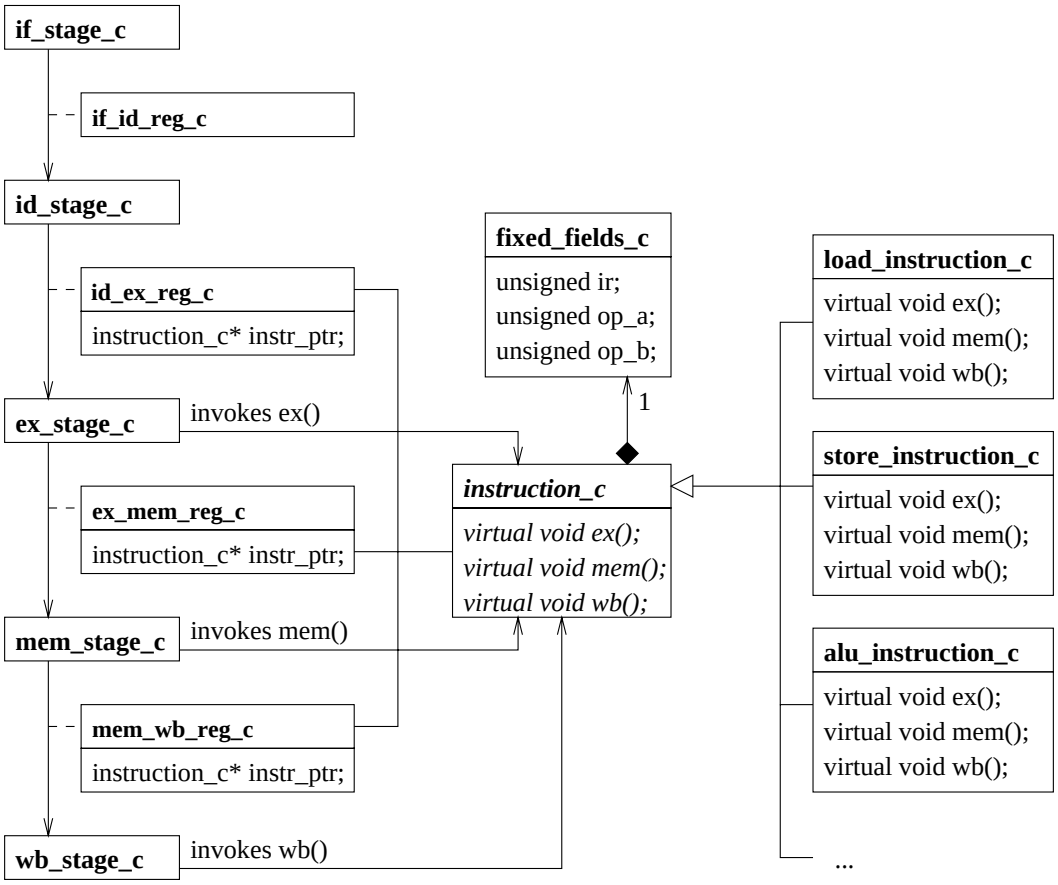

Figure 4: Class Diagram for the DLX Pipeline

Figure 4 presents the class diagram for the DLX processor pipeline. We see the five pipeline stages: the fetch, decode, execute, memory access and write back register are respectively the *if_stage_c*, *id_stage_c*, *ex_stage_c*, *mem_stage_c* and *wb_stage_c* classes.

The registers between the stages are implemented through characterized association classes, and these classes refer the instruction been propagated down the pipeline; starting from the decode stage, where it is instantiated. The abstract base class *instruction_c* defines the interface for all instructions. It aggregates the instruction fixed fields through composition. These fixed fields contains the instruction register and the operands. Each specialization of the instruction interface implements the *ex()*, *mem()* and *wb()* methods, which are called by the execute, memory and write

back stages respectively. The polymorphism mechanism eneble the manipulation of instruction by their *instruction_c* base class, and the calls are made through their interface to the right implementation. This pipeline was implemented using SystemC version 0.8.

3.2 The Pointer Tracing

It is common to debug design modeled using variants of C++ through a console mode output, as it is routinely done in software systems. Hardware modeling- specifically event modeling- requires a waveform display to enable effective debugging of the system design for correct funtionality and timing behavior. This is done through the WIF interface.

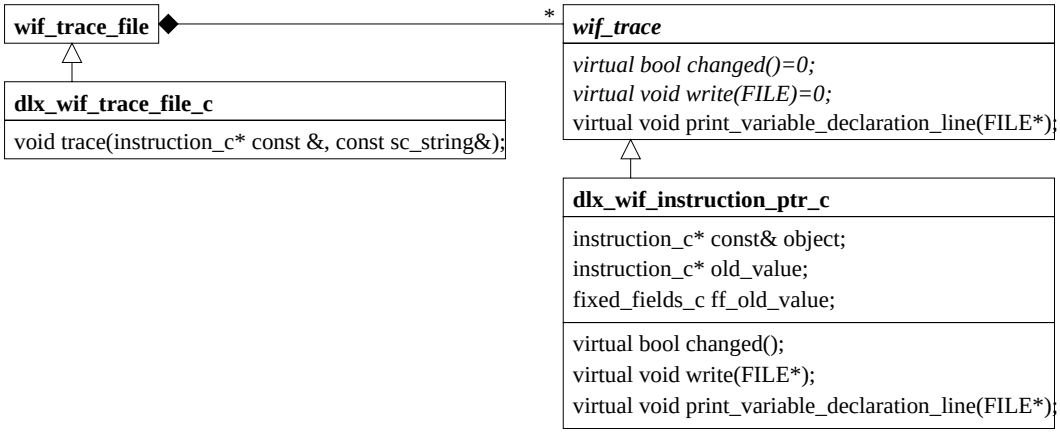

Figure 5: Class Diagram for the Tracing Mechanism

Figure 5 presents the class diagram for our pointer tracing mechanism. We have done the implementation of the tracing by specializing the *wif_trace* and the *wif_trace_file* classes in SystemC. The first specialization is to encapsulate the pointer as an object to trace, and the second one is to contain the first one in the set of objects to trace. The tracing algorithm is implemented in the object of class *dlx_wif_trace_file_c*, and invokes the methods of objects of class *dlx_wif_instruction_ptr_c* to watch the status of the traced object. Let us take a look at a sample program to illustrate the pointer tracing. The follwing code, is a small loop counter program:

```
1: andi 0,r2,r2
2: load 0,r1
3: add  r1,r2,r3
4: subi r1,1,r1
5: bnez r1,2
6: halt
```

Figure 6 shows a part of the wave output for the test program. On the figure, we can see the program counter (*PC*), the first and the second registers of the register bank (*R1*, *R2*) and the inter stages registers (*IF_ID_REG*, *ID_EX_REG*, *EX_MEM_REG*).

The instruction register (*IF_ID.ir*) of the fetch-decode register at cycle 100, is propagated to the instruction register field of the instruction pointer (*instr_ptr.ir*) of the decode-execute register

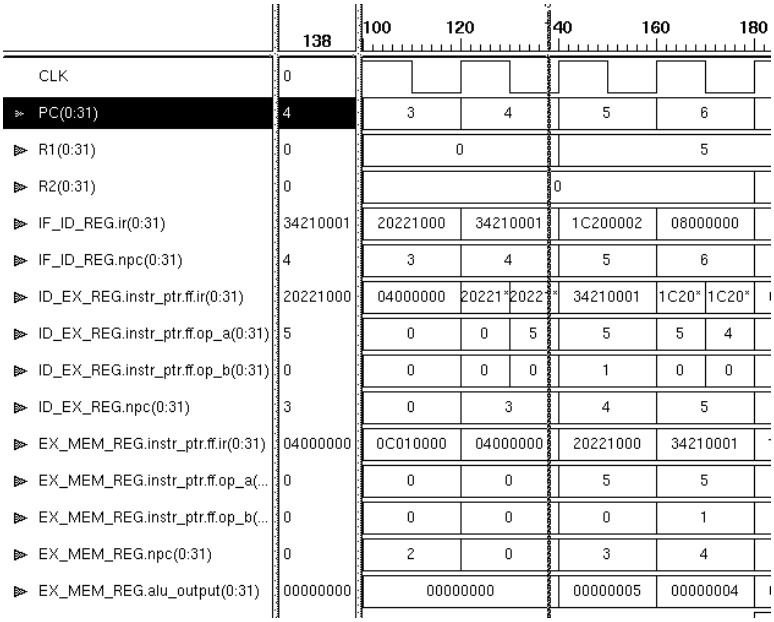

Figure 6: Wave Diagram Output Sample for the Program Example

at cycle 120, and to the *instr_ptr.ir* field of the execute-memory register at cycle 140. Notice that the operand values (*op_a* and *op_b*) are been updated in the same procedure. We can also notice, at cycle 130, that the first operand (*op_a*) of the instruction in the decode-execute register is been forwarded from a deeper stage. The value of the tracing is been updated on that event because it is trapped in the scheduler and the updates functions are called.

Conclusion

This paper shows how modeling can be improved using polymorphism, and addresses debugging aspects in the presence of pointers that enable the use of polymorphism in C++. We have exposed why polymorphism is useful, and addressed the debugging issues it raises. But, still, this is at a very early stage in the experimentations. Many issues need to be addressed concerning debugging and synthesis. For instance, for tracing of attribute less interfaces, the watching of data values becomes impossible at the base class level. In this case, the designer has to use console printing debugging techniques or a standard debugger like gdb. We are also working on other techniques to animate an object diagram to be able to visualize the flow of control and data in a design modelized in C++ for a pipeline fashion view.

The relevance of the usage of polymorphism as a high level hardware design technique is obvious, but there ar still many challenges before it can be proven functional and usable by designers.

Acknowledgments

The authors would also like to acknowledge support from National Science Foundation award number NSF CCR-9806898, from DARPA/ITO DABT63-98-C-004, and from Synopsys Corporation (under UC Micro program) and for providing the Scenic/SystemC v0.8 class library.

References

[1] CynApps. Cynapps home page. http://www.cynapps.com.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Assison-Wesley, 1995.

[3] J. L. Hennessy and D. A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[4] S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf. Object-oriented techniques in hardware design. *Computer*, July 1994.

[5] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in c/c++. In *Proceedings of HLDVT*, 1999.

[6] L. Semeria and G. D. Michelli. Spc: Systhesis of pointers in c. In *Proceedings of ICCAD*, 1998.

[7] SystemC. Systemc home page. http://www.systemc.org.

[8] D. Verkest, J. Cockx, Potargent, G. F., Jong, and H. De Man. On the use of c++ for system-on-chip design. In *Computer Society Workshop on VLSI*. IEEE, April 1999.

[9] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *Computer Society Workshop on VLSI*. IEEE, April 1999.

[10] C. Weiler, U. Kebschull, and W. Rosenstiel. C++ base classes for specification, simulation and partitioning of a hardware/software system. In *VLSI'95*. IEEE, 1995.

[11] W. Wolf. Object-oriented cosynthesis of distributed embeded systems. *ACM Transactions on Design Automation of Electronic Systems*, 1(3):301–314, July 1996.

A Source Code

A.1 dlx_wif_trace_file_c.hh

```
#include <wif_trace.h>
#include "instruction_c.hh"

#ifndef DLX_WIF_TRACE_FILE_C
#define DLX_WIF_TRACE_FILE_C

class dlx_wif_trace_file_c: public wif_trace_file {
  friend class dlx_wif_id_ex_reg_trace_c;
  friend class dlx_wif_instruction_ptr_c;

public:
  dlx_wif_trace_file_c(const char* name): wif_trace_file(name) {};

  void trace(instruction_c* const & object, const sc_string& name);
};

void dlx_sc_trace(dlx_wif_trace_file_c* tf,
  instruction_c* const & object,
  const sc_string&      name);

///////// from wif_trace.cpp
class wif_trace {
public:
  wif_trace(const sc_string& _name, const sc_string& _wif_name);

  // Needs to be pure virtual as has to be defined by the particular
  // type being traced
  virtual void write(FILE* f) = 0;
  virtual void set_width();

  // Comparison function needs to be pure virtual too
  virtual bool changed() = 0;

  //Got to declare this virtual as this will be overwritten by one base class
  virtual void print_variable_declaration_line(FILE* f);

  virtual ~wif_trace();

  const sc_string name;     // Name of the variable
  const sc_string wif_name; // Name of the variable in WIF file
  const char* wif_type;     // WIF data type
  int bit_width;
};
//////////////////////

dlx_wif_trace_file_c *create_dlx_wif_trace_file_c(const char * name);

#endif /* DLX_WIF_TRACE_FILE_C */
```

A.2 dlx_wif_trace_file_c.cc

```
#include "dlx_wif_trace_file_c.hh"
#include "dlx_wif_instruction_ptr_c.hh"
#include <iostream.h>


void dlx_wif_trace_file_c::trace(instruction_c* const & object,
 const sc_string& name) {
  if (initialized) {
    cerr << "DLX WIF Error: No traces can be added";
    cerr << "once simulation has started" << endl;
    assert(false);
  }

  sc_string temp_wif_name;
  create_wif_name(&temp_wif_name);
  traces.append(new dlx_wif_instruction_ptr_c(object, name, temp_wif_name, this));
}


dlx_wif_trace_file_c *create_dlx_wif_trace_file_c(const char * name)
{
  dlx_wif_trace_file_c *tf;

  tf = new dlx_wif_trace_file_c(name);
  sc_get_curr_simcontext()->add_trace_file(tf);
  the_dumpfile = tf; // To help sc_dumpall()
  return tf;
}

void dlx_sc_trace(dlx_wif_trace_file_c* tf,
  instruction_c* const & object,
  const sc_string& name) {
  tf->trace(object, name);
}
```

A.3 dlx_wif_instruction_ptr_c.hh

```
#include <wif_trace.h>
#include "dlx_wif_trace_file_c.hh"
#include "instruction_c.hh"

#ifndef DLX_WIF_INSTRUCTION_PTR_C
#define DLX_WIF_INSTRUCTION_PTR_C

class dlx_wif_instruction_ptr_c: public wif_trace {
private:
  dlx_wif_trace_file_c*  the_trace_file;
  instruction_c* const & object;
  instruction_c*         old_value;
  fixed_fields_c         ff_old_value;

  sc_string instruction_ptr_wif_name;
  sc_string ff_ir_wif_name;
  sc_string ff_op_a_wif_name;
  sc_string ff_op_b_wif_name;

public:
  dlx_wif_instruction_ptr_c(instruction_c* const & _object,
    const sc_string&       _name,
    const sc_string&       _wif_name,
    dlx_wif_trace_file_c*  tr
    );
  virtual bool changed();
  virtual void write(FILE* f);
  virtual void print_variable_declaration_line(FILE* f);

  void handle_unsigned(unsigned, char*);

};

#endif /* DLX_WIF_INSTRUCTION_PTR_C */
```

A.4 dlx_wif_instruction_ptr_c.cc

```
#include "dlx_wif_instruction_ptr_c.hh"


dlx_wif_instruction_ptr_c::\
dlx_wif_instruction_ptr_c(instruction_c* const & _object,
  const sc_string&       _name,
  const sc_string&       _wif_name,
  dlx_wif_trace_file_c*  the_trace_file_in):
  wif_trace(_name, _wif_name), object(_object) {

  wif_type= "BIT";
  bit_width= 32; // Put that as a more subtle parameter later
  the_trace_file= the_trace_file_in;
};


bool dlx_wif_instruction_ptr_c::changed() {
  bool return_value= false;
  if (object != old_value) {
    return_value= true;
  }
  else {
    if (object != NULL) {
      if (object->ff != ff_old_value) {
return_value= true;
      }
      else {
return_value= false;
      }
    }
  }

  return return_value;
};



void dlx_wif_instruction_ptr_c::print_variable_declaration_line(FILE* f) {
  // Dump what is pointed to by the instruction pointer
  sc_string ff_ir_name= name + ".ff.ir";
  the_trace_file->create_wif_name(&ff_ir_wif_name);
  fprintf(f, "declare  %s   \"%s\"  %s  ",
  (const char *) ff_ir_wif_name,
  (const char *) ff_ir_name, "BIT");
  fprintf(f, "0 31 ");
  fprintf(f, "variable ;\n");
  fprintf(f, "start_trace %s ;\n", (const char *) ff_ir_wif_name);

  sc_string ff_op_a_name= name + ".ff.op_a";
  the_trace_file->create_wif_name(&ff_op_a_wif_name);
  fprintf(f, "declare  %s   \"%s\"  %s  ",
```

```
  (const char *) ff_op_a_wif_name,
  (const char *) ff_op_a_name, "BIT");
  fprintf(f, "0 31 ");
  fprintf(f, "variable ;\n");
  fprintf(f, "start_trace %s ;\n", (const char *) ff_op_a_wif_name);

  sc_string ff_op_b_name= name + ".ff.op_b";
  the_trace_file->create_wif_name(&ff_op_b_wif_name);
  fprintf(f, "declare  %s   \"%s\"  %s  ",
  (const char *) ff_op_b_wif_name,
  (const char *) ff_op_b_name, "BIT");
  fprintf(f, "0 31 ");
  fprintf(f, "variable ;\n");
  fprintf(f, "start_trace %s ;\n", (const char *) ff_op_b_wif_name);

};



void dlx_wif_instruction_ptr_c::write(FILE* f) {

  char buf[1000];

  if (object== NULL) {
    fprintf(f, "assign %s \"11111111111111111111111111111111\" ;\n",
    (const char *) ff_ir_wif_name);
    fprintf(f, "assign %s \"11111111111111111111111111111111\" ;\n",
    (const char *) ff_op_a_wif_name);
    fprintf(f, "assign %s \"11111111111111111111111111111111\" ;\n",
    (const char *) ff_op_b_wif_name);
  } else {
    handle_unsigned(object->ff.ir, buf);
    fprintf(f, "assign %s \"%s\" ;\n", (const char *) ff_ir_wif_name, buf);
    handle_unsigned(object->ff.op_a, buf);
    fprintf(f, "assign %s \"%s\" ;\n", (const char *) ff_op_a_wif_name, buf);
    handle_unsigned(object->ff.op_b, buf);
    fprintf(f, "assign %s \"%s\" ;\n", (const char *) ff_op_b_wif_name, buf);
  }

  old_value = object;
  if (object != NULL)
    ff_old_value= object->ff;
};



// This method is not very pointer safe... please dont’ do stupid stuff with it
void dlx_wif_instruction_ptr_c::handle_unsigned(unsigned the_number,
char* buf) {
  // the width is assumed to be 32 bits for now...
  int bitindex;

  unsigned mask;
```

```
  //  bit_width = 32; // assume like that for now
  if (bit_width < 32) {
    mask = ~(-1 << bit_width);
  } else {
    mask = 0xffffffff;
  }


  // Check for overflow
  if ((the_number & mask) != the_number) {
    for (bitindex = 0; bitindex < bit_width; bitindex++){
      buf[bitindex] = '0';
    }
  }
  else{
    unsigned bit_mask = 1 << (bit_width-1);
    for (bitindex = 0; bitindex < bit_width; bitindex++) {
      buf[bitindex] = (the_number & bit_mask)? '1' : '0';
      bit_mask = bit_mask >> 1;
    }
  }
  buf[bitindex] = '\0';

};
```