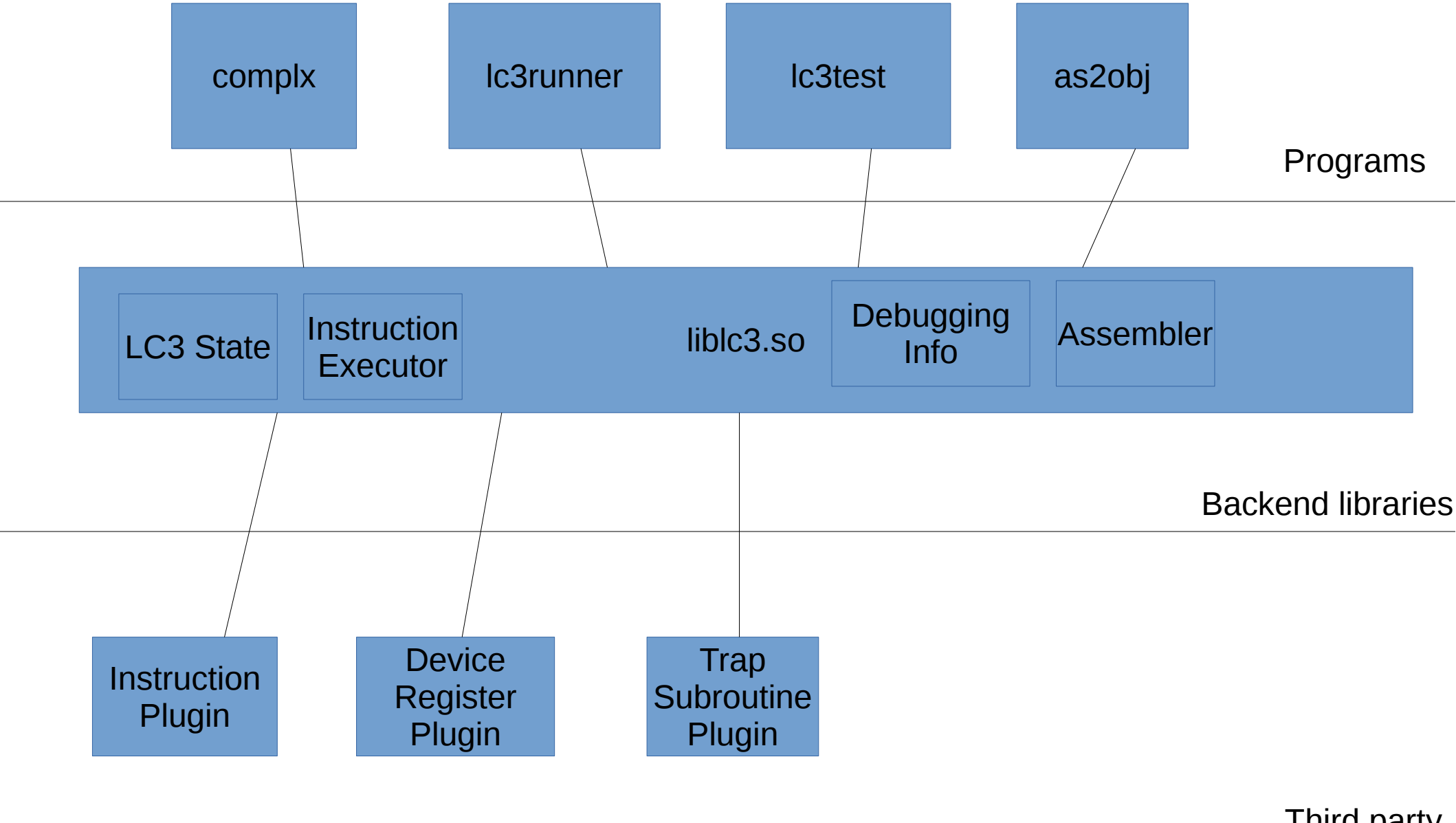
Complx plugin system

Extending an imaginary machine

Overview

- You can extend complx with plugins written in C++
- Add new devices (a screen, a clock, etc) new instructions (only 1), and new trap subroutines, extend the assembler (somewhat!) and much more!
- Can use in assembly code with the ;@plugin comment.

Architecture



Three main types of plugins

- Instruction plugins modify the undefined/error instruction in LC3 to make it do something else. Can only have one in use at a time.
- Device Register plugins act as a new device register any reads/writes to a specified memory address will do whatever you want. Akin to KBSR, KBDR, DSR, DDR, and MCR.
- Trap Subroutine plugins act as a new Trap anytime its called it will do whatever you want. Akin to GETC, OUT, IN, PUTS, PUTSP, and HALT

How to implement a plugin

- First derive a class from either Plugin, InstructionPlugin, DeviceRegisterPlugin, or TrapSubroutinePlugin depending on functionality needed.
- Implement the functions desired, including all pure virtual functions.
- Next implement create_plugin of this signature
  - Plugin* create_plugin(const PluginParams& params)
  - Params will contain a map of all parameters passed in on creation (see lc3_params.hpp for reading them in).
  - Ensure that there is only one instance of any plugin created.
- And a corresponding destroy_plugin of this signature
  - void destroy_plugin(Plugin* plugin)
  - Handle freeing any additional memory used by your plugin
- Be sure to extern “C” the create/destroy plugin declarations to prevent C++'s name mangling.

Compiling / Linking your plugin

- Must link with liblc3
- Simple makefile target
  - libmyplugin.so : myplugin.o

    g++ fPIC -Wall -shared -Wl,-soname,$@ $^ -o $@ -llc3
- Install it in /usr/lib or /usr/local/lib

Example Trap Plugin UDIV

- Specifications
  - UDIV is a subroutine that performs division and modulus.
  - Inputs R0 = n, R1 = d
  - Outputs R0 = n / d, R1 = n % d

- Usage (creation)
  - ;@plugin filename=lc3_udiv vector=x80
  - 1 parameter where to put the trap vector.

Implementation of UDIV

- class and create/destroy plugin functions

```
class UdivPlugin : public TrapFunctionPlugin
{
    public:
        UdivPlugin(unsigned char vector);
        ~UdivPlugin();
        std::string GetTrapName() const;
        void OnExecute(lc3_state& state, lc3_state_change& changes);
};

extern "C" Plugin* create_plugin(const PluginParams& params);
extern "C" void destroy_plugin(Plugin* ptr);
```

Implementation of UDIV II

```
Plugin* create_plugin(const PluginParams& params)
{
    if (instance != NULL)
        return instance;

    unsigned char vector;
    if (!lc3_params_read_uchar(params, "vector", vector))
    {
        fprintf(stderr, "Vector param (vector) not given or in incorrect format: %s\n",
lc3_params_get_value(params, "vector").c_str());
        return NULL;
    }

    instance = new UdivPlugin(vector);
    return instance;
}
```

Ensure only one instance is created

Parameter handling

Creation successful

Implementation of UDIV III

```
void destroy_plugin(Plugin* ptr)
{
    if (ptr == instance)
    {
        delete instance;
        instance = NULL;
    }
}
```

Implementation of UDIV IV

- When the program is assembled whenever it sees a comment with @plugin and the plugin is findablein the path it searches for library files (ex. /usr/lib, /usr/local/lib) it will load it and immediately call create_plugin.
- If the return value is non null then it will query the type of plugin and install it automatically binding it to the trap vector, if it can not then it will throw an error

Implementation of UDIV V

- UdivPlugin::UdivPlugin(unsigned char vector) : TrapFunctionPlugin(UDIV_MAJOR_VERSION, UDIV_MINOR_VERSION, "Division and Modulus Trap", vector) {}
- UdivPlugin::~UdivPlugin() {}
- std::string UdivPlugin::GetTrapName() const

  {

  return "UDIV";

  }

Note to the assembler
Saying UDIV will now assemble
To a trap instruction with vector found in parameter

Implementation of UDIV VI

```
void UdivPlugin::OnExecute(lc3_state& state, lc3_state_change& changes)
{
    changes.changes = LC3_MULTI_CHANGE;
    changes.info.push_back((lc3_change_info) {true, 0, (unsigned short)state.regs[0]});
    changes.info.push_back((lc3_change_info) {true, 1, (unsigned short)state.regs[1]});

    short r0 = state.regs[0];
    short r1 = state.regs[1];

    if (r1 != 0)
    {
        state.regs[0] = r0 / r1;
        state.regs[1] = r0 % r1;
    }
    else
    {
        state.regs[0] = -1;
        state.regs[1] = -1;
    }
}
```

Being a good citizen
Saves state of things
That will change.

The core of the code

lc3_state_change an aside

- Trap and Instruction plugins must record the values of everything that will change in the change object
- This is to get back stepping working, without this code the lc3 may not be able to get into its previous state when the back step button is used.
- Do not worry about saving PC, R7, or NZP bits they are automatically saved.

lc3 state change an aside.

```
enum lc3_change_t
{
    LC3_NO_CHANGE = 0,
    LC3_REGISTER_CHANGE = 1,
    LC3_MEMORY_CHANGE = 2,
    LC3_MULTI_CHANGE = 3,       // Multiple registers / memory addresses or both have changed.  For plugins.
    LC3_SUBROUTINE_BEGIN = 4,
    LC3_SUBROUTINE_END = 5,
    LC3_INTERRUPT_BEGIN = 6,    // Signals begin of interrupt can't backstep past this. (this will be in the undo stack while the interrupt is handled)
    LC3_INTERRUPT_END = 7,      // Signals end of interrupt (this will never be in the undo stack)
    LC3_INTERRUPT = 8,          // Signals a processed interrupt. (LC3_INTERRUPT_BEGIN changes to this after its processed.
};


typedef struct lc3_state_change
{
    unsigned short pc;
    short r7; /* In the case of fake traps two registers can be modified. So we have a special place for r7*/
    unsigned char n:1;
    unsigned char z:1;
    unsigned char p:1;
    unsigned char halted:1;
    unsigned char changes:4;
    unsigned short location;
    unsigned short value;
    unsigned int warnings;
    unsigned int executions;            // Only used for changes = LC3_INTERRUPT(_BEGIN) otherwise we know its changed by 1.
    lc3_subroutine_call subroutine;     // Only used for changes = LC3_SUBROUTINE_*
    std::vector<lc3_change_info> info;  // Only used for changes = LC3_MULTI_CHANGE
} lc3_state_change;
```

Test program for UDIV

```
;@plugin filename=lc3_udiv vector=x80

.orig x3000
    LD R0, A
    LD R1, B
    udiv ; or trap x80
    HALT

    A .fill 2000
    B .fill 8

.end
```

Examples

- Device register plugin → Random Number generator lc3_plugins/random.cpp
- Trap subroutine plugin → UDIV lc3_plugins/udiv.cpp
- Instruction plugin → Multiply Instruction lc3_plugins/multiply.cpp
- Custom plugin → BWLCD / ColorLCD lc3_plugins/bwlcd.cpp lc3_plugins/colorlcd.cpp