TORC

Coding Style Guide

Version 0.1

1. Introduction

1.1 TORC Project

This project is named TORC (Tools for Open Reconfigurable Computing), and is intended to be open-sourced when ready for release. Figure 1 depicts the TORC framework.

Interra Systems, Inc. (Interra) is developing the EDIF importer and exporter, along with the generic netlist (nominally an EDIF object model).

Virginia Tech (VT) is developing the packer and unpacker, and is contributing to MySQLBits and MySQLDelay, along with other non-code tasks.

Brigham Young University (BYU) is contributing to MySQLBits, along with other non-code tasks.

USC Information Sciences Institute (ISI) is developing the BLIF importer and exporter, the XDL importer and exporter and physical netlist (nominally an XDL object model), the router and placer, and the device architecture database.

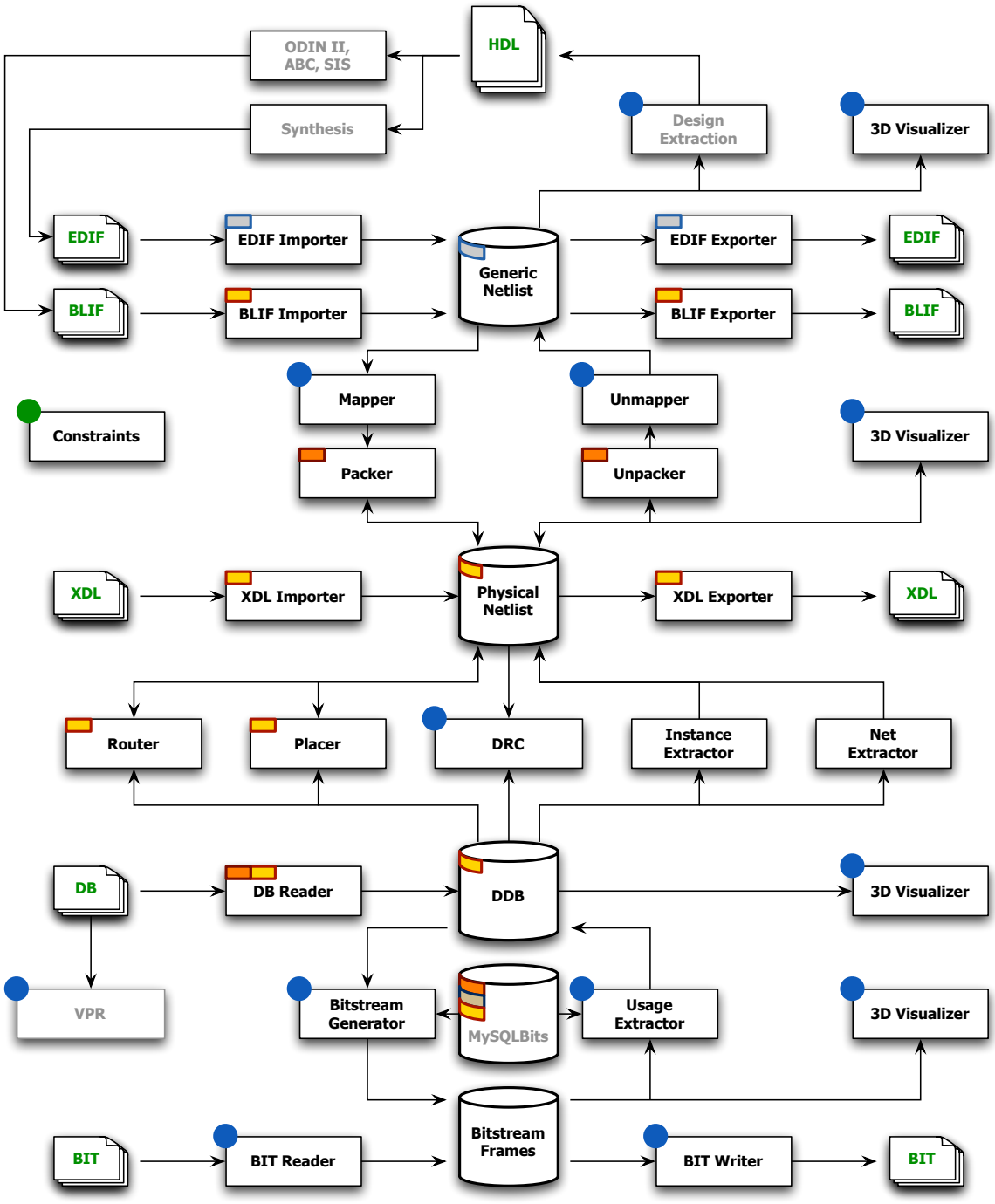

Figure 1. TORC Framework

2. Coding Style

1.2 File Naming

C++ header files will use a .hpp extension. C++ source files will use a .cpp extension.

1.3 Wrapping

Lines should wrap at the 100 character position.

1.4 Tabulation

The preferred approach uses 4 character tabs. Spaces are an acceptable alternative in the Interra code.

1.5 Notice

TORC is expected to be open-sourced when ready for release. Until that time, a simple license or copyright notice can serve as a placeholder.

```
// TORC - Copyright 2010 University of Southern California.  All Rights Reserved.
// $HeadURL: https://svn.east.isi.edu/gremlin/branches/neil-nmt-branch/Pip.hpp $
// $Id: Pip.hpp 665 2010-04-30 04:53:27Z nsteiner $
```

1.6 Guards

Header files should use include guards to avoid multiply defined declarations. The format of the guard name is NAMESPACE_FILENAME_HPP, where word breaks are replaced by underscores. For example, for the file CodingStyle.cpp, the following naming is appropriate:

```
#ifndef TORC_CODING_STYLE_HPP
#define TORC_CODING_STYLE_HPP

// stuff

#endif // TORC_CODING_STYLE_HPP
```

If the namespace is compound, every part of it should be included:

```
#define TORC_PHYSICAL_VERY_LONG_FILE_NAME_HPP
```

1.7 Namespaces

Namespaces are entirely lowercased. All project classes and code should live inside the torc namespace, or an appropriate subspace:

- Generic (EDIF) netlist object model: torc::generic
- Physical (XDL) netlist object model: torc::physical
- Architecture support (DB): torc::architecture
- Bitstream support: torc::bitstream

Components belonging or directly pertaining to any of the above databases should reside in their same namespaces. Components that merely work with those databases should be placed in the torc namespace, or other namespaces deemed appropriate.

1.8 Naming

Names will generally be rendered in CamelCase. Names should generally be descriptive rather than abbreviated. When interfacing with libraries that use other conventions, the other convention may be borrowed as appropriate.

1.8.1 Class and Enum Names

Classes, enums, and related typedefs will begin with an uppercase letter.

```
class CodingStyle;
enum EState {};
```

1.8.2 Variable and Function Names

Variable and functions will begin with a lowercase letter.

```
uint32_t someDescriptiveName = 0;
void sampleFunction(void);
```

1.8.3 Infix Acronyms

If an uppercase acronym appears within a name, all but the first acronym letter will be lowercased.

```
bool mEdifFlag; // even though EDIF is normally uppercased
```

1.8.4 Member Variables

Class members are prefixed with m.

```
string mMember;
```

1.8.5 Static Variables

Class members are prefixed with s, whether or not they belong to a class. *Warning: static variables which are not constants should be avoided at all cost, to prevent multi-threading issues.*

```
static string sStatic;
```

1.8.6 Constants

Declared constants are prefixed with c, whether or not they belong to a class.

```
const string cConstant;
```

1.8.7 Enum Members

Enums are prefixed with E. Enum members are prefixed with e, and usually with the base name of the enum.

```
enum EState { eStateNull, eStateOne, eStateTwo, eStateThree, eStateFour };
```

1.8.8 Pointer Variables

Pointers *may* be suffixed with an explicit Ptr for clarity.

```
void* mVoidPtr;
```

1.8.9 Private Member Variables

Private members generally follow the naming of other member variables. However it is acceptable to instead suffix the name with _.

```
bool mPrivate;
uint32_t private_;
```

1.9 Formatting

Brace and spacing guidelines are as follows.

1.9.1 Braces

The preferred brace formatting style follows K&R, but treats functions in the same manner as other constructs: Since we do not use ANSI style parameter declarations, there is no need to put the opening brace on a separate line.

```
// function definition example
const char* CodingStyle::getPipDirectionString(void) const {
    return sPipDirectionStrings[mPipDirection];
}


// loop example
for(uint16_t i = 0; i < 1024; i++) {
    // do something
    someDescriptiveName++;
}


// if-else example
if(mPrivate) {
    // do a whole bunch of stuff

// description of next condition
} else if(someDescriptiveName < 10) {
    // do a bunch more stuff

// description of next condition
} else if(someDescriptiveName > 1024) {
    // do yet more stuff

// description of default
} else {
    // perform some default behavior

}
```

Blocks can be placed on a single line if they fit, since this sometimes improves readability.

```
EPipDirection getPipDirection(void) const { return mPipDirection; }
```

1.9.2 Spacing

Spaces should be placed after parameters in function prototypes and calls, and after semicolons in for statements.

```
bool result = complexFunction(name, indexVector, indexToNameMap);


for(uint16_t i = 0; i < 1024; i++) {
    // do something
}
```

1.10 Classes

Classes are preferred over structs. Structs should not be used except for data structures that contain little or no code.

1.10.1 Visibility

TORC is an API. The preferred visibility for non-public functions is consequently protected. The coder can demote the visibility to private where appropriate. This practice is open to debate.

1.10.2 Accessor Methods

Accessors should follow standard conventions:

```
const Pin getSource(void) const { return *mSource; }
void setSource(const Pin& inSource) { *mSource = inSource; }
bool hasSource(void) const { return mSource != 0; }
```

1.11 Functions

Use of directional prefixes in function prototypes is encouraged where appropriate. in-, out-, and inout- denote parameters that are inputs, outputs, or both inputs and outputs for the function.

```
bool complexFunction(const string& inName, Uint64Vector& inIndexVector,
    Uint64ToStringMap& inoutIndexToNameMap);
```

3. CodingStyle.hpp

```
// TORC - Copyright 2010 University of Southern California.  All Rights Reserved.
// $HeadURL: https://svn.east.isi.edu/gremlin/branches/neil-nmt-branch/Pip.hpp $
// $Id: Pip.hpp 665 2010-04-30 04:53:27Z nsteiner $

#ifndef TORC_CODING_STYLE_HPP
#define TORC_CODING_STYLE_HPP

#include <stdint.h>
#include <string>
#include <vector>
#include <map>

// namespaces are lowercased
namespace torc {

/// \brief Coding style synopsys.
/// \details Extended dummy description for coding style class.  Lines should be wrapped at 100
///     characters.  4 character tabs are preferred, but spaces are permissible in torc::generic.
class CodingStyle {

    // internal convenience typedefs are just my way of avoiding lots of namespace usage
    // in the class definition.
    typedef std::string string;
    typedef std::vector<uint64_t> Uint64Vector;
    typedef std::map<uint64_t, string> Uint64ToStringMap;

public:
    /// \brief Pip direction enumeration.
    /// \details Setting the enumerations to constants is appropriate when we need to respect
    ///     external constants.  Another common trick is to declare a final ...Count entry, for use
    ///     as an automatic count, even if entries are added or removed.  Refer to the cpp file for
    ///     an example of how this is used.
    enum EPipDirection {
        ePipBidirectionalUnbuffered = 0,
        ePipBidirectionalUnidirectionallyBuffered,
        ePipBidirectionalBidirectionallyBuffered,
        ePipUnidirectionalBuffered,
        ePipDirectionCount
    };

    /// \brief Construct from tile and wire names and directionality.
    CodingStyle(const string& inMember, EPipDirection inPipDirection) : mMember(inMember),
        mPipDirection(inPipDirection), cConstant("constant"), mVoidPtr(0), mEdifFlag(false),
        mState(eStateNull),  mPrivate(false), private_(0) {}

    /// \brief Returns the member, with get___() semantics.
    const string& getMember(void) const { return mMember; }
    /// \brief Sets the member, with set___() semantics.
    void setMember(const string& inMember) { mMember = inMember; }
    /// \brief Returns the pip direction, with get___() semantics.
    EPipDirection getPipDirection(void) const { return mPipDirection; }
    /// \brief Sets the pip direction, with set___() semantics.
    void setPipDirection(EPipDirection inPipDirection) { mPipDirection = inPipDirection; }
    /// \brief Test functions are typically prefixed with has or sometimes is or alternatives.
    bool hasVoidPtr(void) const {
        // C++ uses 0 rather than NULL for uninitialized pointers
        return mVoidPtr != 0;
    }
    /// \brief Returns the pip direction as a string.
    const char* getPipDirectionString(void) const;
    /// \brief Sample function.
    void sampleFunction(void);

    /// \brief Example of a function accepting and returning multiple parameters
    bool complexFunction(const string& inName, Uint64Vector& inIndexVector,
        Uint64ToStringMap& inoutIndexToNameMap);

protected:
    /// \details In many cases there is no need to tie the enumerations to specific values.
    enum EState { eStateNull, eStateOne, eStateTwo, eStateThree, eStateSpecial, eStateFour };

    /// \brief Member variables are prefixed with m.
    string mMember;
    /// \brief Static variables are prefixed with s.
    static string sStatic;
```

```
    /// \brief Declared constants are prefixed with c.
    const string cConstant;
    /// \brief Pointers may be explicitly suffixed with Ptr.
    void* mVoidPtr;
    /// \brief Only the first letter of acronyms is capitalized when used within a name.
    bool mEdifFlag;
    /// \brief Another member variable.
    EPipDirection mPipDirection;
    /// \brief Yet another member variable
    EState mState;

    /// \brief String representation of pip directions.
    static const char* sPipDirectionStrings[];

private:
    /// \brief Private variables normally follow the naming convention described above.
    bool mPrivate;
    /// \brief Private variables may alternatively suffixed with _.
    uint32_t private_;
};

} // namespace torc

#endif // TORC_CODING_STYLE_HPP
```

4. CodingStyle.cpp

```cpp
// TORC - Copyright 2010 University of Southern California.  All Rights Reserved.
// $HeadURL: https://svn.east.isi.edu/gremlin/branches/neil-nmt-branch/Pip.cpp $
// $Id: Pip.cpp 659 2010-04-29 04:36:21Z nsteiner $

#include "CodingStyle.hpp"

namespace torc {

    /// \brief String representation of pip directions.
    const char* CodingStyle::sPipDirectionStrings[ePipDirectionCount] = {
        "==", // sPipBidirectionalUnbufferedString = 0,
        "=>", // sPipBidirectionalUnidirectionallyBufferedString,
        "=-", // sPipBidirectionalBidirectionallyBufferedString,
        "->"  // sPipUnidirectionalBufferedString
    };

    const char* CodingStyle::getPipDirectionString(void) const {
        return sPipDirectionStrings[mPipDirection];
    }

    /// \details Braces follow the K&R convention, with exceptions permitted when appropriate.
    void CodingStyle::sampleFunction(void) {

        // descriptive names are preferred in general
        uint32_t someDescriptiveName = 0;

        // loop counters of limited scope can follow traditional C naming conventions (i, j, k, ...)
        for(uint16_t i = 0; i < 1024; i++) {
            // do something
            someDescriptiveName++;
        }

        // booleans are very convenient in C++
        bool flag = true;
        while(flag) {
            // compact state machines can be coded with cases on single lines when that improves
            // readability
            switch(mState) {
            case eStateNull: if(someDescriptiveName-- != 0) { mState = eStateOne; } break;
            case eStateOne: /* do something; */ mState = mPrivate ? eStateTwo : eStateThree; break;
            case eStateTwo: /* do something; */ mState = eStateThree; break;
            case eStateThree: /* do something; */ mState = eStateFour; break;
            case eStateSpecial:
                someDescriptiveName++;
                mPrivate = !mPrivate;
                // explicit fallthrough
            case eStateFour: /* do something; */ mState = eStateOne; break;
            default: mState = eStateNull; break;
            }
        }

        // complex if-else structures can be written like this (blank lines not mandatory)
        if(mPrivate) {
            // do a whole bunch of stuff

        // description of next condition
        } else if(someDescriptiveName < 10) {
            // do a bunch more stuff

        // description of next condition
        } else if(someDescriptiveName > 1024) {
            // do yet more stuff

        // description of default
        } else {
            // perform some default behavior

        }

    }

} // namespace torc
```