

---

# **Tutorial: Creating an LLVM Toolchain for the Cpu0 Architecture**

***Release 3.7.0***

**Chen Chung-Shu    gamma\_chen@yahoo.com.tw**  
**Anoushe Jamshidi    ajamshidi@gmail.com**

September 25, 2015



## CONTENTS

<b>1</b>	<b>About</b>	<b>1</b>
1.1	Authors . . . . .	1
1.2	Acknowledgments . . . . .	1
1.3	Revision history . . . . .	1
1.4	Licensing . . . . .	2
1.5	Outline of Chapters . . . . .	2
1.6	Outline of Chapters . . . . .	2
<b>2</b>	<b>Cpu0 ELF linker</b>	<b>3</b>
2.1	ELF to Hex . . . . .	4
2.2	Create Cpu0 backend under LLD . . . . .	24
2.3	Summary . . . . .	107
<b>3</b>	<b>Optimization</b>	<b>109</b>
3.1	LLVM IR optimization . . . . .	109
3.2	Project . . . . .	113
<b>4</b>	<b>Library</b>	<b>115</b>
4.1	Compiler-rt . . . . .	115
4.2	Avr libc . . . . .	115
4.3	Software Float Point Support . . . . .	116
<b>5</b>	<b>Book example code</b>	<b>125</b>
<b>6</b>	<b>Alternate formats</b>	<b>127</b>
<b>7</b>	<b>Presentation files</b>	<b>129</b>



## ABOUT

- Authors
- Acknowledgments
- Revision history
- Licensing
- Outline of Chapters
- Outline of Chapters

## 1.1 Authors

陳鍾樞

Chen Chung-Shu [gamma\\_chen@yahoo.com.tw](mailto:gamma_chen@yahoo.com.tw)  
<http://jonathan2251.github.io/web/index.html>

## 1.2 Acknowledgments

I would like to thank Sean Silva, [chisophugis@gmail.com](mailto:chisophugis@gmail.com), for his help, encouragement, and assistance with the Sphinx document generator. Without his help, this book would not have been finished and published online. Also thanking those corrections from readers who make the book more accurate.

## 1.3 Revision history

Version 3.7.1, Not release yet

**Version 3.7.0, Released September 24, 2015** Porting to lld 3.7.

**Version 3.6.2, Released May 4, 2015** Move some test from lbt to lbd. Remove warning in build Cpu0 code.

**Version 3.6.1, Released March 22, 2015** Correct typing.

**Version 3.6.0, Released March 8, 2015** Porting to lld 3.6.

## 1.4 Licensing

<http://llvm.org/docs/DeveloperPolicy.html#license>

## 1.5 Outline of Chapters

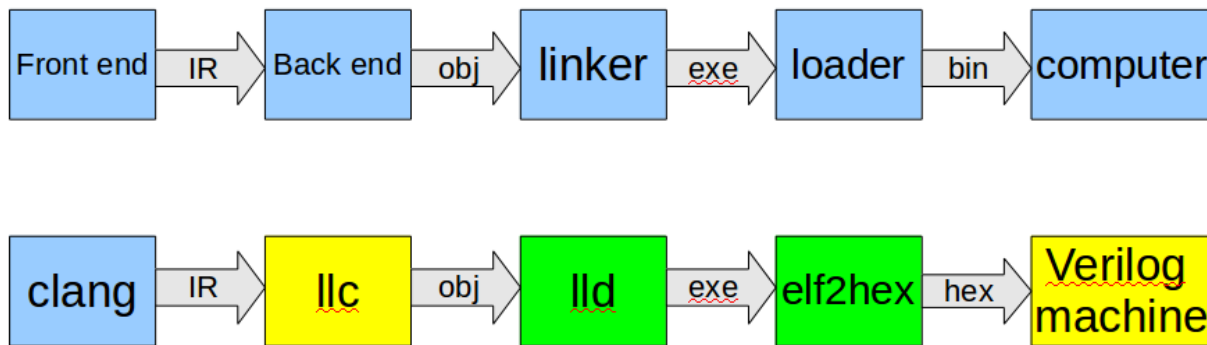


Figure 1.1: Code generation and execution flow

The upper half of Figure 1.1 is the work flow and software package of a computer program be generated and executed. IR stands for Intermediate Representation. The lower half is this book's work flow and software package of the toolchain extended implementation based on llvm. Except clang, the other blocks need to be extended for a new backend development. This book implement the green boxes part. The Cpu0 llvm backend can be find on <http://jonathan2251.github.io/lbd/index.html>.

This book include:

1. ELF linker for Cpu0 which extended from lld. Chapter 2.
2. The elf2hex extended from llvm-objdump. Chapter 2.
3. Optimization. Chapter 3.
4. Porting C standard library from avr libc and software floating point library from LLVM compiler-rt.

With these implementation, reader can generate Cpu0 machine code through Cpu0 llvm backend compiler, linker and elf2hex, then see how it runs on your computer. The pdf and epub are also available in the web.

## 1.6 Outline of Chapters

*Cpu0 ELF linker:*

Develop ELF linker for Cpu0 backend based on lld project.

*Optimization:*

Backend independent optimaization. Under working and I need to avoid something I know since I working with a proprietary company.

*Library:*

Software floating point library and standard C library supporting. Under working.

## CPU0 ELF LINKER

- ELF to Hex
- Create Cpu0 backend under LLD
  - Setup Cpu0 backend under lld
  - Cpu0 backend source code
  - LLD introduction
    - \* How LLD do the linker job
    - \* Linking Steps
      - Command line processing
      - Parsing input files
      - Resolving
      - Passes/Optimizations
      - Generate Output File
  - Static linker
    - \* Run
    - \* Cpu0 lld structure
  - Dynamic linker
    - \* Run
    - \* How to work
- Summary
  - Create a new backend base on LLVM
  - Contribute back to Open Source through working and learning

LLD changes quickly and the figures of this chapter is not up to date. Like llvm, lld linker include a couple of target in ELF format handling. The term Cpu0 backend used in this chapter can refer to the ELF format handling for Cpu0 target machine under lld, llvm compiler backend, or both. But supposing readers will easy knowing what it refer to.

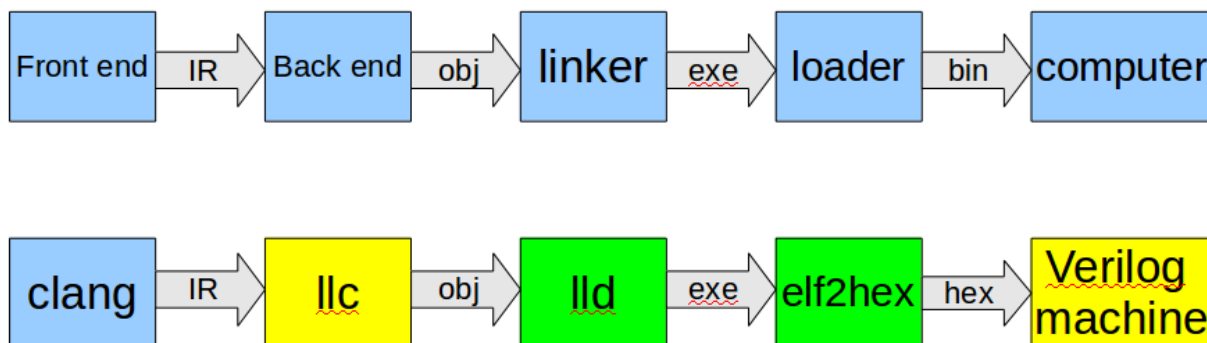


Figure 2.1: Code generation and execution flow

As depicted in Figure 2.1 of chapter About. Beside llvm backend, we implement ELF linker and elf2hex to run on Cpu0 verilog simulator. This chapter extends lld to support Cpu0 backend as well as elf2hex to replace Cpu0 loader. After link with lld, the program with global variables can be allocated in ELF file format layout. Meaning the relocation records of global variables is resolved. In addition, llvm-objdump driver is modified for supporting generate Hex file from ELF by command `llvm-objdump -elf2hex`. With these two tools supported, the global variables exists in section `.data` and `.rodata` can be accessed and transfered to Hex file which feeds to Verilog Cpu0 machine and run on your PC/Laptop.

As the previous chapters mentioned, Cpu0 has two relocation models for static link and dynamic link, respectively, which controlled by option `-relocation-model` in `lld`. This chapter supports the static link fully, and part of dynamic link for demonstration purpose. Since dynamic link needs the OS involvement and the Cpu0 is run on Verilog bare metal simulator, the dynamic linker program is added and the Cpu0 Verilog code is extended to demonstrate these Cpu0 PIC mode instructions are work correctly in dynamic link. However, these part of Cpu0 Verilog codes are not needed in a real machine with OS support, and the Cpu0 lld's and elf2hex's dynamic linker function is not full implemented. They are programmed by a specific shared library name since the shared library locating needs the OS's help (files management is part of OS's job). Without OS, these things cannot be solved and dynamic link is impossible to finish. Anyway, for the dynamic link demonstration, we can implement dynamic linker program and adapt lld, elf2hex, and Cpu0 Verilog code to support a specific shared library and verify the dynamic link result. In reality, the Micro CPUs without OS or tiny OS inside only support static link for C language.

About lld please refer LLD web site here <sup>1</sup> and LLD install requirement on Linux here <sup>2</sup>. Currently, lld can be built by: gcc and clang 3.5 compiler on Ubuntu, and gcc on Fedora, as I have tried. On iMac, lld can be built by clang with the Xcode version as the next sub section. If you run with Virtual Machine (VM), please keep your physical memory size setting over 1GB to avoid insufficient memory link error.

## 2.1 ELF to Hex

Add `elf2hex.h`, `elf2hex-dlink.h` and update `llvm-objdump` driver to support ELF to Hex for Cpu0 backend as follows,

### `exlbt/llvm-objdump/elf2hex-dlinker.h`

```
#ifndef DLINK

class Cpu0DynFunIndex {
private:
    char soStrtab[20][100];
    int soStrtabSize = 0;

    char exePltName[20][100];
    int exePltNameSize = 0;

    int findPltName(const char* pltName);
public:
    void createPltName(const ObjectFile *o);
    void createStrtab();
    uint16_t correctDynFunIndex(const char* pltName);
};

int Cpu0DynFunIndex::findPltName(const char* pltName) {
    for (int i = 0; i < exePltNameSize; i++)
        if (strcmp(pltName, exePltName[i]) == 0)
```

---

<sup>1</sup> <http://lld.llvm.org/>

<sup>2</sup> [http://lld.llvm.org/getting\\_started.html#on-unix-like-systems](http://lld.llvm.org/getting_started.html#on-unix-like-systems)



```

        return i;
    return -1;
}

void Cpu0DynFunIndex::createPltName(const ObjectFile *Obj) {
    std::error_code ec;
    std::string Error;

    for (const SectionRef &Section : Obj->sections()) {
        if (error(ec)) return;
       StringRef Name;
        StringRef Contents;
        uint64_t BaseAddr;
        bool BSS;
        if (error(Section.getName(Name))) continue;
        if (error(Section.getContents(Contents))) continue;
        BaseAddr = Section.getAddress();
        BSS = Section.isBSS();

        if (Name == ".strtab") {
            int num_dyn_entry = 0;
            FILE *fd_num_dyn_entry;
            fd_num_dyn_entry = fopen("dlconfig/num_dyn_entry", "r");
            if (fd_num_dyn_entry != NULL) {
                fscanf(fd_num_dyn_entry, "%d", &num_dyn_entry);
            }
            fclose(fd_num_dyn_entry);

            for (std::size_t addr = 2+strlen(".PLT0"), end = Contents.size();
                addr < end; ) {
                if (Contents.substr(addr, strlen("__plt_")) != "__plt_")
                    break;
                strcpy(exePltName[exePltNameSize], Contents.data()+addr);
                addr = addr + strlen(exePltName[exePltNameSize]) + 1;
                exePltNameSize++;
            }
            break;
        }
    }
}

void Cpu0DynFunIndex::createStrtab() {
    FILE *fd_dynstrAscii;

    fd_dynstrAscii = fopen("dlconfig/dynstrAscii", "r");
    if (fd_dynstrAscii == NULL)
        fclose(fd_dynstrAscii);
    assert(fd_dynstrAscii != NULL && "fd_dynstr == NULL");
    int i = 0;
    // function                result on EOF or error
    // -----                -----
    // fgets()                  NULL
    // fscanf()                 number of succesful conversions
    //                          less than expected
    // fgetc()                  EOF
    // fread()                  number of elements read
    //                          less than expected
    int j = 0;

```

```
for (i=0; i < soStrtabSize; i++) {
    j=fscanf(fd_dynstrAscii, "%s", soStrtab[i]);
    if (j != 1)
        break;
}
soStrtabSize = i;
fclose(fd_dynstrAscii);
}

uint16_t Cpu0DynFunIndex::correctDynFunIndex(const char* pltName) {
    int i = findPltName(pltName);
    if (i != -1) {
        int j = 0;
        for (j=0; j < soStrtabSize; j++)
            if (strcmp(soStrtab[j], (const char*)exePltName[i]+strlen("__plt_")) == 0)
                break;
        if (j == soStrtabSize) {
            outs() << "cannot find " << exePltName[i] << "\n";
            exit(1);
        }
        j++;
        return (uint16_t)(j & 0xffff);
    }
    return (uint16_t)0;
}

Cpu0DynFunIndex cpu0DynFunIndex;

static void DisassembleSoInHexFormat(const ObjectFile *Obj
/*, bool InlineRelocs*/ , std::unique_ptr<MCDisassembler>& DisAsm,
std::unique_ptr<MCInstPrinter>& IP, uint64_t& lastDumpAddr,
std::unique_ptr<const MCSubtargetInfo>& STI) {
    std::string Error;
    uint64_t soLastPrintAddr = 0;
    FILE *fd_so_func_offset;
    int num_dyn_entry = 0;

    fd_so_func_offset = fopen("dlconfig/so_func_offset", "w");
    if (fd_so_func_offset == NULL)
        fclose(fd_so_func_offset);
    assert(fd_so_func_offset != NULL && "fd_so_func_offset == NULL");

#ifdef ELF2HEX_DEBUG
    errs() << format("!!lastDumpAddr %8" PRIx64 "\n", lastDumpAddr);
#endif
    std::error_code ec;
    for (const SectionRef &Section : Obj->sections()) {
        if (error(ec)) break;
       StringRef Name;
        StringRef Contents;
        uint64_t BaseAddr;
        if (error(Section.getName(Name))) continue;
        if (error(Section.getContents(Contents))) continue;
        BaseAddr = Section.getAddress();
#ifdef ELF2HEX_DEBUG
        errs() << "Name " << Name << format(" BaseAddr %8" PRIx64 "\n", BaseAddr);
        errs() << format("!!lastDumpAddr %8" PRIx64 "\n", lastDumpAddr);
#endif
    }
}
```

```

bool text;
text = Section.isText();
if (!text) {
    continue;
}
// It's .text section
uint64_t SectionAddr;
SectionAddr = Section.getAddress();

// Make a list of all the symbols in this section.
std::vector<std::pair<uint64_t,StringRef> > Symbols;
for (const SymbolRef &Symbol : Obj->symbols()) {
    if (Section.containsSymbol(Symbol)) {
        ErrorOr<uint64_t> AddressOrErr = Symbol.getAddress();
        if (error(AddressOrErr.getError()))
            break;
        uint64_t Address = *AddressOrErr;
        Address -= SectionAddr;

        ErrorOr<StringRef> Name = Symbol.getName();
        if (error(Name.getError()))
            break;
        Symbols.push_back(std::make_pair(Address, *Name));
    }
}

// Sort the symbols by address, just in case they didn't come in that way.
array_pod_sort(Symbols.begin(), Symbols.end());

// Make a list of all the relocations for this section.
std::vector<RelocationRef> Rels;

// Sort relocations by address.
std::sort(Rels.begin(), Rels.end(), RelocAddressLess);

StringRef SegmentName = "";
if (const MachOObjectFile *MachO =
    dyn_cast<const MachOObjectFile>(Obj)) {
    DataRefImpl DR = Section.getRawDataRefImpl();
    SegmentName = MachO->getSectionFinalSegmentName(DR);
}
StringRef name;
if (error(Section.getName(name))) break;
if (name == ".plt") continue;
outs() << "/*" << "Disassembly of section ";
if (!SegmentName.empty())
    outs() << SegmentName << ", ";
outs() << name << ':' << "*/";

// If the section has no symbols just insert a dummy one and disassemble
// the whole section.
if (Symbols.empty())
    Symbols.push_back(std::make_pair(0, name));

SmallString<40> Comments;
raw_svector_ostream CommentStream(Comments);

StringRef BytesStr;

```

```
if (error(Section.getContents(BytesStr))) break;
ArrayRef<uint8_t> Bytes(reinterpret_cast<const uint8_t *>(BytesStr.data()),
                        BytesStr.size());

uint64_t Size;
uint64_t Index;
uint64_t SectSize;
SectSize = Section.getSize();

std::vector<RelocationRef>::const_iterator rel_cur = Rels.begin();
std::vector<RelocationRef>::const_iterator rel_end = Rels.end();
// Disassemble symbol by symbol.
for (unsigned si = 0, se = Symbols.size(); si != se; ++si) {
    uint64_t Start = Symbols[si].first;
    uint64_t End;
    // The end is either the size of the section or the beginning of the next
    // symbol.
    if (si == se - 1)
        End = SectSize;
    // Make sure this symbol takes up space.
    else if (Symbols[si + 1].first != Start)
        End = Symbols[si + 1].first - 1;
    else {
        // This symbol has the same address as the next symbol. Skip it.
        fprintf(fd_so_func_offset, "%02x ",
                (uint8_t)(Symbols[si].first >> 24));
        fprintf(fd_so_func_offset, "%02x ",
                (uint8_t)((Symbols[si].first >> 16) & 0xFF));
        fprintf(fd_so_func_offset, "%02x ",
                (uint8_t)((Symbols[si].first >> 8) & 0xFF));
        fprintf(fd_so_func_offset, "%02x ",
                (uint8_t)(Symbols[si].first & 0xFF));
        std::string str = Symbols[si].second.str();
        std::size_t idx = 0;
        std::size_t strSize = 0;
        for (idx = 0, strSize = str.size(); idx < strSize; idx++) {
            fprintf(fd_so_func_offset, "%c%c ",
                    hexdigit((str[idx] >> 4) & 0xF, true),
                    hexdigit(str[idx] & 0xF, true));
        }
        for (idx = strSize; idx < 48; idx++) {
            fprintf(fd_so_func_offset, "%02x ", 0);
        }
        fprintf(fd_so_func_offset, "/* %s */\n", Symbols[si].second.begin());
        num_dyn_entry++;

        outs() << '\n' << "/*" << Symbols[si].second << "*/\n";
        continue;
    }

    soLastPrintAddr = Symbols[si].first;
    fprintf(fd_so_func_offset, "%02x ", (uint8_t)(Symbols[si].first >> 24));
    fprintf(fd_so_func_offset, "%02x ",
            (uint8_t)((Symbols[si].first >> 16) & 0xFF));
    fprintf(fd_so_func_offset, "%02x ",
            (uint8_t)((Symbols[si].first >> 8) & 0xFF));
    fprintf(fd_so_func_offset, "%02x ",
            (uint8_t)(Symbols[si].first & 0xFF));
    std::string str = Symbols[si].second.str();
```

```

std::size_t idx = 0;
std::size_t strSize = 0;
for (idx = 0, strSize = str.size(); idx < strSize; idx++) {
    fprintf(fd_so_func_offset, "%c%c ",
            hexdigit((str[idx] >> 4) & 0xF, true),
            hexdigit(str[idx] & 0xF, true));
}
for (idx = strSize; idx < 48; idx++) {
    fprintf(fd_so_func_offset, "%02x ", 0);
}
fprintf(fd_so_func_offset, "/* %s */\n", Symbols[si].second.begin());
num_dyn_entry++;

outs() << '\n' << "/*" << Symbols[si].second << "*/\n";
#ifdef NDEBUG
    raw_ostream &DebugOut = DebugFlag ? dbgs() : nulls();
#else
    raw_ostream &DebugOut = nulls();
#endif

for (Index = Start; Index < End; Index += Size) {
    MCInst Inst;
    if (DisAsm->getInstruction(Inst, Size, Bytes.slice(Index),
                               SectionAddr + Index, DebugOut,
                               CommentStream)) {
        outs() << format("/%8" PRIx64 "*/", lastDumpAddr + Index);
        if (!NoShowRawInsn) {
            outs() << "\t";
            dumpBytes(Bytes.slice(Index, Size), outs());
        }
        outs() << "/*";
        IP->printInst(&Inst, outs(), "", *STI);
        outs() << CommentStream.str();
        outs() << "*/";
        Comments.clear();
        outs() << "\n";
    } else {
        errs() << ToolName << ": warning: invalid instruction encoding\n";
        if (Size == 0)
            Size = 1; // skip illegible bytes
    }

    // outs() << "Size = " << Size << "Index = " << Index << "lastDumpAddr = "
    // << lastDumpAddr << "\n"; // debug
    // Print relocation for instruction.
    while (rel_cur != rel_end) {
        bool hidden = getHidden(*rel_cur);
        uint64_t addr = rel_cur->getOffset();
        SmallString<16> name;
        SmallString<32> val;

        // If this relocation is hidden, skip it.
        if (hidden) goto skip_print_rel;

        // Stop when rel_cur's address is past the current instruction.
        if (addr >= Index + Size) break;
        rel_cur->getTypeName(name);
        if (error(getRelocationValueString(*rel_cur, val)))

```

```
        goto skip_print_rel;

        outs() << format("\t\t\t\t/*%8" PRIx64 ":", SectionAddr + addr) << name
            << "\t" << val << "*/\n";

        skip_print_rel:
            ++rel_cur;
        }
        lastDumpAddr += Index;
    }
    soLastPrintAddr = End;
#ifdef ELF2HEX_DEBUG
    errs() << format("SectionAddr + Index = %8" PRIx64 "\n", SectionAddr + Index);
    errs() << format("lastDumpAddr %8" PRIx64 "\n", lastDumpAddr);
#endif
}
}

// Dump share obj or lib
// Fix the issue that __tls_get_addr appear as file offset 0.
// Old lld version the __tls_get_addr appear at the last function name.
std::pair<uint64_t, StringRef> dummy(soLastPrintAddr, "dummy");
fprintf(fd_so_func_offset, "%02x ", (uint8_t)(dummy.first >> 24));
fprintf(fd_so_func_offset, "%02x ", (uint8_t)((dummy.first >> 16) & 0xFF));
fprintf(fd_so_func_offset, "%02x ", (uint8_t)((dummy.first >> 8) & 0xFF));
fprintf(fd_so_func_offset, "%02x ", (uint8_t)(dummy.first & 0xFF));
std::string str = dummy.second.str();
std::size_t idx = 0;
std::size_t strSize = 0;
for (idx = 0, strSize = str.size(); idx < strSize; idx++) {
    fprintf(fd_so_func_offset, "%c%c ", hexdigit((str[idx] >> 4) & 0xF, true)
        , hexdigit(str[idx] & 0xF, true));
}
for (idx = strSize; idx < 48; idx++) {
    fprintf(fd_so_func_offset, "%02x ", 0);
}
fprintf(fd_so_func_offset, "/* %s */\n", dummy.second.begin());
num_dyn_entry++;
outs() << '\n' << "/*" << dummy.second << "*/\n";
FILE *fd_num_dyn_entry;
fd_num_dyn_entry = fopen("dlconfig/num_dyn_entry", "w");
if (fd_num_dyn_entry != NULL) {
    fprintf(fd_num_dyn_entry, "%d\n", num_dyn_entry);
}
fclose(fd_num_dyn_entry);
}
```

```
static void PrintSoDataSections(const ObjectFile *o, uint64_t& lastDumpAddr,
                               bool isLittleEndian) {
```

```
    std::error_code ec;

    for (const SectionRef &Section : o->sections()) {
        if (error(ec)) return;
        StringRef Name;
        StringRef Contents;
        uint64_t BaseAddr;
        bool BSS;
        if (error(Section.getName(Name))) continue;
        if (error(Section.getContents(Contents))) continue;
```

```

BaseAddr = Section.getAddress();
BSS = Section.isBSS();
if (Name == ".dynsym") {
    int num_dyn_entry = 0;
    FILE *fd_num_dyn_entry;
    fd_num_dyn_entry = fopen("dlconfig/num_dyn_entry", "r");
    if (fd_num_dyn_entry != NULL) {
        fscanf(fd_num_dyn_entry, "%d", &num_dyn_entry);
    }
    fclose(fd_num_dyn_entry);
    raw_fd_ostream fd_dynsym("dlconfig/dynsym", ec, sys::fs::F_Text);
    int count = 0;
    for (std::size_t addr = 0, end = Contents.size(); addr < end; addr += 16) {
        if (isLittleEndian) {
            fd_dynsym << hexdigit((Contents[addr+3] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+3] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+2] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+2] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+1] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+1] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr] >> 4) & 0xF, true)
                << hexdigit(Contents[addr] & 0xF, true) << " ";
        }
        else {
            fd_dynsym << hexdigit((Contents[addr] >> 4) & 0xF, true)
                << hexdigit(Contents[addr] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+1] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+1] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+2] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+2] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+3] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+3] & 0xF, true) << " ";
        }
        count++;
    }
    for (int i = count; i < num_dyn_entry; i++) {
        fd_dynsym << "00 00 00 00 ";
    }
}
else if (Name == ".dynstr") {
    raw_fd_ostream fd_dynstr("dlconfig/dynstr", ec, sys::fs::F_Text);
    raw_fd_ostream fd_dynstrAscii("dlconfig/dynstrAscii", ec,
        sys::fs::F_Text);
    for (std::size_t addr = 0, end = Contents.size(); addr < end; addr++) {
        fd_dynstr << hexdigit((Contents[addr] >> 4) & 0xF, true)
            << hexdigit(Contents[addr] & 0xF, true) << " ";
        if (addr == 0)
            continue;
        if (Contents[addr] == '\0')
            fd_dynstrAscii << "\n";
        else
            fd_dynstrAscii << Contents[addr];
    }
}
}
}

#endif // DLINK

```

**exlbt/llvm-objdump/elf2hex.h**

```
//===== elf2hex.cpp =====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
// This program is a utility that works with llvm-objdump.
//
//=====//

#include <stdio.h>
#include "llvm/Support/raw_ostream.h"
#define DLINK
//#define ELF2HEX_DEBUG

static cl::opt<bool>
ConvertElf2Hex("elf2hex",
cl::desc("Display the hex content of verilog cpu0 needed sections"));

static cl::opt<bool>
LittleEndian("le",
cl::desc("Little endian format"));

#ifdef DLINK
static cl::opt<bool>
DumpSo("cpu0dumpso",
cl::desc("Dump shared library .so"));

static cl::opt<bool>
LinkSo("cpu0linkso",
cl::desc("Link shared library .so"));

#include "elf2hex-dlinker.h"
#endif

#ifdef ELF2HEX_DEBUG
// Modified from PrintSectionHeaders()
static uint64_t GetSectionHeaderStartAddress(const ObjectFile *Obj,
StringRef sectionName) {
//  outs() << "Sections:\n"
//          "Idx Name          Size      Address          Type\n";
std::error_code ec;
unsigned i = 0;
for (const SectionRef &Section : Obj->sections()) {
if (error(ec)) return 0;
StringRef Name;
if (error(Section.getName(Name))) return 0;
uint64_t Address;
Address = Section.getAddress();
uint64_t Size;
Size = Section.getSize();
bool Text;
Text = Section.isText();

```



```

    if (Name == sectionName)
        return Address;
    else
        return 0;
    ++i;
}
return 0;
}
#endif

// Fill /*address*/ 00 00 00 00 [startAddr..endAddr] from startAddr to endAddr.
// Include startAddr and endAddr.
static void Fill0s(uint64_t startAddr, uint64_t endAddr) {
    std::size_t addr;

    assert((startAddr <= endAddr) && "startAddr must <= BaseAddr");
    // Fill /*address*/ bytes is odd for 4 by 00
    outs() << format("/%8" PRIx64 " */", startAddr);
    // Fill /*address*/ 00 00 00 00 for 4 bytes alignment (1 Cpu0 word size)
    for (addr = startAddr; addr < endAddr; addr += 4) {
        outs() << format("/%8" PRIx64 " */", addr);
        outs() << format("%02" PRIx64 " ", 0) << format("%02" PRIx64 " ", 0) \
        << format("%02" PRIx64 " ", 0) << format("%02" PRIx64 " ", 0) << '\n';
    }

    return;
}

static void PrintDataSection(const ObjectFile *o, uint64_t& lastDumpAddr,
    SectionRef Section) {
    std::string Error;
   StringRef Name;
   StringRef Contents;
    uint64_t BaseAddr;
    bool BSS;
    uint64_t size;
    if (error(Section.getName(Name))) return;
    if (error(Section.getContents(Contents))) return;
    BaseAddr = Section.getAddress();
    BSS = Section.isBSS();

    size = (Contents.size()+3)/4*4;
    if (Contents.size() <= 0) {
        return;
    }

    outs() << "/*Contents of section " << Name << "*/\n";
    // Dump out the content as hex and printable ascii characters.
    for (std::size_t addr = 0, end = Contents.size(); addr < end; addr += 16) {
        outs() << format("/%8" PRIx64 " */", BaseAddr + addr);
        // Dump line of hex.
        for (std::size_t i = 0; i < 16; ++i) {
            if (i != 0 && i % 4 == 0)
                outs() << ' ';
            if (addr + i < end)
                outs() << hexdigit((Contents[addr + i] >> 4) & 0xF, true)
                << hexdigit(Contents[addr + i] & 0xF, true) << " ";
        }
    }
}

```

```
// Print ascii.
outs() << "/*" << " ";
for (std::size_t i = 0; i < 16 && addr + i < end; ++i) {
    if (std::isprint(static_cast<unsigned char>(Contents[addr + i]) & 0xFF))
        outs() << Contents[addr + i];
    else
        outs() << ".";
}
outs() << "*/" << "\n";
}
for (std::size_t i = Contents.size(); i < size; i++) {
    outs() << "00 ";
}
outs() << "\n";
#ifdef ELF2HEX_DEBUG
errs() << "Name " << Name << " BaseAddr ";
errs() << format("%8" PRIx64 " Contents.size() ", BaseAddr);
errs() << format("%8" PRIx64 " size ", Contents.size());
errs() << format("%8" PRIx64 " \n", size);
#endif
// save the end address of this section to lastDumpAddr
lastDumpAddr = BaseAddr + size;
}

// Modified from DisassembleObject()
static void DisassembleObjectInHexFormat(const ObjectFile *Obj
/*, bool InlineRelocs*/ , std::unique_ptr<MCDisassembler>& DisAsm,
    std::unique_ptr<MCInstPrinter>& IP, uint64_t& lastDumpAddr,
    std::unique_ptr<const MCSubtargetInfo>& STI) {

#ifdef ELF2HEX_DEBUG
    errs() << format("!!lastDumpAddr %8" PRIx64 "\n", lastDumpAddr);
#endif
    std::error_code ec;
    for (const SectionRef &Section : Obj->sections()) {
        if (error(ec)) break;
       StringRef Name;
       StringRef Contents;
       uint64_t BaseAddr;
        if (error(Section.getName(Name))) continue;
        if (error(Section.getContents(Contents))) continue;
        BaseAddr = Section.getAddress();
        if (BaseAddr < 0x100)
            continue;
#ifdef ELF2HEX_DEBUG
        errs() << "Name " << Name << format(" BaseAddr %8" PRIx64 "\n", BaseAddr);
        errs() << format("!!lastDumpAddr %8" PRIx64 "\n", lastDumpAddr);
#endif
        bool text;
        text = Section.isText();
        if (!text) {
#ifdef ELF2HEX_DEBUG
            errs() << "!text\n";
#endif
        }
        if (lastDumpAddr < BaseAddr) {
            Fill0s(lastDumpAddr, BaseAddr - 1);
            lastDumpAddr = BaseAddr;
        }
    }
}
```

```

if (Name == ".got.plt") {
    uint64_t BaseAddr;
    BaseAddr = Section.getAddress();
#ifdef DLINK
    if (LinkSo) {
        raw_fd_ostream fd_global_offset("dlconfig/global_offset", ec,
                                         sys::fs::F_Text);

        fd_global_offset << format("%02" PRIx64 " ", BaseAddr >> 24);
        fd_global_offset << format("%02" PRIx64 " ", (BaseAddr >> 16) & 0xFF);
        fd_global_offset << format("%02" PRIx64 " ", (BaseAddr >> 8) & 0xFF);
        fd_global_offset << format("%02" PRIx64 " ", BaseAddr & 0xFF);
    }
#endif
    PrintDataSection(Obj, lastDumpAddr, Section);
}
else if ((Name == ".bss" || Name == ".sbss") && Contents.size() > 0) {
    uint64_t size = (Contents.size() + 3)/4*4;
    Fill0s(BaseAddr, BaseAddr + size - 1);
    lastDumpAddr = BaseAddr + size;
    continue;
}
else {
    PrintDataSection(Obj, lastDumpAddr, Section);
}
continue;
}
else {
    if (lastDumpAddr < BaseAddr) {
        Fill0s(lastDumpAddr, BaseAddr - 1);
        lastDumpAddr = BaseAddr;
    }
}
// It's .text section
uint64_t SectionAddr;
SectionAddr = Section.getAddress();
uint64_t SectSize = Section.getSize();
if (!SectSize)
    continue;

// Make a list of all the symbols in this section.
std::vector<std::pair<uint64_t, StringRef> > Symbols;
for (const SymbolRef &Symbol : Obj->symbols()) {
    if (Section.containsSymbol(Symbol)) {
        ErrorOr<uint64_t> AddressOrErr = Symbol.getAddress();
        if (error(AddressOrErr.getError()))
            break;
        uint64_t Address = *AddressOrErr;
        Address -= SectionAddr;
        if (Address >= SectSize)
            continue;

        ErrorOr<StringRef> Name = Symbol.getName();
        if (error(Name.getError()))
            break;
        Symbols.push_back(std::make_pair(Address, *Name));
    }
}

```

```
// Sort the symbols by address, just in case they didn't come in that way.
array_pod_sort(Symbols.begin(), Symbols.end());
#ifdef ELF2HEX_DEBUG
    for (unsigned si = 0, se = Symbols.size(); si != se; ++si) {
        errs() << '\n' << "/*" << Symbols[si].first << " " << Symbols[si].second << "*/\n";
    }
#endif

// Make a list of all the relocations for this section.
std::vector<RelocationRef> Rels;

// Sort relocations by address.
std::sort(Rels.begin(), Rels.end(), RelocAddressLess);

StringRef SegmentName = "";
if (const MachOObjectFile *MachO =
    dyn_cast<const MachOObjectFile>(Obj)) {
    DataRefImpl DR = Section.getRawDataRefImpl();
    SegmentName = MachO->getSectionFinalSegmentName(DR);
}
StringRef name;
if (error(Section.getName(name))) break;
outs() << "/*" << "Disassembly of section ";
if (!SegmentName.empty())
    outs() << SegmentName << ",";
outs() << name << ':' << "*/";

// If the section has no symbols just insert a dummy one and disassemble
// the whole section.
if (Symbols.empty())
    Symbols.push_back(std::make_pair(0, name));

SmallString<40> Comments;
raw_svector_ostream CommentStream(Comments);

StringRef BytesStr;
if (error(Section.getContents(BytesStr))) break;
ArrayRef<uint8_t> Bytes(reinterpret_cast<const uint8_t *>(BytesStr.data()),
    BytesStr.size());

uint64_t Size;
uint64_t Index;
SectSize = Section.getSize();

std::vector<RelocationRef>::const_iterator rel_cur = Rels.begin();
std::vector<RelocationRef>::const_iterator rel_end = Rels.end();
// Disassemble symbol by symbol.
for (unsigned si = 0, se = Symbols.size(); si != se; ++si) {
    uint64_t Start = Symbols[si].first;
    uint64_t End;
    // The end is either the size of the section or the beginning of the next
    // symbol.
    if (si == se - 1)
        End = SectSize;
    // Make sure this symbol takes up space.
    else if (Symbols[si + 1].first != Start)
        End = Symbols[si + 1].first - 1;
    else {
        outs() << '\n' << "/*" << Symbols[si].second << "*/\n";
    }
}
```

```

        continue;
    }

    outs() << '\n' << "/*" << Symbols[si].second << "*/\n";
#ifdef DLINK
    uint16_t funIndex = 0;
    if (LinkSo) {
        // correctDynFunIndex
        funIndex = cpu0DynFunIndex.correctDynFunIndex(Symbols[si].second.data());
    }
#endif

#ifndef NDEBUG
    raw_ostream &DebugOut = DebugFlag ? dbggs() : nulls();
#else
    raw_ostream &DebugOut = nulls();
#endif

    for (Index = Start; Index < End; Index += Size) {
        MCInst Inst;

#ifdef DLINK
#ifdef ELF2HEX_DEBUG
        errs() << "funIndex: " << funIndex << "Index: " << Index << "Size: " << Size << "\n";
#endif
#endif
        if (LinkSo && funIndex && Index == Start) {
            outs() << format("/%8" PRIx64 "*/\t", SectionAddr + Index);
            outs() << "01 6b " << format("%02" PRIx64, (funIndex*4+16) & 0xff00)
                << format(" %02" PRIx64, (funIndex*4+16) & 0x00ff);
            outs() << "                                /* ld\t$t9, "
                << funIndex*4+16 << "($gp)\n";
        }
        else
#ifdef DLINK
#endif
        {
            if (DisAsm->getInstruction(Inst, Size, Bytes.slice(Index),
                                     SectionAddr + Index, DebugOut,
                                     CommentStream)) {
                outs() << format("/%8" PRIx64 "*/", SectionAddr + Index);
                if (!NoShowRawInsn) {
                    outs() << "\t";
                    dumpBytes(Bytes.slice(Index, Size), outs());
                }
                outs() << "/*";
                IP->printInst(&Inst, outs(), "", *STI);
                outs() << CommentStream.str();
                outs() << "*/";
                Comments.clear();
                outs() << "\n";
            } else {
                errs() << ToolName << ": warning: invalid instruction encoding\n";
                if (Size == 0)
                    Size = 1; // skip illegible bytes
            }
        }
    }

    // outs() << "Size = " << Size << "Index = " << Index << "lastDumpAddr = "
    // << lastDumpAddr << "\n"; // debug

```

```
// Print relocation for instruction.
while (rel_cur != rel_end) {
    bool hidden = getHidden(*rel_cur);
    uint64_t addr = rel_cur->getOffset();
    SmallString<16> name;
    SmallString<32> val;

    // If this relocation is hidden, skip it.
    if (hidden) goto skip_print_rel;

    // Stop when rel_cur's address is past the current instruction.
    if (addr >= Index + Size) break;
    rel_cur->getTypeName(name);
    if (error(getRelocationValueString(*rel_cur, val)))
        goto skip_print_rel;

    outs() << format("\t\t\t\t/*%8" PRIx64 " ": ", SectionAddr + addr) << name
           << "\t" << val << "*/\n";

    skip_print_rel:
        ++rel_cur;
    }
}
#ifdef ELF2HEX_DEBUG
    errs() << format("SectionAddr + Index = %8" PRIx64 "\n", SectionAddr + Index);
    errs() << format("lastDumpAddr %8" PRIx64 "\n", lastDumpAddr);
#endif
}
// In section .plt or .text, the Contents.size() maybe < (SectionAddr + Index)
if (Contents.size() < (SectionAddr + Index))
    lastDumpAddr = SectionAddr + Index;
else
    lastDumpAddr = SectionAddr + Contents.size();
}
}

static uint64_t SectionOffset(const ObjectFile *o,StringRef secName) {
    std::error_code ec;

    for (const SectionRef &Section : o->sections()) {
        if (error(ec)) return 0;
        StringRef Name;
        StringRef Contents;
        uint64_t BaseAddr;
        bool BSS;
        if (error(Section.getName(Name))) return 0;
        if (error(Section.getContents(Contents))) return 0;
        BaseAddr = Section.getAddress();
        BSS = Section.isBSS();

        if (Name == secName)
            return BaseAddr;
    }
    return 0;
}

static void PrintBootSection(uint64_t pltOffset, bool isLittleEndian) {
    uint64_t offset = pltOffset - 4;
```

```

if (isLittleEndian) {
    outs() << "/*      0:*/      ";
    outs() << format("%02" PRIx64 " ", (offset & 0xff));
    outs() << format("%02" PRIx64 " ", (offset & 0xff00) >> 8);
    outs() << " 00 36";
    outs() << "                                /*      jmp      0x";
    outs() << format("%02" PRIx64 "%02" PRIx64 " */\n", (offset & 0xff00) >> 8,
                    (offset & 0xff));
    outs() <<
        "/*      4:*/      04 00 00 36                                /*      jmp      4 */\n";
    outs() << "/*      8:*/      ";
    outs() << format("%02" PRIx64 " ", (offset & 0xff));
    outs() << format("%02" PRIx64 " ", (offset & 0xff00) >> 8);
    outs() << " 00 36";
    outs() << "                                /*      jmp      0x";
    outs() << format("%02" PRIx64 "%02" PRIx64 " */\n", (offset & 0xff00) >> 8,
                    (offset & 0xff));
    outs() <<
        "/*      c:*/      fc ff ff 36                                /*      jmp      -4 */\n";
}
else {
    outs() << "/*      0:*/      36 00 ";
    outs() << format("%02" PRIx64 " ", (offset & 0xff00) >> 8);
    outs() << format("%02" PRIx64 " ", (offset & 0xff));
    outs() << "                                /*      jmp      0x";
    outs() << format("%02" PRIx64 "%02" PRIx64 " */\n", (offset & 0xff00) >> 8,
                    (offset & 0xff));
    outs() <<
        "/*      4:*/      36 00 00 04                                /*      jmp      4 */\n";
    outs() << "/*      8:*/      36 00 ";
    outs() << format("%02" PRIx64 " ", (offset & 0xff00) >> 8);
    outs() << format("%02" PRIx64 " ", (offset & 0xff));
    outs() << "                                /*      jmp      0x";
    outs() << format("%02" PRIx64 "%02" PRIx64 " */\n", (offset & 0xff00) >> 8,
                    (offset & 0xff));
    outs() <<
        "/*      c:*/      36 ff ff fc                                /*      jmp      -4 */\n";
}
}

#if 0
// Create by ref PrintSymbolTable()
static void FillJTI(const ObjectFile *o) {
    for (const SymbolRef &Symbol : o->symbols()) {
       StringRef JTIBlockName;
        uint64_t BBAddr[0x10000];
        StringRef Name;
        uint64_t Address;
        SymbolRef::Type Type;
        uint64_t Size;
        uint32_t Flags = Symbol.getFlags();
        section_iterator Section = o->section_end();
        if (error(Section->getName(SectionName)))
            SectionName = "";
        if (SectionName != ".rodata") continue;
        if (error(Symbol.getName(Name)))
            continue;
        // For example: Name=JTI8_0 => JTIBlockName=BB8 (rule hit until _)
    }
}

```

```
    if (strncmp(Name.c_str(), "JTI", strlen("JTI") == 0) {
        int i = 0;
        for (i = strlen("JTI"); Name[i] != '_'; i++);
        if (i > strlen("JTI"))
            JTIBlockName = "BB" + Name.substr(strlen("JTI"), i-strlen("JTI"));
    }
    // Then get all BB_* address.
    Address = Section.getAddress();
    if (error(Symbol.getSection(Section)))
        continue;
    if (Section != ".text")
        continue;
}
#endif

static void Elf2Hex(const ObjectFile *o) {
    uint64_t lastDumpAddr = 0;

    const Target *TheTarget = getTarget(o);
    // getTarget() will have already issued a diagnostic if necessary, so
    // just bail here if it failed.
    if (!TheTarget)
        return;

    // Package up features to be passed to target/subtarget
    std::string FeaturesStr;
    if (MAttrs.size()) {
        SubtargetFeatures Features;
        for (unsigned i = 0; i != MAttrs.size(); ++i)
            Features.AddFeature(MAttrs[i]);
        FeaturesStr = Features.getString();
    }

    std::unique_ptr<const MCRegisterInfo> MRI(TheTarget->createMCRegInfo(TripleName));
    if (!MRI) {
        errs() << "error: no register info for target " << TripleName << "\n";
        return;
    }

    // Set up disassembler.
    std::unique_ptr<const MCAsmInfo> AsmInfo(
        TheTarget->createMCAsmInfo(*MRI, TripleName));
    if (!AsmInfo) {
        errs() << "error: no assembly info for target " << TripleName << "\n";
        return;
    }

    std::unique_ptr<const MCSubtargetInfo> STI(
        TheTarget->createMCSubtargetInfo(TripleName, "", FeaturesStr));
    if (!STI) {
        errs() << "error: no subtarget info for target " << TripleName << "\n";
        return;
    }

    std::unique_ptr<const MCInstrInfo> MII(TheTarget->createMCInstrInfo());
    if (!MII) {
        errs() << "error: no instruction info for target " << TripleName << "\n";
        return;
    }
}
```



```

}

std::unique_ptr<const MCObjectFileInfo> MOFI(new MCObjectFileInfo);
MCContext Ctx(AsmInfo.get(), MRI.get(), MOFI.get());

std::unique_ptr<MCDisassembler> DisAsm(
    TheTarget->createMCDisassembler(*STI, Ctx));
if (!DisAsm) {
    errs() << "error: no disassembler for target " << TripleName << "\n";
    return;
}

std::unique_ptr<const MCInstrAnalysis> MIA(
    TheTarget->createMCInstrAnalysis(MII.get()));

int AsmPrinterVariant = AsmInfo->getAssemblerDialect();
std::unique_ptr<MCInstPrinter> IP(TheTarget->createMCInstPrinter(
    Triple(TripleName), AsmPrinterVariant, *AsmInfo, *MII, *MRI));
if (!IP) {
    errs() << "error: no instruction printer for target " << TripleName
        << '\n';
    return;
}

#ifdef ELF2HEX_DEBUG
    uint64_t startAddr = GetSectionHeaderStartAddress(o, "_start");
    errs() << format("_start address:%08" PRIx64 "\n", startAddr);
#endif
#ifdef DLINK
    if (DumpSo) {
        DisassembleSoInHexFormat(o, DisAsm, IP, lastDumpAddr, STI);
        PrintSoDataSections(o, lastDumpAddr, LittleEndian);
    }
    else
#endif
    {
        std::error_code EC;
        uint64_t pltOffset = SectionOffset(o, ".plt");
        PrintBootSection(pltOffset, LittleEndian);
#ifdef DLINK
        if (LinkSo) {
            cpu0DynFunIndex.createPltName(o);
            cpu0DynFunIndex.createStrtab();
            raw_fd_ostream fd_plt_offset("dlconfig/plt_offset", EC,
                sys::fs::F_Text);
            fd_plt_offset << format("%08" PRIx64 " ", pltOffset);
        }
#endif
        lastDumpAddr = 16;
        Fill0s(lastDumpAddr, 0x100);
        lastDumpAddr = 0x100;
        DisassembleObjectInHexFormat(o, DisAsm, IP, lastDumpAddr, STI);
    }
}

```

**exlbt/llvm-objdump/llvm-objdump.cpp**

```
#include "elf2hex.h"

static void DumpObject(const ObjectFile *o) {
    // Avoid other output when using a raw option.
    if (!RawClangAST) {
        outs() << '\n';
        if (ConvertElf2Hex)
            outs() << "/*";
        outs() << o->getFileName()
            << ":tfile format " << o->getFileFormatName() << "\n\n";
        if (ConvertElf2Hex)
            outs() << "*/";
        outs() << "\n\n";
    }

    if (Disassemble)
        DisassembleObject(o, Relocations);
    if (Relocations && !Disassemble)
        PrintRelocations(o);
    if (SectionHeaders)
        PrintSectionHeaders(o);
    if (SectionContents)
        PrintSectionContents(o);
    if (ConvertElf2Hex)
        Elf2Hex(o);
    if (SymbolTable)
        PrintSymbolTable(o);
    if (UnwindInfo)
        PrintUnwindInfo(o);
    if (PrivateHeaders)
        printPrivateFileHeader(o);
    if (ExportsTrie)
        printExportsTrie(o);
    if (Rebase)
        printRebaseTable(o);
    if (Bind)
        printBindTable(o);
    if (LazyBind)
        printLazyBindTable(o);
    if (WeakBind)
        printWeakBindTable(o);
    if (RawClangAST)
        printRawClangAST(o);
    if (PrintFaultMaps)
        printFaultMaps(o);
}

int main(int argc, char **argv) {
    // Print a stack trace if we signal out.
    sys::PrintStackTraceOnErrorSignal();
    PrettyStackTraceProgram X(argc, argv);
    llvm_shutdown_obj Y; // Call llvm_shutdown() on exit.

    // Initialize targets and assembly printers/parsers.
    llvm::InitializeAllTargetInfos();
    llvm::InitializeAllTargetMCs();
}
```

```

llvm::InitializeAllAsmParsers();
llvm::InitializeAllDisassemblers();

// Register the target printer for --version.
cl::AddExtraVersionPrinter(TargetRegistry::printRegisteredTargetsForVersion);

cl::ParseCommandLineOptions(argc, argv, "llvm object file dumper\n");
TripleName = Triple::normalize(TripleName);

ToolName = argv[0];

// Defaults to a.out if no filenames specified.
if (InputFilenames.size() == 0)
    InputFilenames.push_back("a.out");

if (!Disassemble
    && !Relocations
    && !SectionHeaders
    && !SectionContents
    && !ConvertElf2Hex
    && !SymbolTable
    && !UnwindInfo
    && !PrivateHeaders
    && !ExportsTrie
    && !Rebase
    && !Bind
    && !LazyBind
    && !WeakBind
    && !RawClangAST
    && !(UniversalHeaders && MachOOpt)
    && !(ArchiveHeaders && MachOOpt)
    && !(IndirectSymbols && MachOOpt)
    && !(DataInCode && MachOOpt)
    && !(LinkOptHints && MachOOpt)
    && !(InfoPlist && MachOOpt)
    && !(DylibsUsed && MachOOpt)
    && !(DylibId && MachOOpt)
    && !(ObjcMetaData && MachOOpt)
    && !(DumpSections.size() != 0 && MachOOpt)
    && !PrintFaultMaps) {
    cl::PrintHelpMessage();
    return 2;
}

std::for_each(InputFilenames.begin(), InputFilenames.end(),
              DumpInput);

return ReturnValue;
}

```

The code included in “`#ifdef DLINK`” are for dynamic linker support. The `elf2hex.h` supports both endian dump.

## 2.2 Create Cpu0 backend under LLD

### 2.2.1 Setup Cpu0 backend under lld

LLD project is under development and can be compiled only with c++11 standard (C++2011 year announced standard). For iMac, our software is OS X version 10.9.1 and Xcode version 5.0.2. For old iMac software version, you can install VM (such as Virtual Box) and build lld as Linux platform. Please download lld from llvm web<sup>3</sup> and put lld source code on {llvm-src}/tools/lld just like we download llvm and clang as shown in Appendix A of book “Tutorial: Creating an LLVM Backend for the Cpu0 Architecture” as follows.

```
1-160-136-173:tools Jonathan$ pwd
/Users/Jonathan/llvm/test/src/tools
1-160-136-173:tools Jonathan$ ls
...
lld                  llvm-config          llvm-extract          llvm-nm                llvm-stress          obj2yaml
```

Next, setup Cpu0 backend as follows,

```
1-160-136-173:lld Jonathan$ pwd
/Users/Jonathan/llvm/test/src/tools/lld
1-160-136-173:lld Jonathan$ cp -rf ~/test/lbt/exlbt/lld/* .
```

Finally, update llvm-objdump to support converting ELF file to Hex file as follows,

```
1-160-136-173:llvm-objdump Jonathan$ pwd
/Users/Jonathan/llvm/test/src/tools/llvm-objdump
1-160-136-173:llvm-objdump Jonathan$ cp -rf ~/test/lbt/exlbt/llvm-objdump/* .
```

Now, build lld with Cpu0 backend as follows,

```
1-160-136-173:cmake_debug_build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++ -
DCMAKE_C_COMPILER=clang -DCMAKE_CXX_FLAGS=-std=c++11 -DCMAKE_BUILD_TYPE=Debug
-G "Xcode" ../src
...
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/Jonathan/llvm/test/cmake_debug_build
```

If using VM (guest machine is Linux) or Linux, build as follows,

```
[Gamma@localhost cmake_debug_build]$ cmake -DCMAKE_CXX_COMPILER=g++ -
DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_FLAGS=-std=c++11 -DCMAKE_BUILD_TYPE=Debug
-G "Unix Makefiles" ../src
...
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cschen/llvm/test/cmake_debug_build
```

### 2.2.2 Cpu0 backend source code

The code added on lld to support Cpu0 ELF as follows,

---

<sup>3</sup> <http://llvm.org/releases/download.html#3.5>

**exlbt/lld/include/Core/Reference.h**

```
// Which architecture the kind value is for.
enum class KindArch { ..., Cpu0 };
```

**exlbt/lld/include/ReaderWriter/Reference.h**

```
std::unique_ptr<ELFLinkingContext> createCpu0LinkingContext(llvm::Triple);
```

**exlbt/lld/lib/Driver/CMakeLists.txt**

```
add_llvm_library(lldDriver
    ...
    lldCpu0ELFTarget
)
```

**exlbt/lld/lib/Driver/GnuLdDriver.cpp**

```
std::unique_ptr<ELFLinkingContext>
GnuLdDriver::createELFLinkingContext(llvm::Triple triple) {
    ...
    if ((p = elf::createCpu0LinkingContext(triple))) return p;
    ...
}
```

**exlbt/lld/lib/ReaderWriter/ELF/CMakeLists.txt**

```
add_subdirectory(Cpu0)
```

**exlbt/lld/lib/ReaderWriter/ELF/Atoms.h**

```
class SimpleELFDefinedAtom : public SimpleDefinedAtom {
    ...
    void addReferenceELF_Cpu0(Reference::KindValue relocType, uint64_t off,
                             const Atom *t, Reference::Addend a) {
        addReferenceELF(Reference::KindArch::Cpu0, relocType, off, t, a);
    }
}
```

**exlbt/lld/ELFFile.h**

```
template <class ELFT> Reference::KindArch ELFFile<ELFT>::kindArch() {
    switch (_objFile->getHeader()->e_machine) {
        ...
        case llvm::ELF::EM_CPU0:
            return Reference::KindArch::Cpu0;
        case llvm::ELF::EM_CPU0_LE:
```

```
    return Reference::KindArch::Cpu0el;
    ...
}
```

### exlbt/lld/ELFLinkingContext.cpp

```
uint16_t ELFLinkingContext::getOutputMachine() const {
    switch (getTriple().getArch()) {
        ...
        case llvm::Triple::cpu0:
            return llvm::ELF::EM_CPU0;
        case llvm::Triple::cpu0el:
            return llvm::ELF::EM_CPU0_LE;
        ...
    }
}
...
```

### exlbt/lld/lib/ReaderWriter/ELF/ELFFile.h

```
std::unique_ptr<ELFLinkingContext>
ELFLinkingContext::create(llvm::Triple triple) {
    switch (triple.getArch()) {
        ...
        case llvm::ELF::EM_CPU0:
            return Reference::KindArch::Cpu0;
    }
}
```

### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/CMakeLists.txt

```
add_llvm_library(lldCpu0ELFTarget
    Cpu0ExecutableWriter.cpp
    Cpu0LinkingContext.cpp
    Cpu0TargetHandler.cpp
    Cpu0RelocationHandler.cpp
    Cpu0RelocationPass.cpp
    Cpu0SectionChunks.cpp
    LINK_LIBS
        lldELF
        lldReaderWriter
        lldCore
        LLVMObject
        LLVMSupport
)

include_directories(.
```

### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0DynamicLibraryWriter.h

```
//====- lib/ReaderWriter/ELF/Cpu0/Cpu0DynamicLibraryWriter.h -=====//
//
```

```
//
//                                     The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
#ifndef CPU0_DYNAMIC_LIBRARY_WRITER_H
#define CPU0_DYNAMIC_LIBRARY_WRITER_H

#include "DynamicLibraryWriter.h"
#include "Cpu0LinkingContext.h"
#include "Cpu0TargetHandler.h"

namespace lld {
namespace elf {

template <class ELFT>
class Cpu0DynamicLibraryWriter : public DynamicLibraryWriter<ELFT> {
public:
    Cpu0DynamicLibraryWriter(Cpu0LinkingContext &ctx,
                             Cpu0TargetLayout<ELFT> &layout);

protected:
    // Add any runtime files and their atoms to the output
    void createImplicitFiles(std::vector<std::unique_ptr<File>> &) override;
};

} // namespace elf
} // namespace lld

using namespace lld::elf;

template <class ELFT> Cpu0DynamicLibraryWriter<ELFT>::Cpu0DynamicLibraryWriter(
    Cpu0LinkingContext &ctx, Cpu0TargetLayout<ELFT> &layout)
    : DynamicLibraryWriter<ELFT>(ctx, layout) {}

template <class ELFT> void Cpu0DynamicLibraryWriter<ELFT>::createImplicitFiles(
    std::vector<std::unique_ptr<File>> &result) {
    DynamicLibraryWriter<ELFT>::createImplicitFiles(result);
    auto gotFile = llvm::make_unique<SimpleFile>("GOTFile");
    gotFile->addAtom(*new (gotFile->allocator()) GlobalOffsetTableAtom(*gotFile));
    gotFile->addAtom(*new (gotFile->allocator()) DynamicAtom(*gotFile));
    result.push_back(std::move(gotFile));
}

#endif
```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0ExecutableWriter.h

```
//====- lib/ReaderWriter/ELF/Cpu0/Cpu0ExecutableWriter.h -=====//
//
//                                     The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
```

```
#ifndef CPU0_EXECUTABLE_WRITER_H
#define CPU0_EXECUTABLE_WRITER_H

#include "ExecutableWriter.h"
#include "Cpu0LinkingContext.h"
#include "Cpu0TargetHandler.h"

namespace lld {
namespace elf {

template <class ELFT>
class Cpu0ExecutableWriter : public ExecutableWriter<ELFT> {
public:
    Cpu0ExecutableWriter(Cpu0LinkingContext &ctx, Cpu0TargetLayout<ELFT> &layout);

protected:
    // Add any runtime files and their atoms to the output
    void
    createImplicitFiles(std::vector<std::unique_ptr<File>> &result) override {
        ExecutableWriter<ELFT>::createImplicitFiles(result);
        auto gotFile = llvm::make_unique<SimpleFile>("GOTFile");
        gotFile->addAtom(*new (gotFile->allocator())
                        GlobalOffsetTableAtom(*gotFile));
        if (this->_ctx.isDynamic())
            gotFile->addAtom(*new (gotFile->allocator()) DynamicAtom(*gotFile));
        result.push_back(std::move(gotFile));
    }

    void buildDynamicSymbolTable(const File &file) override {
        for (auto sec : this->_layout.sections()) {
            if (auto section = dyn_cast<AtomSection<ELFT>>(sec)) {
                for (const auto &atom : section->atoms()) {
                    if (_targetLayout.getGOTSection().hasGlobalGOTEntry(atom->_atom)) {
                        this->_dynamicSymbolTable->addSymbol(atom->_atom, section->ordinal(),
                                                            atom->_virtualAddr, atom);
                    }
                }
            }
        }
        ExecutableWriter<ELFT>::buildDynamicSymbolTable(file);
    }

    Cpu0TargetLayout<ELFT> &_targetLayout;
};

} // namespace elf
} // namespace lld

#endif
```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0ExecutableWriter.cpp

```
#include "Cpu0ExecutableWriter.h"

namespace lld {
namespace elf {
```



```

template <class ELFT>
Cpu0ExecutableWriter<ELFT>::
Cpu0ExecutableWriter(Cpu0LinkingContext &ctx, Cpu0TargetLayout<ELFT> &layout)
    : ExecutableWriter<ELFT>(ctx, layout), _targetLayout(layout) {}

template class Cpu0ExecutableWriter<ELF32BE>;
template class Cpu0ExecutableWriter<ELF32LE>;

}
}

```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.h

```

//===- lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.h -----===//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===-----//

#ifndef LLD_READER_WRITER_ELF_CPU0_CPU0_LINKING_CONTEXT_H
#define LLD_READER_WRITER_ELF_CPU0_CPU0_LINKING_CONTEXT_H

#include "lld/ReaderWriter/ELFLinkingContext.h"
#include "llvm/Object/ELF.h"
#include "llvm/Support/ELF.h"

namespace lld {
namespace elf {

/// \brief Cpu0 internal references.
enum {
    /// \brief The 32 bit index of the relocation in the got this reference refers
    /// to.
    LLD_R_CPU0_GOTREINDEX = 1024,
};

class Cpu0LinkingContext : public ELFLinkingContext {
protected:
    Cpu0LinkingContext(llvm::Triple, std::unique_ptr<TargetHandler>);

public:
    static std::unique_ptr<ELFLinkingContext> create(llvm::Triple);
    int getMachineType() const override { return llvm::ELF::EM_CPU0; }
    Cpu0LinkingContext(llvm::Triple);

    void addPasses(PassManager &) override;
    void registerRelocationNames(Registry &r) override;

    uint64_t getBaseAddress() const override {
        if (_baseAddress == 0)
            return 0x00;
        return _baseAddress;
    }
}

```

```
bool isDynamicRelocation(const Reference &r) const override {
    if (r.kindNamespace() != Reference::KindNamespace::ELF)
        return false;
    assert(r.kindArch() == Reference::KindArch::Cpu0);
    switch (r.kindValue()) {
    case llvm::ELF::R_CPU0_GLOB_DAT:
        return true;
    default:
        return false;
    }
}

bool isPLTRelocation(const Reference &r) const override {
    if (r.kindNamespace() != Reference::KindNamespace::ELF)
        return false;
    assert(r.kindArch() == Reference::KindArch::Cpu0);
    switch (r.kindValue()) {
    case llvm::ELF::R_CPU0_JUMP_SLOT:
    case llvm::ELF::R_CPU0_RELGOT:
        return true;
    default:
        return false;
    }
}

/// \brief Cpu0 has one relative relocations
/// a) for supporting relative relocs - R_CPU0_RELGOT
bool isRelativeReloc(const Reference &r) const override {
    if (r.kindNamespace() != Reference::KindNamespace::ELF)
        return false;
    assert(r.kindArch() == Reference::KindArch::Cpu0);
    switch (r.kindValue()) {
    case llvm::ELF::R_CPU0_RELGOT:
        return true;
    default:
        return false;
    }
}

bool isStaticExecutable() const { return _isStaticExecutable; }
};
} // end namespace elf
} // end namespace lld

#endif
```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.cpp

```
//===- lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.cpp -=====//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===-----
```

```

#include "llvm/Support/Endian.h"
#include "Cpu0LinkingContext.h"
#include "Cpu0RelocationPass.h"
#include "Cpu0TargetHandler.h"

using namespace lld;
using namespace lld::elf;
using namespace llvm::support;

Cpu0LinkingContext::Cpu0LinkingContext(
    llvm::Triple triple, std::unique_ptr<TargetHandler> handler)
    : ELFLinkingContext(triple, std::move(handler)) {}

static std::unique_ptr<TargetHandler> createTarget(llvm::Triple triple,
                                                  Cpu0LinkingContext &ctx) {
    switch (triple.getArch()) {
    case llvm::Triple::cpu0:
        return llvm::make_unique<Cpu0TargetHandler<ELF32BE>>(ctx, big);
    case llvm::Triple::cpu0el:
        return llvm::make_unique<Cpu0TargetHandler<ELF32LE>>(ctx, little);
    default:
        llvm_unreachable("Unhandled arch");
    }
}

Cpu0LinkingContext::Cpu0LinkingContext(llvm::Triple triple)
    : Cpu0LinkingContext(triple, createTarget(triple, *this)) {}

void Cpu0LinkingContext::addPasses(PassManager &pm) {
    auto pass = createCpu0RelocationPass(*this);
    if (pass)
        pm.add(std::move(pass));
    ELFLinkingContext::addPasses(pm);
}

std::unique_ptr<ELFLinkingContext>
elf::createCpu0LinkingContext(llvm::Triple triple) {
    switch (triple.getArch()) {
    case llvm::Triple::cpu0:
        return llvm::make_unique<Cpu0LinkingContext>(triple);
    case llvm::Triple::cpu0el:
        return llvm::make_unique<Cpu0LinkingContext>(triple);
    }
    return nullptr;
}

static const Registry::KindStrings kindStrings[] = {
#define ELF_RELOC(name, value) LLD_KIND_STRING_ENTRY(name),
#include "llvm/Support/ELFRelocs/Cpu0.def"
#undef ELF_RELOC
    LLD_KIND_STRING_ENTRY(LLD_R_CPU0_GOTREINDEX),
    LLD_KIND_STRING_END
};

void Cpu0LinkingContext::registerRelocationNames(Registry &registry) {
    registry.addKindTable(Reference::KindNamespace::ELF,
                          Reference::KindArch::Cpu0, kindStrings);
}

```

**exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.h**

```
//==-- lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.h ==-//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//==-----//

#ifndef CPU0_RELOCATION_HANDLER_H
#define CPU0_RELOCATION_HANDLER_H

#include "Cpu0TargetHandler.h"
#include "lld/ReaderWriter/ELFLinkingContext.h"
#include "llvm/Support/Endian.h"

using namespace llvm::support;

namespace lld {
namespace elf {
template<typename ELFT> class Cpu0TargetLayout;
template<typename ELFT> class Cpu0TargetHandler;

template <class ELFT>
class Cpu0TargetRelocationHandler final : public TargetRelocationHandler {
public:
    Cpu0TargetRelocationHandler(Cpu0LinkingContext &ctx,
                                Cpu0TargetLayout<ELFT> &layout,
                                Cpu0TargetHandler<ELFT> &targetHandler,
                                endianness endian)
        : _tlsSize(0), _ctx(ctx), _layout(layout), _targetHandler(targetHandler),
          _endian(endian)
    {}

    std::error_code applyRelocation(ELFWriter &, llvm::FileOutputBuffer &,
                                    const AtomLayout &,
                                    const Reference &) const override;

    endianness endian() const { return _endian; };

private:
    // Cached size of the TLS segment.
    mutable uint64_t _tlsSize;
    Cpu0LinkingContext &_ctx;
    Cpu0TargetLayout<ELFT> &_layout;
    Cpu0TargetHandler<ELFT> &_targetHandler;
    endianness _endian;
};

} // end namespace elf
} // end namespace lld

#endif // CPU0_RELOCATION_HANDLER_H
```

## exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.cpp

```
//==-- lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.cpp -----==//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//

#include "Cpu0LinkingContext.h"
#include "Cpu0TargetHandler.h"
#include "llvm/Support/Endian.h"

using namespace lld;
using namespace lld::elf;
using namespace llvm::support;
using namespace llvm::support::endian;

//template <class ELFT>
// \brief R_CPU0_HI16 - word64: (S + A) >> 16
void relocHI16(uint8_t *location, uint64_t P, uint64_t S, int64_t A, endianness endian) {
    // Don't know why A, ref.addend(), = 9
    uint32_t result = (uint32_t)(S >> 16);
    // endian::write<uint32_t, ELFT::TargetEndianness, unaligned>(location, result);
    if (endian == big) {
        write32be(location, result | read32be(location));
    }
    else if (endian == little) {
        write32le(location, result | read32le(location));
    }
    else {
        llvm_unreachable("Unexpected size");
    }
}

void relocLO16(uint8_t *location, uint64_t P, uint64_t S, uint64_t A, endianness endian) {
    // Don't know why A, ref.addend(), = 9
    uint32_t result = (uint32_t)(S & 0x0000ffff);
    if (endian == big) {
        write32be(location, result | read32be(location));
    }
    else if (endian == little) {
        write32le(location, result | read32le(location));
    }
    else {
        llvm_unreachable("Unexpected size");
    }
}

#if 0 // Not support yet
// \brief R_CPU0_GOT16 - word32: S
void relocGOT16(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
    uint32_t result = (uint32_t)(S);
    write32be(location, result | read32be(location));
}
#endif
#endif
```

```
///  
void relocPC24(uint8_t *location, uint64_t P, uint64_t S, int64_t A, Endianness endian) {  
    uint32_t result = (uint32_t) (S - P);  
    if (endian == big) {  
        uint32_t machinecode = (uint32_t) *  
            reinterpret_cast<llvm::support::ubig32_t *>(location);  
        uint32_t opcode = (machinecode & 0xff000000);  
        uint32_t offset = (machinecode & 0x00ffffff);  
        write32be(location, ((result + offset) & 0x00ffffff) | opcode);  
    }  
    else if (endian == little) {  
        uint32_t machinecode = (uint32_t) *  
            reinterpret_cast<llvm::support::ulittle32_t *>(location);  
        uint32_t opcode = (machinecode & 0xff000000);  
        uint32_t offset = (machinecode & 0x00ffffff);  
        write32le(location, ((result + offset) & 0x00ffffff) | opcode);  
    }  
    else {  
        llvm_unreachable("Unexpected size");  
    }  
}  
  
///  
void reloc32(uint8_t *location, uint64_t P, uint64_t S, int64_t A, Endianness endian) {  
    int32_t result = (int32_t) (S);  
    if (endian == big) {  
        write32be(location, result | read32be(location));  
    }  
    else if (endian == little) {  
        write32le(location, result | read32be(location));  
    }  
    else {  
        llvm_unreachable("Unexpected size");  
    }  
    // TODO: Make sure that the result zero extends to the 64bit value.  
}  
  
template <class ELFT> std::error_code Cpu0TargetRelocationHandler<ELFT>::applyRelocation(  
    ELFWriter &writer, llvm::FileOutputBuffer &buf, const AtomLayout &atom,  
    const Reference &ref) const {  
#ifdef DLINKER  
    static bool firstTime = true;  
    std::string soName("libfoobar.cpu0.so");  
    int idx = 0;  
    if (firstTime) {  
        if (this->_ctx.getOutputELFType() == llvm::ELF::ET_DYN) {  
            _targetHandler.soPlt.createFunAddr(_layout, buf);  
        }  
        else if (this->_ctx.getOutputELFType() == llvm::ELF::ET_EXEC &&  
            !this->_ctx.isStaticExecutable()) {  
            _targetHandler.soPlt.createFunAddr(_layout, buf);  
        }  
        firstTime = false;  
    }  
#endif // DLINKER  
    uint8_t *atomContent = buf.getBufferStart() + atom._fileOffset;  
    uint8_t *loc = atomContent + ref.offsetInAtom();  
    uint64_t target = writer.addressOfAtom(ref.target());
```

```

    uint64_t reloc = atom._virtualAddr + ref.offsetInAtom();
#ifdef DLINKER
    #if 1 // For case R_CPU0_GOT16:
    // auto gotAtomIter = _context.getTargetHandler<Cpu0ELFType>().targetLayout().
    // findAbsoluteAtom("_GLOBAL_OFFSET_TABLE_");
    // uint64_t globalOffsetTableAddress = writer.addressOfAtom(*gotAtomIter);
    // .got.plt start from _GLOBAL_OFFSET_TABLE_
    auto gotpltSection = _layout.findOutputSection(".got.plt");
    uint64_t gotPltFileOffset;
    if (gotpltSection)
        gotPltFileOffset = gotpltSection->fileOffset();
    else
        gotPltFileOffset = 0;
    #endif
#endif // DLINKER

    if (ref.kindNamespace() != Reference::KindNamespace::ELF)
        return std::error_code();
    assert(ref.kindArch() == Reference::KindArch::Cpu0);
    switch (ref.kindValue()) {
    case R_CPU0_NONE:
        break;
    case R_CPU0_HI16:
        relocHI16(loc, reloc, target, ref.addend(), _endian);
        break;
    case R_CPU0_LO16:
        relocLO16(loc, reloc, target, ref.addend(), _endian);
        break;
    #if 0 // Not support yet
    case R_CPU0_GOT16:
    #if 1
        idx = _targetHandler.soPlt.getDynFunIndexByTargetAddr(targetVAddress);
        relocGOT16(location, reloc, idx, ref.addend(), _endian);
    #else
        relocGOT16(location, reloc, (target - gotPltFileOffset),
                    ref.addend());
    #endif
    #endif
        break;
    #endif
    case R_CPU0_PC24:
        relocPC24(loc, reloc, target, ref.addend(), _endian);
        break;
#ifdef DLINKER
    case R_CPU0_CALL16:
        // offset at _GLOBAL_OFFSET_TABLE_ and $gp point to _GLOBAL_OFFSET_TABLE_.
        idx = _targetHandler.soPlt.getDynFunIndexByTargetAddr(target);
        reloc32(loc, reloc, idx*0x04+16, ref.addend(), _endian);
        break;
#endif // DLINKER
    case R_CPU0_32:
        reloc32(loc, reloc, target, ref.addend(), _endian);
        break;
    // Runtime only relocations. Ignore here.
    case R_CPU0_JUMP_SLOT:
    case R_CPU0_GLOB_DAT:
        break;
    default:
        return make_unhandled_reloc_error();
}

```

```
    }

    return std::error_code();
}

namespace lld {
namespace elf {

template class Cpu0TargetRelocationHandler<ELF32BE>;
template class Cpu0TargetRelocationHandler<ELF32LE>;

} // end namespace elf
} // end namespace lld
```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.h

```
//==-- lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.h -----//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
///
/// \file
/// \brief Declares the relocation processing pass for x86-64. This includes
///        GOT and PLT entries, TLS, COPY, and ifunc.
///
//=====//

#ifndef LLD_READER_WRITER_ELF_CPU0_CPU0_RELOCATION_PASS_H
#define LLD_READER_WRITER_ELF_CPU0_CPU0_RELOCATION_PASS_H

#include <memory>
#include "Cpu0TargetHandler.h"

namespace lld {
class Pass;
namespace elf {
class Cpu0LinkingContext;

/// \brief Create x86-64 relocation pass for the given linking context.
std::unique_ptr<Pass>
createCpu0RelocationPass(const Cpu0LinkingContext &);
}
}

#endif
```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.cpp

```
//==-- lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.cpp -----//
//
//                               The LLVM Linker
```



```
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
//
// \file
// \brief Defines the relocation processing pass for cpu0. This includes
// GOT and PLT entries, TLS, COPY, and ifunc.
//
// This also includes additional behavior that gnu-ld and gold implement but
// which is not specified anywhere.
//
//=====//

#include "Cpu0RelocationPass.h"
#include "Atoms.h"
#include "Cpu0LinkingContext.h"
#include "lld/Core/Simple.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/Support/Endian.h"

using namespace lld;
using namespace lld::elf;
using namespace llvm::ELF;
using namespace llvm::support;

extern endianness gEndian; // set in constructor of Cpu0TargetHandler.cpp

// .plt value (entry 0), big endian
const uint8_t cpu0BootAtomContent[16] = {
    0x36, 0xff, 0xff, 0xfc, // jmp start
    0x36, 0x00, 0x00, 0x04, // jmp 4
    0x36, 0xff, 0xff, 0xfc, // jmp ISR
    0x36, 0xff, 0xff, 0xfc // jmp -4
};
// little endian
const uint8_t cpu0elBootAtomContent[16] = {
    0xfc, 0xff, 0xff, 0x36, // jmp start
    0x04, 0x00, 0x00, 0x36, // jmp 4
    0xfc, 0xff, 0xff, 0x36, // jmp ISR
    0xfc, 0xff, 0xff, 0x36 // jmp -4
};

// .got values
const uint8_t cpu0GotAtomContent[16] = { 0 };

#ifdef DLINKER
// .plt value (entry 0), big endian
const uint8_t cpu0Plt0AtomContent[16] = {
    0x02, 0xeb, 0x00, 0x04, // st $lr, $zero, reloc-index ($gp)
    0x02, 0xcb, 0x00, 0x08, // st $fp, $zero, reloc-index ($gp)
    0x02, 0xdb, 0x00, 0x0c, // st $sp, $zero, reloc-index ($gp)
    0x36, 0xff, 0xff, 0xfc // jmp dynamic_linker
};
// little endian
const uint8_t cpu0elPlt0AtomContent[16] = {
```

```
    0x04, 0x00, 0xeb, 0x02, // st $lr, $zero, reloc-index ($gp)
    0x08, 0x00, 0xcb, 0x02, // st $fp, $zero, reloc-index ($gp)
    0x0c, 0x00, 0xdb, 0x02, // st $sp, $zero, reloc-index ($gp)
    0xfc, 0xff, 0xff, 0x36 // jmp dynamic_linker
};

// .plt values (other entries), big endian
const uint8_t cpu0PltAtomContent[16] = {
    0x01, 0x6b, 0x00, 0x10, // ld $t9, 0x10($gp) (0x10($gp) point to plt0
    0x3c, 0x60, 0x00, 0x00, // ret $t9 // jump to Cpu0.Stub
    0x00, 0x00, 0x00, 0x00, // nop
    0x00, 0x00, 0x00, 0x00 // nop
};
// little endian
const uint8_t cpu0elPltAtomContent[16] = {
    0x10, 0x00, 0x6b, 0x01, // ld $t9, 0x10($gp) (0x10($gp) point to plt0
    0x00, 0x00, 0x60, 0x3c, // ret $t9 // jump to Cpu0.Stub
    0x00, 0x00, 0x00, 0x00, // nop
    0x00, 0x00, 0x00, 0x00 // nop
};

// TLS GD Entry
static const uint8_t cpu0GotTlsGdAtomContent[] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};
#endif // DLINKER

/// boot record
class Cpu0BootAtom : public PLT0Atom {
public:
    Cpu0BootAtom(const File &f) : PLT0Atom(f) {
#ifdef NDEBBUG
        _name = ".PLT0";
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        if (gEndian == big)
            return ArrayRef<uint8_t>(cpu0BootAtomContent, 16);
        else
            return ArrayRef<uint8_t>(cpu0elBootAtomContent, 16);
    }
};

namespace {
/// \brief Atoms that are used by Cpu0 dynamic linking
class Cpu0GOTAtom : public GOTAtom {
public:
    Cpu0GOTAtom(const File &f,StringRef secName) : GOTAtom(f, secName) {}

    ArrayRef<uint8_t> rawContent() const override {
        return ArrayRef<uint8_t>(cpu0GotAtomContent, 16);
    }
};

#ifdef DLINKER
/// \brief Cpu0 GOT TLS GD entry.
class GOTTLSGdAtom : public Cpu0GOTAtom {
```

```

public:
    GOTTLSGdAtom(const File &f,StringRef secName) : Cpu0GOTAtom(f, secName) {}

    ArrayRef<uint8_t> rawContent() const override {
        return llvm::makeArrayRef(cpu0GotTlsGdAtomContent);
    }
};

class Cpu0PLT0Atom : public PLT0Atom {
public:
    Cpu0PLT0Atom(const File &f) : PLT0Atom(f) {}
    ArrayRef<uint8_t> rawContent() const override {
        if (gEndian == big)
            return ArrayRef<uint8_t>(cpu0Plt0AtomContent, 16);
        else
            return ArrayRef<uint8_t>(cpu0elPlt0AtomContent, 16);
    }
};

class Cpu0PLTAtom : public PLTAtom {
public:
    Cpu0PLTAtom(const File &f,StringRef secName) : PLTAtom(f, secName) {}

    ArrayRef<uint8_t> rawContent() const override {
        if (gEndian == big)
            return ArrayRef<uint8_t>(cpu0PltAtomContent, 16);
        else
            return ArrayRef<uint8_t>(cpu0elPltAtomContent, 16);
    }
};
#endif // DLINKER

class ELFPassFile : public SimpleFile {
public:
    ELFPassFile(const ELFLinkingContext &eti) : SimpleFile("ELFPassFile") {
        setOrdinal(eti.getNextOrdinalAndIncrement());
    }

    llvm::BumpPtrAllocator _alloc;
};

/// \brief CRTP base for handling relocations.
template <class Derived> class RelocationPass : public Pass {
    /// \brief Handle a specific reference.
    void handleReference(const DefinedAtom &atom, const Reference &ref) {
        if (ref.kindNamespace() != Reference::KindNamespace::ELF)
            return;
        assert(ref.kindArch() == Reference::KindArch::Cpu0);
        switch (ref.kindValue()) {
            case R_CPU0_PC24:
                static_cast<Derived *>(this)->handlePlain(ref);
            case R_CPU0_CALL16:
                static_cast<Derived *>(this)->handlePLT32(ref);
                break;
        }
    }
};

protected:

```

```
#ifndef DLINKER
    /// \brief get the PLT entry for a given IFUNC Atom.
    ///
    /// If the entry does not exist. Both the GOT and PLT entry is created.
    const PLTAtom *getIFUNCPLTEntry(const DefinedAtom *da) {
        auto plt = _pltMap.find(da);
        if (plt != _pltMap.end())
            return plt->second;
        auto ga = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
        ga->addReferenceELF_Cpu0(R_CPU0_RELGOT, 0, da, 0);
        auto pa = new (_file._alloc) Cpu0PLTAtom(_file, ".plt");
        pa->addReferenceELF_Cpu0(R_CPU0_PC24, 2, ga, -4);
#ifdef NDEBUG
        ga->_name = "__got_ifunc_";
        ga->_name += da->name();
        pa->_name = "__plt_ifunc_";
        pa->_name += da->name();
#endif
        _gotMap[da] = ga;
        _pltMap[da] = pa;
        _gotVector.push_back(ga);
        _pltVector.push_back(pa);
        return pa;
    }
#endif // DLINKER

    /// \brief Redirect the call to the PLT stub for the target IFUNC.
    ///
    /// This create a PLT and GOT entry for the IFUNC if one does not exist. The
    /// GOT entry and a IRELATIVE relocation to the original target resolver.
    std::error_code handleIFUNC(const Reference &ref) {
#ifdef DLINKER
        auto target = dyn_cast_or_null<const DefinedAtom>(ref.target());
        if (target && target->contentType() == DefinedAtom::typeResolver)
            const_cast<Reference &>(ref).setTarget(getIFUNCPLTEntry(target));
#endif // DLINKER
        return std::error_code();
    }

    /// \brief Create a GOT entry for the TP offset of a TLS atom.
    const GOTAtom *getGOTTPOFF(const Atom *atom) {
        auto got = _gotMap.find(atom);
        if (got == _gotMap.end()) {
            auto g = new (_file._alloc) Cpu0GOTAtom(_file, ".got");
            g->addReferenceELF_Cpu0(R_CPU0_TLS_TPREL32, 0, atom, 0);
#ifdef NDEBUG
            g->_name = "__got_tls_";
            g->_name += atom->name();
#endif
            _gotMap[atom] = g;
            _gotVector.push_back(g);
            return g;
        }
        return got->second;
    }

    /// \brief Create a GOT entry containing 0.
    const GOTAtom *getNullGOT() {
```

```

    if (!_null) {
        _null = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
#ifdef NDEBBUG
        _null->_name = "__got_null";
#endif
    }
    return _null;
}

const GOTAtom *getGOT(const DefinedAtom *da) {
    auto got = _gotMap.find(da);
    if (got == _gotMap.end()) {
        auto g = new (_file._alloc) Cpu0GOTAtom(_file, ".got");
        g->addReferenceELF_Cpu0(R_CPU0_32, 0, da, 0);
#ifdef NDEBBUG
        g->_name = "__got_";
        g->_name += da->name();
#endif
        _gotMap[da] = g;
        _gotVector.push_back(g);
        return g;
    }
    return got->second;
}

public:
    RelocationPass(const ELFLinkingContext &ctx)
        : _file(ctx), _ctx(ctx), _boot(new Cpu0BootAtom(_file)) {}

    /// \brief Do the pass.
    ///
    /// The goal here is to first process each reference individually. Each call
    /// to handleReference may modify the reference itself and/or create new
    /// atoms which must be stored in one of the maps below.
    ///
    /// After all references are handled, the atoms created during that are all
    /// added to mf.
    std::error_code perform(SimpleFile &mf) override {
        ScopedTask task(getDefaultDomain(), "Cpu0 GOT/PLT Pass");
        // Process all references.
        for (const auto &atom : mf.defined())
            for (const auto &ref : *atom)
                handleReference(*atom, *ref);

        // Add all created atoms to the link.
        uint64_t ordinal = 0;
        if (_ctx.getOutputELFType() == llvm::ELF::ET_EXEC) {
            SimpleFile::DefinedAtomRange atomRange = mf.definedAtoms();
            auto it = atomRange.begin();
            bool find = false;
            for (it = atomRange.begin(); it < atomRange.end(); it++) {
                if ((*it)->name() == "start") {
                    find = true;
                    break;
                }
            }
            assert(find && "not found start\n");
            _boot->addReferenceELF_Cpu0(R_CPU0_PC24, 0, *it, -3);
        }
    }

```

```
    find = false;
    for (it = atomRange.begin(); it < atomRange.end(); it++) {
        if ((*it)->name() == "ISR") {
            find = true;
            break;
        }
    }
    assert(find && "not found ISR\n");
    _boot->addReferenceELF_Cpu0(R_CPU0_PC24, 8, *it, -3);
    _boot->setOrdinal(ordinal++);
    mf.addAtom(*_boot);
}
#ifdef DLINKER
if (_plt0) {
    SimpleFile::DefinedAtomRange atomRange = mf.definedAtoms();
    auto it = atomRange.begin();
    bool find = false;
    for (it = atomRange.begin(); it < atomRange.end(); it++) {
        if ((*it)->name() == "_Z14dynamic_linkerv") {
            find = true;
            break;
        }
    }
    assert(find && "Cannot find _Z14dynamic_linkerv()");
    _plt0->addReferenceELF_Cpu0(R_CPU0_PC24, 12, *it, -3);
    _plt0->setOrdinal(ordinal++);
    mf.addAtom(*_plt0);
}
#endif // DLINKER
for (auto &plt : _pltVector) {
    plt->setOrdinal(ordinal++);
    mf.addAtom(*plt);
}
if (_null) {
    _null->setOrdinal(ordinal++);
    mf.addAtom(*_null);
}
if (_plt0) {
    _got0->setOrdinal(ordinal++);
    mf.addAtom(*_got0);
}
for (auto &got : _gotVector) {
    got->setOrdinal(ordinal++);
    mf.addAtom(*got);
}
for (auto &got : _tlsGotVector) {
    got->setOrdinal(ordinal++);
    mf.addAtom(*got);
}
for (auto obj : _objectVector) {
    obj->setOrdinal(ordinal++);
    mf.addAtom(*obj);
}
return std::error_code();
}

protected:
    /// \brief Owner of all the Atoms created by this pass.
```

```

ELFPassFile _file;
const ELFLinkingContext &_ctx;

/// \brief Map Atoms to their GOT entries.
llvm::DenseMap<const Atom *, GOTAtom *> _gotMap;

/// \brief Map Atoms to their PLT entries.
llvm::DenseMap<const Atom *, PLTAtom *> _pltMap;

/// \brief Map Atoms to TLS GD GOT entries.
llvm::DenseMap<const Atom *, GOTAtom *> _gotTLSGdMap;

/// \brief Map Atoms to their Object entries.
llvm::DenseMap<const Atom *, ObjectAtom *> _objectMap;

/// \brief the list of GOT/PLT atoms
std::vector<GOTAtom *> _gotVector;
std::vector<PLTAtom *> _pltVector;
std::vector<ObjectAtom *> _objectVector;
PLT0Atom *_boot;

/// \brief the list of TLS GOT atoms.
std::vector<GOTAtom *> _tlsGotVector;

/// \brief GOT entry that is always 0. Used for undefined weaks.
GOTAtom *_null = nullptr;

/// \brief The got and plt entries for .PLT0. This is used to call into the
/// dynamic linker for symbol resolution.
/// @{
PLT0Atom *_plt0 = nullptr;
GOTAtom *_got0 = nullptr;
/// @}
};

/// This implements the static relocation model. Meaning GOT and PLT entries are
/// not created for references that can be directly resolved. These are
/// converted to a direct relocation. For entries that do require a GOT or PLT
/// entry, that entry is statically bound.
///
/// TLS always assumes module 1 and attempts to remove indirection.
class StaticRelocationPass final
    : public RelocationPass<StaticRelocationPass> {
public:
    StaticRelocationPass(const elf::Cpu0LinkingContext &ctx)
        : RelocationPass(ctx) {}

    std::error_code handlePlain(const Reference &ref) { return handleIFUNC(ref); }

    std::error_code handlePLT32(const Reference &ref) {
        // __tls_get_addr is handled elsewhere.
        if (ref.target() && ref.target()->name() == "__tls_get_addr") {
            const_cast<Reference &>(ref).setKindValue(R_CPU0_NONE);
            return std::error_code();
        }
        // Static code doesn't need PLTs.
        const_cast<Reference &>(ref).setKindValue(R_CPU0_PC24);
        // Handle IFUNC.

```

```
    if (const DefinedAtom *da =
        dyn_cast_or_null<const DefinedAtom>(ref.target()))
        if (da->contentType() == DefinedAtom::typeResolver)
            return handleIFUNC(ref);
    return std::error_code();
}

std::error_code handleGOT(const Reference &ref) {
    if (isa<UndefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getNullGOT());
    else if (const DefinedAtom *da = dyn_cast<const DefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getGOT(da));
    return std::error_code();
}
};

#ifdef DLINKER
class DynamicRelocationPass final
    : public RelocationPass<DynamicRelocationPass> {
public:
    DynamicRelocationPass(const elf::Cpu0LinkingContext &ctx)
        : RelocationPass(ctx) {}

    const PLT0Atom *getPLT0() {
        if (!_plt0)
            return _plt0;
        // Fill in the null entry.
        getNullGOT();
        _plt0 = new (_file._alloc) Cpu0PLT0Atom(_file);
        _got0 = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
#ifdef NDEBUG
        _got0->_name = "__got0";
#endif
        return _plt0;
    }

    const PLTAtom *getPLTEEntry(const Atom *a) {
        auto plt = _pltMap.find(a);
        if (plt != _pltMap.end())
            return plt->second;
        auto ga = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
        ga->addReferenceELF_Cpu0(R_CPU0_JUMP_SLOT, 0, a, 0);
        auto pa = new (_file._alloc) Cpu0PLTAtom(_file, ".plt");
        getPLT0(); // add _PLT0 and _got0

        // Set the starting address of the got entry to the second instruction in
        // the plt entry.
        ga->addReferenceELF_Cpu0(R_CPU0_32, 0, pa, 4);
#ifdef NDEBUG
        ga->_name = "__got_";
        ga->_name += a->name();
        pa->_name = "__plt_";
        pa->_name += a->name();
#endif
        _gotMap[a] = ga;
        _pltMap[a] = pa;
        _gotVector.push_back(ga);
        _pltVector.push_back(pa);
    }
};
```



```

    return pa;
}

std::error_code handlePlain(const Reference &ref) {
    if (!ref.target())
        return std::error_code();
    if (auto sla = dyn_cast<SharedLibraryAtom>(ref.target())) {
        if (sla->type() == SharedLibraryAtom::Type::Code)
            const_cast<Reference &>(ref).setTarget(getPLTEntry(sla));
    } else
        return handleIFUNC(ref);
    return std::error_code();
}

std::error_code handlePLT32(const Reference &ref) {
    // Turn this into a PC24 to the PLT entry.
    // Must move setKindValue() to below.
    // const_cast<Reference &>(ref).setKindValue(R_CPU0_PC24);
    // Handle IFUNC.
    if (const DefinedAtom *da =
        dyn_cast_or_null<const DefinedAtom>(ref.target()))
        if (da->contentType() == DefinedAtom::typeResolver)
            return handleIFUNC(ref);
    if (isa<const SharedLibraryAtom>(ref.target()) ||
        isa<UndefinedAtom>(ref.target())) {
        const_cast<Reference &>(ref).setTarget(getPLTEntry(ref.target()));
        // Turn this into a PC24 to the PLT entry.
#ifdef 1
        const_cast<Reference &>(ref).setKindValue(R_CPU0_PC24);
#endif
    }
    return std::error_code();
}

const GOTAtom *getSharedGOT(const Atom *a) {
    auto got = _gotMap.find(a);
    if (got == _gotMap.end()) {
        auto g = new (_file._alloc) Cpu0GOTAtom(_file, ".got");
        g->addReferenceELF_Cpu0(R_CPU0_GLOB_DAT, 0, a, 0);
#ifdef NDEBUG
        g->_name = "__got_";
        g->_name += a->name();
#endif
        _gotMap[a] = g;
        _gotVector.push_back(g);
        return g;
    }
    return got->second;
}

std::error_code handleGOT(const Reference &ref) {
    if (const DefinedAtom *da = dyn_cast<const DefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getGOT(da));
    // Handle undefined atoms in the same way as shared lib atoms: to be
    // resolved at run time.
    else if (isa<SharedLibraryAtom>(ref.target()) ||
        isa<UndefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getSharedGOT(ref.target()));
}

```

```

        return std::error_code();
    }
};
#endif // DLINKER
} // end anon namespace

std::unique_ptr<Pass>
lld::elf::createCpu0RelocationPass(const Cpu0LinkingContext &ctx) {
    switch (ctx.getOutputELFType()) {
        case llvm::ELF::ET_EXEC:
            // when the output file is execution file: e.g. a.out
#ifdef DLINKER
            if (ctx.isDynamic())
                return llvm::make_unique<DynamicRelocationPass>(ctx);
            else
                return std::unique_ptr<Pass>(new StaticRelocationPass(ctx));
#else
            return llvm::make_unique<StaticRelocationPass>(ctx);
#endif // DLINKER
#ifdef DLINKER
        case llvm::ELF::ET_DYN:
            // when the output file is shared object: e.g. foobar.so
            return llvm::make_unique<DynamicRelocationPass>(ctx);
#endif // DLINKER
        case llvm::ELF::ET_REL:
            return nullptr;
        default:
            llvm_unreachable("Unhandled output file type");
    }
}

```

### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0SectionChunks.h

```

//===- lib/ReaderWriter/ELF/Cpu0/Cpu0SectionChunks.h -----===//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//

#ifndef LLD_READER_WRITER_ELF_CPU0_CPU0_SECTION_CHUNKS_H
#define LLD_READER_WRITER_ELF_CPU0_CPU0_SECTION_CHUNKS_H

#include "TargetLayout.h"

namespace lld {
namespace elf {

template <class ELFT> class Cpu0GOTSection : public AtomSection<ELFT> {
public:
    Cpu0GOTSection(const ELFLinkingContext &ctx);

    bool hasGlobalGOTEntry(const Atom *a) const {
        return _tlsMap.count(a);
    }
};

```

```

    }

    const AtomLayout *appendAtom(const Atom *atom) override;

private:
    /// \brief Map TLS Atoms to their GOT entry index.
    llvm::DenseMap<const Atom *, std::size_t> _tlsMap;
};

} // elf
} // lld

#endif

```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0SectionChunks.cpp

```

//===- lib/ReaderWriter/ELF/Cpu0/Cpu0SectionChunks.cpp -----==//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===-----==//

#include "Cpu0SectionChunks.h"
#include "TargetLayout.h"

namespace lld {
namespace elf {

template <class ELFT> Cpu0GOTSection<ELFT>::Cpu0GOTSection(const ELFLinkingContext &ctx)
    : AtomSection<ELFT>(ctx, ".got", DefinedAtom::typeGOT, DefinedAtom::permRW_,
        TargetLayout<ELFT>::ORDER_GOT) {
    this->_alignment = 16;
}

template <class ELFT> const AtomLayout *Cpu0GOTSection<ELFT>::appendAtom(const Atom *atom) {
    return AtomSection<ELFT>::appendAtom(atom);
}

template class Cpu0GOTSection<ELF32BE>;
template class Cpu0GOTSection<ELF32LE>;
} // elf
} // lld

```

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0TargetHandler.h

```

//===- lib/ReaderWriter/ELF/Cpu0/Cpu0TargetHandler.h -----==//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//

```

```
//=====//

#ifndef LLD_READER_WRITER_ELF_CPU0_CPU0_TARGET_HANDLER_H
#define LLD_READER_WRITER_ELF_CPU0_CPU0_TARGET_HANDLER_H

#include "ELFReader.h"
#include "TargetLayout.h"
#include "Cpu0LinkingContext.h"
#include "Cpu0RelocationHandler.h"
#include "Cpu0SectionChunks.h"
#include "lld/Core/Simple.h"
#include "llvm/Support/Endian.h"

using namespace llvm::support;

#define DLINKER

namespace lld {
namespace elf {

#ifdef DLINKER
template <class ELFT>
class Cpu0SoPlt {
private:
    uint32_t funAddr[100];
    int funAddrSize = 0;
public:
    void createFunAddr(Cpu0TargetLayout<ELFT> &cpu0Layout,
                      llvm::FileOutputBuffer &buf);
    // Return function index, 1: 1st function appear on section .text of .so.
    // 2: 2nd function ...
    // For example: 3 functions _Z2laii, _Z3fooii and _Z3barv. 1: is _Z2laii
    // 2 is _Z3fooii, 3: is _Z3barv.
    int getDynFunIndexByTargetAddr(uint64_t fAddr);
};
#endif

template <class ELFT> class Cpu0TargetLayout : public TargetLayout<ELFT> {
public:
    Cpu0TargetLayout(Cpu0LinkingContext &ctx) : TargetLayout<ELFT>(ctx),
        _gotSection(new (this->_allocator) Cpu0GOTSection<ELFT>(ctx)) {}

    AtomSection<ELFT> *
    createSection(StringRef name, int32_t type,
                  DefinedAtom::ContentPermissions permissions,
                  typename TargetLayout<ELFT>::SectionOrder order) override {
        if (type == DefinedAtom::typeGOT && name == ".got")
            return _gotSection;
        return TargetLayout<ELFT>::createSection(name, type, permissions, order);
    }

    void finalizeOutputSectionLayout() override {
        sortOutputSectionByPriority<ELFT>(".init_array");
        sortOutputSectionByPriority<ELFT>(".fini_array");
    }

    const Cpu0GOTSection<ELFT> &getGOTSection() const { return *_gotSection; }
};
}
```

```

private:
    uint32_t getPriority(StringRef sectionName) const {
        StringRef priority = sectionName.drop_front().rsplit('.').second;
        uint32_t prio;
        if (priority.getAsInteger(10, prio))
            return std::numeric_limits<uint32_t>::max();
        return prio;
    }

    template <typename T> void sortOutputSectionByPriority(StringRef prefix) {
        OutputSection<T> *section = Cpu0TargetLayout<ELFT>::findOutputSection(prefix);
        if (!section)
            return;
        auto sections = section->sections();
        std::sort(sections.begin(), sections.end(),
            [&](Chunk<T> *lhs, Chunk<T> *rhs) {
                Section<T> *lhsSection = dyn_cast<Section<T>>(lhs);
                Section<T> *rhsSection = dyn_cast<Section<T>>(rhs);
                if (!lhsSection || !rhsSection)
                    return false;
                StringRef lhsName = lhsSection->inputSectionName();
                StringRef rhsName = rhsSection->inputSectionName();
                if (!lhsName.startswith(prefix) || !rhsName.startswith(prefix))
                    return false;
                return getPriority(lhsName) < getPriority(rhsName);
            });
    }

private:
    Cpu0GOTSection<ELFT> *_gotSection;
};

template <class ELFT> class Cpu0TargetHandler : public TargetHandler {
public:
    Cpu0TargetHandler(Cpu0LinkingContext &ctx, endianness endian);

    const TargetRelocationHandler &getRelocationHandler() const override {
        return *_relocationHandler;
    }

    std::unique_ptr<Reader> getObjReader() override {
        return llvm::make_unique<ELFReader<ELFFile<ELFT>>>>(_ctx);
    }

    std::unique_ptr<Reader> getDSOReader() override {
        return llvm::make_unique<ELFReader<DynamicFile<ELFT>>>>(_ctx);
    }

    std::unique_ptr<Writer> getWriter() override;

#ifdef DLINKER
    Cpu0SoPlt<ELFT> soPlt;
#endif
protected:
    Cpu0LinkingContext &_ctx;
    std::unique_ptr<Cpu0TargetLayout<ELFT>> _targetLayout;
    std::unique_ptr<Cpu0TargetRelocationHandler<ELFT>> _relocationHandler;
};

```

```
} // end namespace elf
} // end namespace lld
```

```
#endif
```

#### exlbt/lld/Cpu0/lib/ReaderWriter/ELF/Cpu0TargetHandler.cpp

```
//==-- lib/ReaderWriter/ELF/Cpu0/Cpu0TargetHandler.cpp -----==//
//
//                               The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//

#include "Atoms.h"
#include "Cpu0DynamicLibraryWriter.h"
#include "Cpu0ExecutableWriter.h"
#include "Cpu0LinkingContext.h"
#include "Cpu0TargetHandler.h"

endianness gEndian = native;

namespace lld {
namespace elf {

#ifdef DLINKER
template <class ELFT>
void Cpu0SoPlt<ELFT>::createFunAddr(Cpu0TargetLayout<ELFT> &cpu0Layout,
                                   llvm::FileOutputBuffer &buf) {
    auto dynsymSection = cpu0Layout.
        findOutputSection(".dynsym");
    uint64_t dynsymFileOffset, dynsymSize;
    if (dynsymSection) {
        dynsymFileOffset = dynsymSection->fileOffset();
        dynsymSize = dynsymSection->memSize();
        uint8_t *atomContent = buf.getBufferStart() + dynsymFileOffset;
        for (uint64_t i = 4; i < dynsymSize; i += 16) {
            funAddr[funAddrSize] =
                *reinterpret_cast<llvm::support::ubig32_t*>((uint32_t*)
                    (atomContent + i));
            funAddrSize++;
        }
    }
    return;
}

template <class ELFT>
int Cpu0SoPlt<ELFT>::getDynFunIndexByTargetAddr(uint64_t fAddr) {
    for (int i = 0; i < funAddrSize; i++) {
        // Below statement fix the issue that both __tls_get_addr and first
        // function has the same file offset 0 issue.
        if (i < (funAddrSize-1) && funAddr[i] == funAddr[i+1])
            continue;
    }
}
```

```

    if (fAddr == funAddr[i]) {
        return i;
    }
}
return -1;
}

template class Cpu0SoPlt<ELF32BE>;
template class Cpu0SoPlt<ELF32LE>;
#ifdef DLINKER

template <class ELFT>
Cpu0TargetHandler<ELFT>::Cpu0TargetHandler(Cpu0LinkingContext &ctx,
                                           endianness endian)
    : _ctx(ctx), _targetLayout(new Cpu0TargetLayout<ELFT>(ctx)),
      _relocationHandler(new Cpu0TargetRelocationHandler<ELFT>
                        (ctx, *_targetLayout, *this, endian)) {
    gEndian = endian;
}

template <class ELFT>
std::unique_ptr<Writer> Cpu0TargetHandler<ELFT>::getWriter() {
    switch (this->_ctx.getOutputELFType()) {
        case llvm::ELF::ET_EXEC:
            return llvm::make_unique<Cpu0ExecutableWriter<ELFT>>(_ctx, *_targetLayout);
        case llvm::ELF::ET_DYN:
            return llvm::make_unique<Cpu0DynamicLibraryWriter<ELFT>>(_ctx, *_targetLayout);
        case llvm::ELF::ET_REL:
            llvm_unreachable("TODO: support -r mode");
        default:
            llvm_unreachable("unsupported output type");
    }
}

template class Cpu0TargetHandler<ELF32BE>;
template class Cpu0TargetHandler<ELF32LE>;

}
}

```

Above code in Cpu0 lld support both endian for static link and dynamic link. The “#ifdef DLINKER” is for dynamic link support.

## 2.2.3 LLD introduction

In general, linker do the Relocation Records Resolve as Chapter ELF support depicted, and optimization for those cannot finish in compiler stage. One of the optimization opportunities in linker is Dead Code Stripping which is explained in this section.

List the LLD project status as follows,

- The lld project aims to be the built-in linker for clang/llvm. Currently, clang must invoke the system linker to produce executables.
- web site <http://lld.llvm.org/>
- Current Status
  - lld is in its early stages of development.

- It can currently self host on Linux x86-64 with -static.
- How to build
  - cmake -DCMAKE\_CXX\_COMPILER=g++ -DCMAKE\_C\_COMPILER=gcc -DCMAKE\_CXX\_FLAGS=-std=c++11 -DCMAKE\_BUILD\_TYPE=Debug -G “Unix Makefiles” ./src/

This whole book focuses on backend design, and this chapter is same. To help readers understand the lld document, first we list the linking steps from lld web. After that, explain each step with the class of source code and what kind of Cpu0 backend implementation needed in each step. Since some of the following come from our understanding, please read the lld design web document first (only a few pages), <http://lld.llvm.org/design.html>, then reading the following to ensure you agree to our understanding.

### How LLD do the linker job

- LLD structure
  - Internal structure Atom
    - \* Like llvm IR, lld operating and optimize in Atom.
  - ELF reader/writer, Mach-O reader/writer, COFF
    - \* Connect to any specific linker format by implement the concrete Read/Writer.
    - \* e.g. Implement Microsoft link format Reader/Writer => extend lld to support Microsoft link format.
- Atom
  - An atom is an indivisible chunk of code or data.
  - Typically each user written function or global variable is an atom.
  - In addition, the compiler may emit other atoms, such as for literal c-strings or floating point constants, or for runtime data structures like dwarf unwind info or pointers to initializers.
- Atoms classified:
  - The following Hello World code can be classified with these different kinds of Atoms as follows,

#### Atom example code

```
extern int printf(const char *format, ...);

int main(void)
{
    char *ptr = "Hello world!";

    printf("%s\n", ptr);
}
```

- DefinedAtom
  - \* 95% of all atoms. This is a chunk of code or data
- UndefinedAtom
  - \* printf in this example.
- SharedLibraryAtom
  - \* Symbols defined in shared library (file \*.so).



- AbsoluteAtom
  - \* This is for embedded support where some stuff is implemented in ROM at some fixed address.

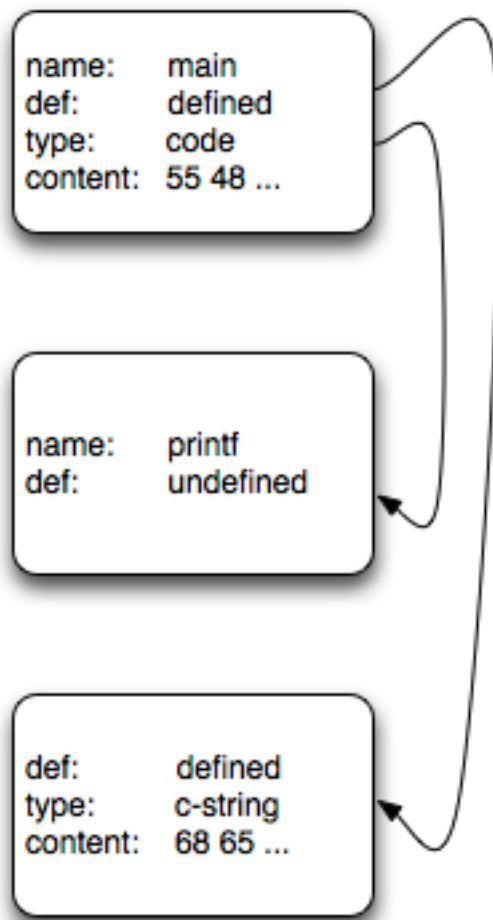


Figure 2.2: Atom classified (from lld web)

## Linking Steps

- Command line processing
  - `lld -flavor gnu -target cpu0-unknown-linux-gnu hello.o printf-stdarg.o -o a.out`
- Parsing input files
  - ELF reader => create `lld::File`
- Resolving
  - dead code stripping
- Passes/Optimizations

- Like llvm passes, give backend a chance to do something like optimization.
- Generate output file
  - Resolving Relocation Records – I guess in this step

### Command line processing

To support a new backend, refer to Cpu0 code added above.

### Parsing input files

- Input Files
  - A goal of lld is to be file format independent.
  - The lld::Reader is the base class for all object file readers
  - Every Reader subclass defines its own “options” class (for instance the mach-o Reader defines the class ReaderOptionsMachO). This options class is the one-and-only way to control how the Reader operates when parsing an input file into an Atom graph
- Reader
  - Refer lld/lib/ReaderWriter/ELF/Reader.cpp and it’s related files since lld change quickly.
- lld::File representations
  - In memory, abstract C++ classes (lld::Atom, lld::Reference, and lld::File).
    - \* Data structure kept in memory to be fast
  - textual (in YAML)
    - \* target-triple: x86\_64-apple-darwin11
    - \* atoms:
      - name: \_main
      - scope: global
      - type: code
      - content: [ 55, 48, 89, e5, 48, 8d, 3d, 00, 00, 00, 00, 30, c0, e8, 00, 00, 00, 00, 31, c0, 5d, c3 ]
  - binary format (“native”)
    - \* With this model for the native file format, files can be read and turned into the in-memory graph of lld::Atoms with just a few memory allocations. And the format can easily adapt over time to new features.

### Resolving

- Dead code stripping (if requested) is done at the end of resolving.
- The linker does a simple mark-and-sweep. It starts with “**root**” atoms (like “main” in a main executable) and follows each references and marks each Atom that it visits as “**live**”.
- When done, all atoms not marked “**live**” are removed.

## Dead code stripping - example (modified from llvm lto document web)

a.h

```
extern int foo1(void);
extern void foo2(void);
extern int foo4(void);
```

a.cpp

```
#include "a.h"

static signed int i = 0;

void foo2(void) {
    i = -1;
}

static int foo3() {
    return (10+foo4());
}

int foo1(void) {
    int data = 0;

    if (i < 0)
        data = foo3();

    data = data + 42;
    return data;
}
```

ch13\_1.cpp

```
#include "a.h"

void ISR() {
    asm("ISR:");
    return;
}

int foo4(void) {
    return 5;
}

int main() {
    return foo1();
}
```

Above code can be reduced to [Figure 2.3](#) to perform mark and swip in graph for Dead Code Stripping.

As above example, the foo2() is an isolated node without any reference. It's dead code and can be removed in linker optimization. We test this example by build-ch13\_1.sh and find foo2() cannot be removed. There are two possibilities for this situation. One is we do not trigger lld dead code stripping optimization in command (the default is not do it).

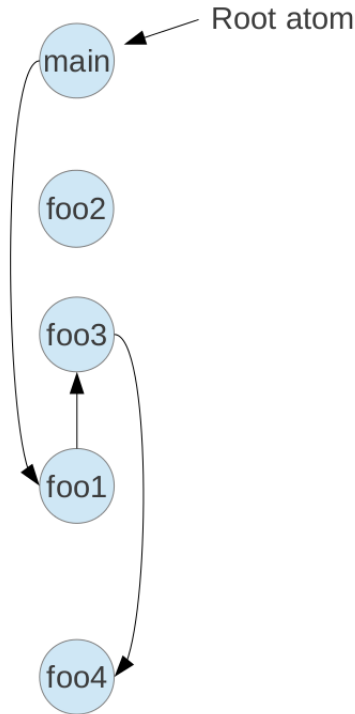


Figure 2.3: Atom classified (from lld web)

The other is lld hasn't implemented it yet at this point. It's reasonable since the lld is in its early stages of development. We didn't dig it more, since the Cpu0 backend tutorial just need a linker to finish Relocation Records Resolve and see how it runs on PC.

Remind, llvm-linker is the linker works on IR level linker optimization. Sometime when you got the obj file only (if you have a.o in this case), the native linker (such as lld) have the opportunity to do Dead Code Stripping while the IR linker hasn't.

## Passes/Optimizations

- Passes
  - stub (PLT) generation
  - GOT instantiation
  - order\_file optimization
  - branch island generation
  - branch shim generation
  - Objective-C optimizations (Darwin specific)
  - TLV instantiation (Darwin specific)
  - DTrace probe processing (Darwin specific)
  - compact unwind encoding (Darwin specific)

The Cpu0RelocationPass.cpp and Cpu0RelocationPass.h are example codes for lld backend Passes. The Relocation Pass structure shown as Figure 2.4. The Cpu0 backend has two Relocation Passes and both of them are children of

RelocationPass. The StaticRelocationPass is for static linker and DynamicRelocationPass is for dynamic linker. We will see how to register a relocation pass according the staic or dynamic linker you like to do in next section.

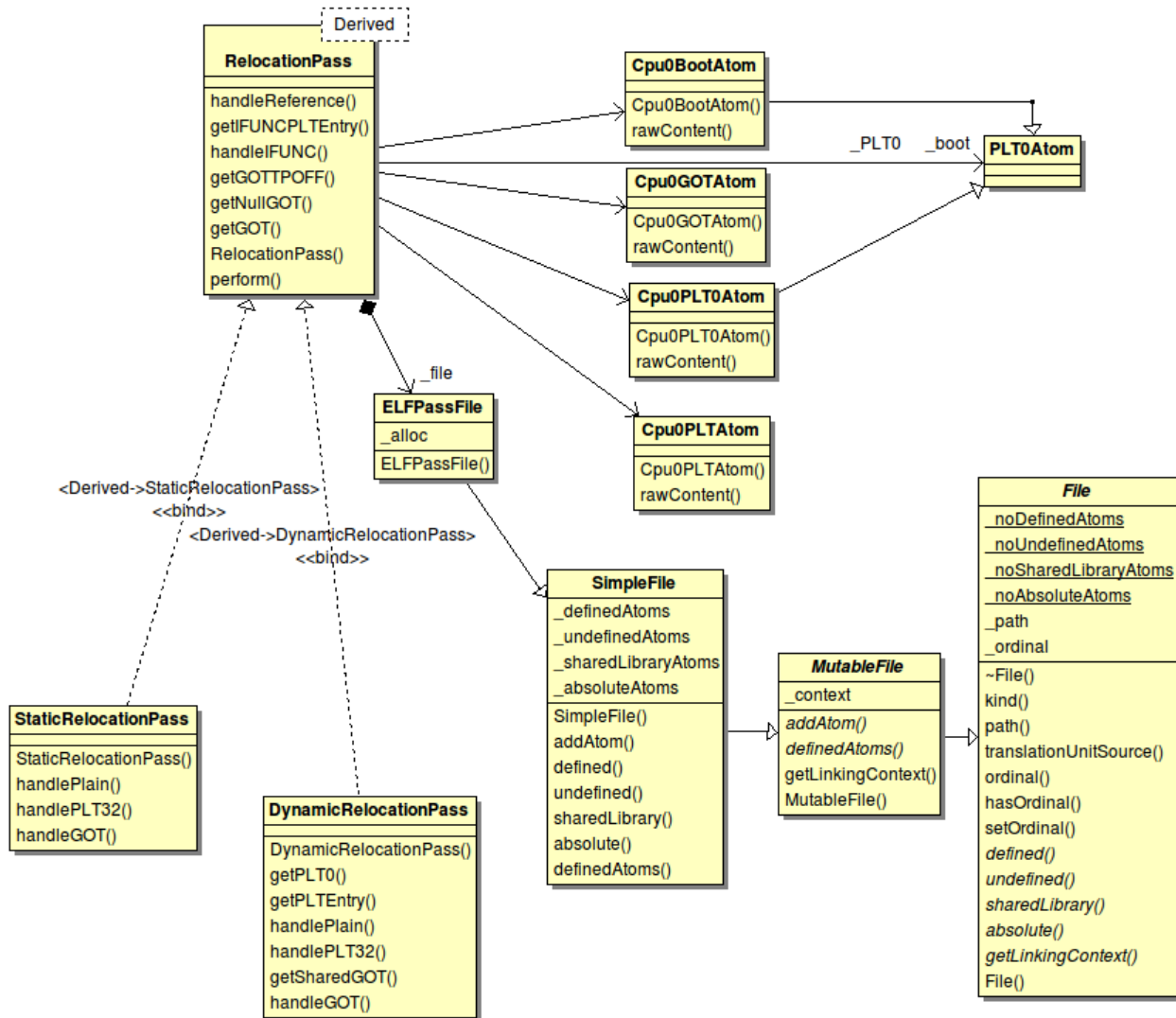


Figure 2.4: Cpu0 lld RelocationPass

All lld backends which want to handle the Relocation Records Resolve need to register a pass when the lld backend code is up. After register the pass, LLD will do last two steps, Passes/Optimization and Generate Output file, inter-actively just like the “Parsing and Generating code” in compiler. LLD will do Passes/Optimization and call your lld backend hook function “applyRelocation()” (defined in Cpu0TargetRelocationHandler.cpp) to finish the address binding in linker stage. Based on this understanding, we believe the “applyRelocation()” is at the step of Generate output file rather than Passes/Optimization even LLD web document didn’t indicate this.

The following code will register a pass when the lld backend code is up.

#### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.cpp

```
std::unique_ptr<Pass>
lld::elf::createCpu0RelocationPass(const Cpu0LinkingContext &ctx) {
```

```
switch (ctx.getOutputELFType()) {
case llvm::ELF::ET_EXEC:
    // when the output file is execution file: e.g. a.out
#ifdef DLINKER
    if (ctx.isDynamic())
        return llvm::make_unique<DynamicRelocationPass>(ctx);
    else
        return std::unique_ptr<Pass>(new StaticRelocationPass(ctx));
#else
    return llvm::make_unique<StaticRelocationPass>(ctx);
#endif // DLINKER
#ifdef DLINKER
case llvm::ELF::ET_DYN:
    // when the output file is shared object: e.g. foobar.so
    return llvm::make_unique<DynamicRelocationPass>(ctx);
#endif // DLINKER
case llvm::ELF::ET_REL:
    return nullptr;
default:
    llvm_unreachable("Unhandled output file type");
}
```

## Generate Output File

- All concrete writers (e.g. ELF, mach-o, etc) are subclasses of the `lld::Writer` class.
- Every `Writer` subclass defines its own “options” class (for instance the mach-o `Writer` defines the class `WriterOptionsMachO`). This options class is the one-and-only way to control how the `Writer` operates when producing an output file from an `Atom` graph.
- `Writer`
  - Refer `lld/lib/ReaderWriter/ELF/Writer.cpp` and it’s related files since `lld` change quickly.

After register a relocation pass, `lld` backend hook function “`applyRelocation()`” will be called by `lld` driver to finish the address binding in linker stage.

### `exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.cpp`

```
template <class ELFT> std::error_code Cpu0TargetRelocationHandler<ELFT>::applyRelocation(
    ELFWriter &writer, llvm::FileOutputBuffer &buf, const AtomLayout &atom,
    const Reference &ref) const {
    ...
    uint8_t *atomContent = buf.getBufferStart() + atom._fileOffset;
    uint8_t *loc = atomContent + ref.offsetInAtom();
    uint64_t target = writer.addressOfAtom(ref.target());
    uint64_t reloc = atom._virtualAddr + ref.offsetInAtom();
    ...
    switch (ref.kind()) {
    case R_CPU0_NONE:
        break;
    case R_CPU0_HI16:
        relocHI16(loc, reloc, target, ref.addend(), _endian);
        break;
    case R_CPU0_LO16:

```

```

    relocLO16(loc, reloc, target, ref.addend(), _endian);
    break;
...
case R_CPU0_PC24:
    relocPC24(loc, reloc, target, ref.addend(), _endian);
    break;
...
}
return error_code::success();
}

```

### exlbt/input/ch\_hello.c

```

extern int printf(const char *format, ...);

int main(void)
{
    char *ptr = "Hello world!";

    printf("%s\n", ptr);
}

```

### exlbt/input/build-hello.sh

```

#!/usr/bin/env bash

source functions.sh

sh_name=build-hello.sh
argNum=$#
arg1=$1
arg2=$2

prologue;

clang -target mips-unknown-linux-gnu -c start.cpp -emit-llvm -o start.bc
clang -target mips-unknown-linux-gnu -c debug.cpp -emit-llvm -o debug.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg-def.c -emit-llvm \
-o printf-stdarg-def.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg.c -emit-llvm \
-o printf-stdarg.bc
clang -target mips-unknown-linux-gnu -c ch_hello.c -emit-llvm -o ch_hello.bc
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj start.bc -o start.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj debug.bc -o debug.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg-def.bc -o printf-stdarg-def.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg.c -o printf-stdarg.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch_hello.bc -o ch_hello.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj lib_cpu0.ll -o lib_cpu0.o
${TOOLDIR}/lld -flavor gnu -target cpu0${endian}-unknown-linux-gnu \

```

```
start.cpu0.o debug.cpu0.o printf-stdarg-def.cpu0.o printf-stdarg.cpu0.o \
ch_hello.cpu0.o lib_cpu0.o -o a.out
```

```
epilogue;
```

### exlbt/verilog/Cpu0.hex

```
...
/*printf:*/
/*      b4:*/ 09 dd ff e0                                /* addiu    $sp, $sp, -32*/
...
/*main:*/
/*      9e0:*/ 09 dd ff e8                                /* addiu    $sp, $sp, -24*/
...
/*      9f0:*/ 0f 20 00 00                                /* lui      $2, 0*/
/*      9f4:*/ 0d 22 0b 9f                                /* ori      $2, $2, 2975*/
...
/*      a0c:*/ 3b ff f6 a4                                /* jsub     16774820*/
...
/*Contents of section .rodata:*/
/*0b98 */28 6e 75 6c 6c 29 00 48 65 6c 6c 6f 20 77 6f 72 /* (null).Hello wor*/
/*0ba8 */6c 64 21 00 25 73 0a 00 /* ld!.\%s..*/
```

As you can see, `applyRelocation()` get four values for the Relocation Records Solving. When meets `R_CPU0_LO16`, `targetVAddress` is the only one value needed for this Relocation Solving in these four values. For this `ch_hello.c` example code, the `lld` set the “Hello world!” string begin at `0x0b98+7=0x0b9f`. So, `targetVAddress` is `0x0b9f`. These two instructions “`lui`” and “`ori`” at address `0x9f0` and `0x9f4`, respectively, loading the address of “Hello world!” string to register `$2`. The “`lui`” got the HI 16 bits while the “`ori`” got the LO 16 bits of address of “Hello world!” string. This “`lui`” Relocation Record, `R_CPU0_HI16`, is 0 since the HI 16 bits of `0xb9f` is 0 while the “`ori`” Relocation Record, `R_CPU0_LO16`, is `0xb9f`. The instruction “`jsub`” at `0xa0c` is an instruction jump to `printf()`. This instruction is a PC relative address Relocation Record, `R_CPU0_PC24`, while the `R_CPU0_LO16` is an absolute address Relocation Record. To solve this Relocation Record, it needs “location” in addition to `targetVAddress`. In this case, the `targetVAddress` is `0xb4` where is the `printf` subroutine start address and the location is `0xa0c` since the instruction “`jsub`” sit at this address. The `R_CPU0_PC24` is solved by  $(0xb4 - (0xa0c + 4) = 0xf6a4$  for 16 bits with sign extension) since after this “`jsub`” instruction executed the PC counter is  $(0xa0c+4)$ . To +4 at current instruction because PC counter increased at instruction fetch stage in Verilog design.

Remind, we explain the Relocation Records Solving according file `cpu0.hex` list as above because the the Cpu0 machine boot at memory address `0x0` while the elf text section or plt section as follows start at `0x140`. The `0x0` is the header of machine architecture information. The elf2hex code must keeps the address relative distance between text and plt sections just like the Cpu0 elf2hex.h did. The .rodata and other data sections are binding with absolute address, Cpu0 elf2hex must keeps them as the same address of elf.

For the following example code run, the book example code, `exlbt.tar.gz`, untared in directory `/Users/Jonathan/test/lbt/`. The Cpu0 backend code, `lbdex.tar.gz`, untared in the same directory too. The `lbdex.tar.gz` can be get from the bottom of web, <http://jonathan2251.github.io/lbd/index.html>.

```
1-160-136-173:input Jonathan$ pwd
/Users/Jonathan/test/lbt/exlbt/input
1-160-136-173:input Jonathan$ ls ../../
... exlbt ... lbdex ...
/Users/Jonathan/test/lbt/exlbt/input
1-160-136-173:input Jonathan$ bash build-hello.sh cpu032I be
1-160-136-173:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llvm-objdump -s a.out
...
.
```



```

Contents of section .plt:
 0140 3600000c 36000004 36000004 36ffffffc  6...6...6...6...
Contents of section .text:
 0150 09ddfff8 02ed0004 02cd0000 11cd0000  .....
...
Contents of section .rodata:
0b98 286e756c 6c290048 656c6c6f 20776f72  (null).Hello wor
0ba8 6c642100 25730a00                ld!.%s..

```

Next section will show you how to design your lld backend and register a pass for Relocation Records Solve in details through Cpu0 backend code explanation.

## 2.2.4 Static linker

Let's run the static linker first and explain it next.

### Run

File printf-stdarg.c come from internet download which is GPL2 license. GPL2 is more restricted than LLVM license. File printf-stdarg-1.c is the file for testing the printf() function which implemented on PC OS platform. Let's run printf-stdarg-2.cpp on Cpu0 and compare it against the result of PC's printf() as below.

#### exlbt/input/printf-stdarg-1.c

```

/*
  Copyright 2001, 2002 Georges Menie (www.menie.org)
  stdarg version contributed by Christian Ettinger

  This program is free software; you can redistribute it and/or modify
  it under the terms of the GNU Lesser General Public License as published by
  the Free Software Foundation; either version 2 of the License, or
  (at your option) any later version.

  This program is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
  GNU Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public License
  along with this program; if not, write to the Free Software
  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

/*
  putchar is the only external dependency for this file,
  if you have a working putchar, leave it commented out.
  If not, uncomment the define below and
  replace outbyte(c) by your own function call.

#define putchar(c) outbyte(c)
*/

// gcc printf-stdarg-1.c
// ./a.out

```

```
#include <stdio.h>

#define TEST_PRINTF

#ifndef TEST_PRINTF
int main(void)
{
    char *ptr = "Hello world!";
    char *np = 0;
    int i = 5;
    unsigned int bs = sizeof(int)*8;
    int mi;
    char buf[80];

    mi = (1 << (bs-1)) + 1;
    printf("%s\n", ptr);
    printf("printf test\n");
    printf("%s is null pointer\n", np);
    printf("%d = 5\n", i);
    printf("%d = - max int\n", mi);
    printf("char %c = 'a'\n", 'a');
    printf("hex %x = ff\n", 0xff);
    printf("hex %02x = 00\n", 0);
    printf("signed %d = unsigned %u = hex %x\n", -3, -3, -3);
    printf("%d %s(s)%", 0, "message");
    printf("\n");
    printf("%d %s(s) with %%\n", 0, "message");
    sprintf(buf, "justif: \"%-10s\"\n", "left"); printf("%s", buf);
    sprintf(buf, "justif: \"%10s\"\n", "right"); printf("%s", buf);
    sprintf(buf, " 3: %04d zero padded\n", 3); printf("%s", buf);
    sprintf(buf, " 3: %-4d left justif.\n", 3); printf("%s", buf);
    sprintf(buf, " 3: %4d right justif.\n", 3); printf("%s", buf);
    sprintf(buf, "-3: %04d zero padded\n", -3); printf("%s", buf);
    sprintf(buf, "-3: %-4d left justif.\n", -3); printf("%s", buf);
    sprintf(buf, "-3: %4d right justif.\n", -3); printf("%s", buf);

    return 0;
}

/*
 * if you compile this file with
 * gcc -Wall $(YOUR_C_OPTIONS) -DTEST_PRINTF -c printf.c
 * you will get a normal warning:
 * printf.c:214: warning: spurious trailing '%' in format
 * this line is testing an invalid % at the end of the format string.
 *
 * this should display (on 32bit int machine) :
 *
 * Hello world!
 * printf test
 * (null) is null pointer
 * 5 = 5
 * -2147483647 = - max int
 * char a = 'a'
 * hex ff = ff
 * hex 00 = 00
 * signed -3 = unsigned 4294967293 = hex ffffffff
 * 0 message(s)
 */
```

```

* 0 message(s) with %
* justif: "left      "
* justif: "      right"
* 3: 0003 zero padded
* 3: 3    left justif.
* 3:    3 right justif.
* -3: -003 zero padded
* -3: -3   left justif.
* -3:   -3 right justif.
*/

```

```
#endif
```

#### exlbt/input/printf-stdarg-2.cpp

```

#include "debug.h"
#include "print.h"

#define TEST_PRINTF

extern "C" int putchar(int c);

extern "C" {
#include "printf-stdarg.c"
}

```

#### exlbt/input/printf-stdarg-def.c

```

#include "print.h"

// Definition putchar(int c) for printf-stdarg.c
// For memory IO
int putchar(int c)
{
    char *p = (char*)OUT_MEM;
    *p = c;

    return 0;
}

```

#### exlbt/input/printf-stdarg.c

```

/*
Copyright 2001, 2002 Georges Menie (www.menie.org)
stdarg version contributed by Christian Ettinger

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of

```

```
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/

/*
    putchar is the only external dependency for this file,
    if you have a working putchar, leave it commented out.
    If not, uncomment the define below and
    replace outbyte(c) by your own function call.

#define putchar(c) outbyte(c)
*/

#include <stdarg.h>

static void printchar(char **str, int c)
{
    extern int putchar(int c);

    if (str) {
        **str = c;
        ++(*str);
    }
    else (void)putchar(c);
}

#define PAD_RIGHT 1
#define PAD_ZERO 2

static int prints(char **out, const char *string, int width, int pad)
{
    register int pc = 0, padchar = ' ';

    if (width > 0) {
        register int len = 0;
        register const char *ptr;
        for (ptr = string; *ptr; ++ptr) ++len;
        if (len >= width) width = 0;
        else width -= len;
        if (pad & PAD_ZERO) padchar = '0';
    }
    if (!(pad & PAD_RIGHT)) {
        for ( ; width > 0; --width) {
            printchar (out, padchar);
            ++pc;
        }
    }
    for ( ; *string ; ++string) {
        printchar (out, *string);
        ++pc;
    }
    for ( ; width > 0; --width) {
        printchar (out, padchar);
        ++pc;
    }
}
```

```

    }

    return pc;
}

/* the following should be enough for 32 bit int */
#define PRINT_BUF_LEN 12

static int printi(char **out, int i, int b, int sg, int width, int pad, int letbase)
{
    char print_buf[PRINT_BUF_LEN];
    register char *s;
    register int t, neg = 0, pc = 0;
    register unsigned int u = i;

    if (i == 0) {
        print_buf[0] = '0';
        print_buf[1] = '\\0';
        return prints (out, print_buf, width, pad);
    }

    if (sg && b == 10 && i < 0) {
        neg = 1;
        u = -i;
    }

    s = print_buf + PRINT_BUF_LEN-1;
    *s = '\\0';

    while (u) {
        t = u % b;
        if( t >= 10 )
            t += letbase - '0' - 10;
        *--s = t + '0';
        u /= b;
    }

    if (neg) {
        if( width && (pad & PAD_ZERO) ) {
            printchar (out, '-');
            ++pc;
            --width;
        }
        else {
            *--s = '-';
        }
    }

    return pc + prints (out, s, width, pad);
}

static int print(char **out, const char *format, va_list args )
{
    register int width, pad;
    register int pc = 0;
    char scr[2];

    for (; *format != 0; ++format) {

```

```
if (*format == '%') {
    ++format;
    width = pad = 0;
    if (*format == '\\0') break;
    if (*format == '%') goto out;
    if (*format == '-') {
        ++format;
        pad = PAD_RIGHT;
    }
    while (*format == '0') {
        ++format;
        pad |= PAD_ZERO;
    }
    for ( ; *format >= '0' && *format <= '9'; ++format) {
        width *= 10;
        width += *format - '0';
    }
    if (*format == 's') {
        register char *s = (char *)va_arg( args, int );
        pc += prints (out, s?s:"(null)", width, pad);
        continue;
    }
    if (*format == 'd') {
        pc += printi (out, va_arg( args, int ), 10, 1, width, pad, 'a');
        continue;
    }
    if (*format == 'x') {
        pc += printi (out, va_arg( args, int ), 16, 0, width, pad, 'a');
        continue;
    }
    if (*format == 'X') {
        pc += printi (out, va_arg( args, int ), 16, 0, width, pad, 'A');
        continue;
    }
    if (*format == 'u') {
        pc += printi (out, va_arg( args, int ), 10, 0, width, pad, 'a');
        continue;
    }
    if (*format == 'c') {
        /* char are converted to int then pushed on the stack */
        scr[0] = (char)va_arg( args, int );
        scr[1] = '\\0';
        pc += prints (out, scr, width, pad);
        continue;
    }
}
else {
out:
    putchar (out, *format);
    ++pc;
}
}
if (out) **out = '\\0';
va_end( args );
return pc;
}

int printf(const char *format, ...)
```

```

{
    va_list args;

    va_start( args, format );
    return print( 0, format, args );
}

int sprintf(char *out, const char *format, ...)
{
    va_list args;

    va_start( args, format );
    return print( &out, format, args );
}

#ifdef TEST_PRINTF
int main(void)
{
    char *ptr = "Hello world!";
    char *np = 0;
    int i = 5;
    unsigned int bs = sizeof(int)*8;
    int mi;
    char buf[80];

    mi = (1 << (bs-1)) + 1;
    printf("%s\n", ptr);
    printf("printf test\n");
    printf("%s is null pointer\n", np);
    printf("%d = 5\n", i);
    printf("%d = - max int\n", mi);
    printf("char %c = 'a'\n", 'a');
    printf("hex %x = ff\n", 0xff);
    printf("hex %02x = 00\n", 0);
    printf("signed %d = unsigned %u = hex %x\n", -3, -3, -3);
    printf("%d %s(s)%", 0, "message");
    printf("\n");
    printf("%d %s(s) with %%\n", 0, "message");
    sprintf(buf, "justif: \"%-10s\"\n", "left"); printf("%s", buf);
    sprintf(buf, "justif: \"%10s\"\n", "right"); printf("%s", buf);
    sprintf(buf, " 3: %04d zero padded\n", 3); printf("%s", buf);
    sprintf(buf, " 3: %-4d left justif.\n", 3); printf("%s", buf);
    sprintf(buf, " 3: %4d right justif.\n", 3); printf("%s", buf);
    sprintf(buf, "-3: %04d zero padded\n", -3); printf("%s", buf);
    sprintf(buf, "-3: %-4d left justif.\n", -3); printf("%s", buf);
    sprintf(buf, "-3: %4d right justif.\n", -3); printf("%s", buf);

    return 0;
}

/*
 * if you compile this file with
 * gcc -Wall $(YOUR_C_OPTIONS) -DTEST_PRINTF -c printf.c
 * you will get a normal warning:
 * printf.c:214: warning: spurious trailing '%' in format
 * this line is testing an invalid % at the end of the format string.
 *
 * this should display (on 32bit int machine) :

```

```
*
* Hello world!
* printf test
* (null) is null pointer
* 5 = 5
* -2147483647 = - max int
* char a = 'a'
* hex ff = ff
* hex 00 = 00
* signed -3 = unsigned 4294967293 = hex ffffffff
* 0 message(s)
* 0 message(s) with %
* justif: "left      "
* justif: "      right"
* 3: 0003 zero padded
* 3: 3    left justif.
* 3:    3 right justif.
* -3: -003 zero padded
* -3: -3   left justif.
* -3:   -3 right justif.
*/

#endif
```

#### exlbt/input/start.cpp

```
#include "dynamic_linker.h"
#include "start.h"

extern int main();

// Real entry (first instruction) is from cpu0BootAtomContent of
// Cpu0RelocationPass.cpp jump to asm("start:") of start.cpp.
void start() {
    asm("start:");

    asm("lui $sp, 0x6");
    asm("addiu $sp, $sp, 0xffffc");
    int *gpaddr;
    gpaddr = (int*) GPADDR;
    __asm__ __volatile__ ("ld  $gp, %0"
                          : // no output register, specify output register to $gp
                          : "m" (*gpaddr)
                          );

    initRegs();
    main();
    asm("addiu $lr, $ZERO, -1");
    asm("ret $lr");
}
```

#### exlbt/input/lib\_cpu0.ll

```
; The @_start() exist to prevent lld linker error.
; Real entry (first instruction) is from cpu0BootAtomContent of
; Cpu0RelocationPass.cpp jump to asm("start:") of start.cpp.
```



```
define void @_start() nounwind {
entry:
    ret void
}

define void @__start() nounwind {
entry:
    ret void
}

define void @__stack_chk_fail() nounwind {
entry:
    ret void
}

define void @__stack_chk_guard() nounwind {
entry:
    ret void
}

define void @_ZdlPv() nounwind {
entry:
    ret void
}

define void @__dso_handle() nounwind {
entry:
    ret void
}

define void @_ZNSt8ios_base4InitC1Ev() nounwind {
entry:
    ret void
}

define void @__cxa_atexit() nounwind {
entry:
    ret void
}

define void @_ZTVN10__cxxabiv120__si_class_type_infoE() nounwind {
entry:
    ret void
}

define void @_ZTVN10__cxxabiv117__class_type_infoE() nounwind {
entry:
    ret void
}

define void @_Znwm() nounwind {
entry:
    ret void
}

define void @__cxa_pure_virtual() nounwind {
entry:
    ret void
}
```

```
}

define void @_ZNSt8ios_base4InitD1Ev() nounwind {
entry:
    ret void
}
```

### **exlbt/input/functions.sh**

```
prologue() {
    LBDEXDIR=../../lbdex

    if [ $# == 0 ]; then
        echo "usage: bash $sh_name cpu_type endian"
        echo "  cpu_type: cpu032I or cpu032II"
        echo "  endian: be (big endian, default) or le (little endian)"
        echo "for example:"
        echo "  bash build-slinker.sh cpu032I be"
        exit 1;
    fi

    if [ $arg1 != cpu032I ] && [ $arg1 != cpu032II ]; then
        echo "1st argument is cpu032I or cpu032II"
        exit 1
    fi

    INCDIR=../../lbdex/input
    OS=`uname -s`
    echo "OS =" "${OS}"

    if [ "$OS" == "Linux" ]; then
        CLANGDIR=~/.llvm/release/cmake_debug_build/bin
        TOOLDIR=~/.llvm/test/cmake_debug_build/bin
    else
        CLANGDIR=~/.llvm/release/cmake_debug_build/Debug/bin
        TOOLDIR=~/.llvm/test/cmake_debug_build/Debug/bin
    fi

    CPU=$arg1
    echo "CPU =" "${CPU}"

    if [ "$arg2" != "" ] && [ $arg2 != le ] && [ $arg2 != be ]; then
        echo "2nd argument is be (big endian, default) or le (little endian)"
        exit 1
    fi

    if [ "$arg2" == "" ] || [ $arg2 == be ]; then
        endian=
    else
        endian=le
    fi
    echo "endian =" "${endian}"

    bash clean.sh
}

isLittleEndian() {
    echo "endian = " "$endian"
}
```

```

if [ "$endian" == "LittleEndian" ] ; then
    le="true"
elif [ "$endian" == "BigEndian" ] ; then
    le="false"
else
    echo "!endian unknown"
    exit 1
fi
}

elf2hex() {
    ${TOOLDIR}/llvm-objdump -elf2hex -le=${le} a.out > ${LBDEXDIR}/verilog/cpu0.hex
    if [ ${le} == "true" ] ; then
        echo "1 /* 0: big endian, 1: little endian */" > ${LBDEXDIR}/verilog/cpu0.config
    else
        echo "0 /* 0: big endian, 1: little endian */" > ${LBDEXDIR}/verilog/cpu0.config
    fi
    cat ${LBDEXDIR}/verilog/cpu0.config
}

epilogue() {
    endian='${TOOLDIR}/llvm-readobj -h a.out|grep "DataEncoding"|awk '{print $2}'`
    isLittleEndian;
    elf2hex;
}

```

### exlbt/input/build-printf-stdarg-2.sh

```

#!/usr/bin/env bash

source functions.sh

sh_name=build-printf-stdarg-2.sh
argNum=$#
arg1=$1
arg2=$2

prologue;

clang -target mips-unknown-linux-gnu -c start.cpp -emit-llvm -o start.bc
clang -target mips-unknown-linux-gnu -c debug.cpp -emit-llvm -o debug.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg-def.c -emit-llvm \
-o printf-stdarg-def.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg-2.cpp -emit-llvm -o \
printf-stdarg-2.bc
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj start.bc -o start.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj debug.bc -o debug.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg-def.bc -o printf-stdarg-def.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg-2.bc -o printf-stdarg-2.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj lib_cpu0.ll -o lib_cpu0.o
${TOOLDIR}/lld -flavor gnu -target cpu0${endian}-unknown-linux-gnu \

```

```
start.cpu0.o debug.cpu0.o printf-stdarg-def.cpu0.o printf-stdarg-2.cpu0.o \
lib_cpu0.o -o a.out
```

```
epilogue;
```

The verilog/cpu0Is.v support cmp instruction and static linker as follows,

### lbdex/verilog/cpu0Is.v

```
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE
```

```
`include "cpu0.v"
```

The verilog/cpu0IIs.v support slt instruction and static linker as follows,

### lbdex/verilog/cpu0IIs.v

```
`define CPU0II
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE
```

```
`include "cpu0.v"
```

The build-printf-stdarg-2.sh is for my PC setting. Please change this script to the directory of your llvm/lld setting. After that run static linker example code as follows,

```
1-160-136-173:input Jonathan$ pwd
/Users/Jonathan/test/lbt/exlbt/input
1-160-136-173:input Jonathan$ bash build-printf-stdarg-2.sh cpu032I be
In file included from printf-stdarg-2.cpp:11:
./printf-stdarg.c:206:15: warning: conversion from string literal to 'char *'
is deprecated [-Wdeprecated-writable-strings]
    char *ptr = "Hello world!";
                  ^
1 warning generated.

1-160-136-173:input Jonathan$ cd ../../lbdex/verilog/
1-160-136-173:verilog Jonathan$ pwd
/Users/Jonathan/test/lbt/lbdex/lbdex/verilog
1-160-136-173:verilog Jonathan$ bash clean.sh
1-160-136-173:verilog Jonathan$ iverilog -o cpu0II cpu0IIs.v
Hello world!
printf test
(null) is null pointer
5 = 5
-2147483647 = - max int
char a = 'a'
hex ff = ff
hex 00 = 00
signed -3 = unsigned 4294967293 = hex ffffffff
0 message(s)
0 message(s) with \%
justif: "left      "
justif: "      right"
3: 0003 zero padded
```

```

3: 3    left justif.
3:    3 right justif.
-3: -003 zero padded

```

Let's check the result with PC program printf-stdarg-1.c output as follows,

```

1-160-136-173:input Jonathan$ clang printf-stdarg-1.c
printf-stdarg-1.c:58:19: warning: incomplete format specifier [-Wformat]
    printf("%d %s(s)%", 0, "message");
                        ^
1 warning generated.
1-160-136-173:input Jonathan$ ./a.out
Hello world!
printf test
(null) is null pointer
5 = 5
-2147483647 = - max int
char a = 'a'
hex ff = ff
hex 00 = 00
signed -3 = unsigned 4294967293 = hex ffffffff
0 message(s)
0 message(s) with \%
justif: "left    "
justif: "    right"
3: 0003 zero padded
3: 3    left justif.
3:    3 right justif.
-3: -003 zero padded
-3: -3   left justif.
-3:   -3 right justif.

```

They are same. You can verify the slt instructions is work fine too by change variable cpu from cpu032I to cpu032II as follows,

### exlbt/input/build-printf-stdarg-2.sh

```

1-160-136-173:verilog Jonathan$ pwd
/Users/Jonathan/test/lbt/lbdex/verilog
1-160-136-173:verilog Jonathan$ cd ../../exlbt/input
1-160-136-173:input Jonathan$ pwd
/Users/Jonathan/test/lbt/exlbt/input
1-160-136-173:input Jonathan$ bash build-printf-stdarg-2.sh cpu032II be
...
1-160-136-173:input Jonathan$ cd ../lbdex/verilog/
1-160-136-173:verilog Jonathan$ ./cpu0II.s

```

The verilog machine cpu0II.s include all instructions of cpu032I and add slt, beq, ..., instructions. Run build-printf-stdarg-2.sh with cpu=cpu032II will generate slt, beq and bne instructions instead of cmp, jeq, ... instructions.

With the printf() of GPL source code, we can program more test code with it to verify the previous llvm Cpu0 backend generated program. The following code is for this purpose.

**exlbt/input/debug.cpp**

```
#include "debug.h"

extern "C" int printf(const char *format, ...);

// With read variable from asm, such as sw in this example, the function,
// ISR_Handler() must entry from beginning. The ISR() enter from "ISR:" will
// has incorrect value for reload instruction in offset.
// For example, the correct one is:
//  "addiu $sp, $sp, -12"
//  "mov $fp, $sp"
// ISR:
//  "ld $2, 32($fp)"
// Go to ISR directly, then the $fp is 12+ than original, then it will get
//  "ld $2, 20($fp)" actually.
void ISR_Handler() {
    SAVE_REGISTERS;
    asm("lui $7, 0xffff");
    asm("ori $7, $7, 0xfdf");
    asm("and $sw, $sw, $7"); // clear 'IE

    volatile int sw;
    __asm__ __volatile__ ("addiu %0, $sw, 0"
                          : "=r" (sw)
                          );
    int interrupt = (sw & INT);
    int softint = (sw & SOFTWARE_INT);
    int overflow = (sw & OVERFLOW);
    int int1 = (sw & INT1);
    int int2 = (sw & INT2);
    if (interrupt) {
        if (softint) {
            if (overflow) {
                printf("Overflow exception\n");
                CLEAR_OVERFLOW;
            }
            else {
                printf("Software interrupt\n");
            }
            CLEAR_SOFTWARE_INT;
        }
        else if (int1) {
            printf("Hardware interrupt 0\n");
            asm("lui $7, 0xffff");
            asm("ori $7, $7, 0x7fff");
            asm("and $sw, $sw, $7");
        }
        else if (int2) {
            printf("Hardware interrupt 1\n");
            asm("lui $7, 0xfffe");
            asm("ori $7, $7, 0xffff");
            asm("and $sw, $sw, $7");
        }
        asm("lui $7, 0xffff");
        asm("ori $7, $7, 0xdfff");
        asm("and $sw, $sw, $7"); // clear 'I
    }
}
```

```

    asm("ori $sw, $sw, 0x200"); // int enable
    RESTORE_REGISTERS;
    return;
}

void ISR() {
    asm("ISR:");
    asm("lui $at, 7");
    asm("ori $at, $at, 0xff00");
    asm("st $14, 48($at)");
    ISR_Handler();
    asm("lui $at, 7");
    asm("ori $at, $at, 0xff00");
    asm("ld $14, 48($at)");
    asm("c0mov $pc, $epc");
}

void int_sim() {
    asm("ori $sw, $sw, 0x200"); // int enable
    asm("ori $sw, $sw, 0x2000"); // set interrupt
    asm("ori $sw, $sw, 0x4000"); // Software interrupt
    asm("ori $sw, $sw, 0x200"); // int enable
    asm("ori $sw, $sw, 0x2000"); // set interrupt
    asm("ori $sw, $sw, 0x8000"); // hardware interrupt 0
    asm("ori $sw, $sw, 0x200"); // int enable
    asm("ori $sw, $sw, 0x2000"); // set interrupt
    asm("lui $at, 1");
    asm("or $sw, $sw, $at"); // hardware interrupt 1
    return;
}

```

#### exlbt/input/ch\_ild\_staticlink.h

```

#include "debug.h"
#include "print.h"

#define PRINT_TEST

extern "C" int printf(const char *format, ...);
extern "C" int sprintf(char *out, const char *format, ...);

extern unsigned char sBuffer[4];
extern int test_overflow();
extern int test_add_overflow();
extern int test_sub_overflow();
extern int test_ctrl2();
extern int test_alloc();

extern int test_staticlink();

```

#### exlbt/input/ch\_ild\_staticlink.cpp

```

void verify_test_ctrl2()
{
    int a = -1;
}

```

```
int b = -1;
int c = -1;
int d = -1;

sBuffer[0] = (unsigned char)0x35;
sBuffer[1] = (unsigned char)0x35;
a = test_ctrl2();
sBuffer[0] = (unsigned char)0x30;
sBuffer[1] = (unsigned char)0x29;
b = test_ctrl2();
sBuffer[0] = (unsigned char)0x35;
sBuffer[1] = (unsigned char)0x35;
c = test_ctrl2();
sBuffer[0] = (unsigned char)0x34;
d = test_ctrl2();
printf("test_ctrl2(): a = %d, b = %d, c = %d, d = %d", a, b, c, d);
if (a == 1 && b == 0 && c == 1 && d == 0)
    printf(", PASS\n");
else
    printf(", FAIL\n");

return;
}

int test_staticlink()
{
    int a = 0;

    a = test_add_overflow();
    a = test_sub_overflow();
    a = test_global(); // gI = 100
    printf("global variable gI = %d", a);
    if (a == 100)
        printf(", PASS\n");
    else
        printf(", FAIL\n");
    verify_test_ctrl2();
    a = test_func_arg_struct();
    a = test_constructor();
    a = test_template();
    printf("test_template() = %d", a); // a = 15
    if (a == 15)
        printf(", PASS\n");
    else
        printf(", FAIL\n");
    a = test_alloc(); // 31
    printf("test_alloc() = %d", a);
    if (a == 31)
        printf(", PASS\n");
    else
        printf(", FAIL\n");
    a = inlineasm_global(); // 4
    printf("inlineasm_global() = %d", a); // a = 4
    if (a == 4)
        printf(", PASS\n");
    else
        printf(", FAIL\n");
    a = test_cpp_polymorphism();
```



```
printf("test_cpp_polymorphism() = %d", a); // a = 0
if (a == 0)
    printf(", PASS\n");
else
    printf(", FAIL\n");

int_sim();

return 0;
}
```

#### exlbt/input/ch\_slinker.cpp

```
#include "ch_nolld.h"
#include "ch_lld_staticlink.h"

int main()
{
    bool pass = true;
    pass = test_nolld();
    if (pass)
        printf("test_nolld(): PASS\n");
    else
        printf("test_nolld(): FAIL\n");
    pass = true;
    pass = test_staticlink();

    return pass;
}

#include "ch_nolld.cpp"
#include "ch_lld_staticlink.cpp"
```

#### exlbt/input/build-slinker.sh

```
#!/usr/bin/env bash

source functions.sh

sh_name=build-slinker.sh
argNum=$#
arg1=$1
arg2=$2

prologue;

clang -target mips-unknown-linux-gnu -c start.cpp -emit-llvm -o start.bc
clang -target mips-unknown-linux-gnu -c debug.cpp -emit-llvm -o debug.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg-def.c -emit-llvm \
-o printf-stdarg-def.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg.c -emit-llvm \
-o printf-stdarg.bc
clang -target mips-unknown-linux-gnu -c ${LBDEXDIR}/input/ch4_1_2.cpp \
-emit-llvm -o ch4_1_2.bc
clang -target mips-unknown-linux-gnu -c ${LBDEXDIR}/input/ch8_1_5.cpp \
```

```
-emit-llvm -o ch8_1_5.bc
clang -c ${LBDEXDIR}/input/ch9_4.cpp -emit-llvm -o ch9_4.bc
clang -I${LBDEXDIR}/input/ -target mips-unknown-linux-gnu -c ch_slinker.cpp \
-emit-llvm -o ch_slinker.bc
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj start.bc -o start.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj debug.bc -o debug.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg-def.bc -o printf-stdarg-def.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg.bc -o printf-stdarg.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj -cpu0-enable-overflow=true ch4_1_2.bc -o ch4_1_2.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch8_1_5.bc -o ch8_1_5.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch9_4.bc -o ch9_4.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch_slinker.bc -o ch_slinker.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj lib_cpu0.ll -o lib_cpu0.o
${TOOLDIR}/lld -flavor gnu -target cpu0${endian}-unknown-linux-gnu start.cpu0.o \
debug.cpu0.o printf-stdarg-def.cpu0.o printf-stdarg.cpu0.o ch4_1_2.cpu0.o \
ch8_1_5.cpu0.o ch9_4.cpu0.o ch_slinker.cpu0.o lib_cpu0.o -o a.out
```

epilogue;

```
1-160-136-173:input Jonathan$ pwd
/Users/Jonathan/test/lbt/exlbt/input
114-37-148-111:input Jonathan$ bash build-slinker.sh cpu032I le
...
In file included from ch_slinker.cpp:23:
./ch_lld_staticlink.cpp:8:15: warning: conversion from string literal to
'char *' is deprecated
    [-Wdeprecated-writable-strings]
    char *ptr = "Hello world!";
                ^
1 warning generated.
114-37-148-111:input Jonathan$ cd ../../lbdex/verilog/
114-37-148-111:verilog Jonathan$ ./cpu0IIs
WARNING: ./cpu0.v:369: $readmemh(cpu0.hex): Not enough words in the file for
the requested range [0:524287].
taskInterrupt(001)
...
test_nolld(): PASS
taskInterrupt(011)
Overflow exception
taskInterrupt(011)
Overflow exception
test_overflow = 0, PASS
global variable gI = 100, PASS
test_ctrl2(): a = 1, b = 0, c = 1, d = 0, PASS
date1 = 2012 10 12 1 2 3, PASS
date2 = 2012 10 12 1 2 3, PASS
time2 = 1 10 12, PASS
time3 = 1 10 12, PASS
date1 = 2013 1 26 12 21 10, PASS
```

```

date2 = 2013 1 26 12 21 10, PASS
test_template() = 15, PASS
test_alloc() = 31, PASS
inlineasm_global() = 4, PASS
20
10
5
test_cpp_polymorphism() = 0, PASS
taskInterrupt(011)
Software interrupt
taskInterrupt(011)
Hardware interrupt 0
taskInterrupt(011)
Hardware interrupt 1
...

```

As above, by taking the open source code advantage, Cpu0 got the more stable printf() program. Once Cpu0 backend can translate the printf() function of the open source C printf() program into machine instructions, the llvm Cpu0 backend can be verified with printf(). With the quality code of open source printf() program, the Cpu0 toolchain is extended from compiler backend to C std library support. (Notice that some GPL open source code are not quality code, but some are.)

The “Overflow exception is printed twice meaning the ISR() of debug.cpp is called twice from ch4\_1\_2.cpp. The printed “taskInterrupt(001)” and “taskInterrupt(011)” just are trace message from cpu0.v code.

## Cpu0 lld structure

The Cpu0LinkingContext include the context information for those input obj files and output execution file you want to link. When do linking, the code added in GnuLdDriver.cpp will create Cpu0LinkingContext.

While Cpu0LinkingContext is created by lld ELF driver as above, the code in Cpu0LinkingContext constructor will create Cpu0TargetHandler and passing the Cpu0LinkingContext object pointer to Cpu0TargeHandler.

Finally, the Cpu0TargeHandler constructor will create other related objects and set up the relation reference object pointers as [Figure 2.5](#) depicted.

According chapter ELF, the linker stands for resolve the relocation records. The following code give the chance to let lld system call our relocation function at proper time.

### exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.cpp

```

std::unique_ptr<Pass>
lld::elf::createCpu0RelocationPass(const Cpu0LinkingContext &ctx) {
    switch (ctx.getOutputELFType()) {
        case llvm::ELF::ET_EXEC:
            // when the output file is execution file: e.g. a.out
#ifdef DLINKER
            if (ctx.isDynamic())
                return llvm::make_unique<DynamicRelocationPass>(ctx);
            else
                return std::unique_ptr<Pass>(new StaticRelocationPass(ctx));
#else
            return llvm::make_unique<StaticRelocationPass>(ctx);
#endif // DLINKER
#ifdef DLINKER
        case llvm::ELF::ET_DYN:

```

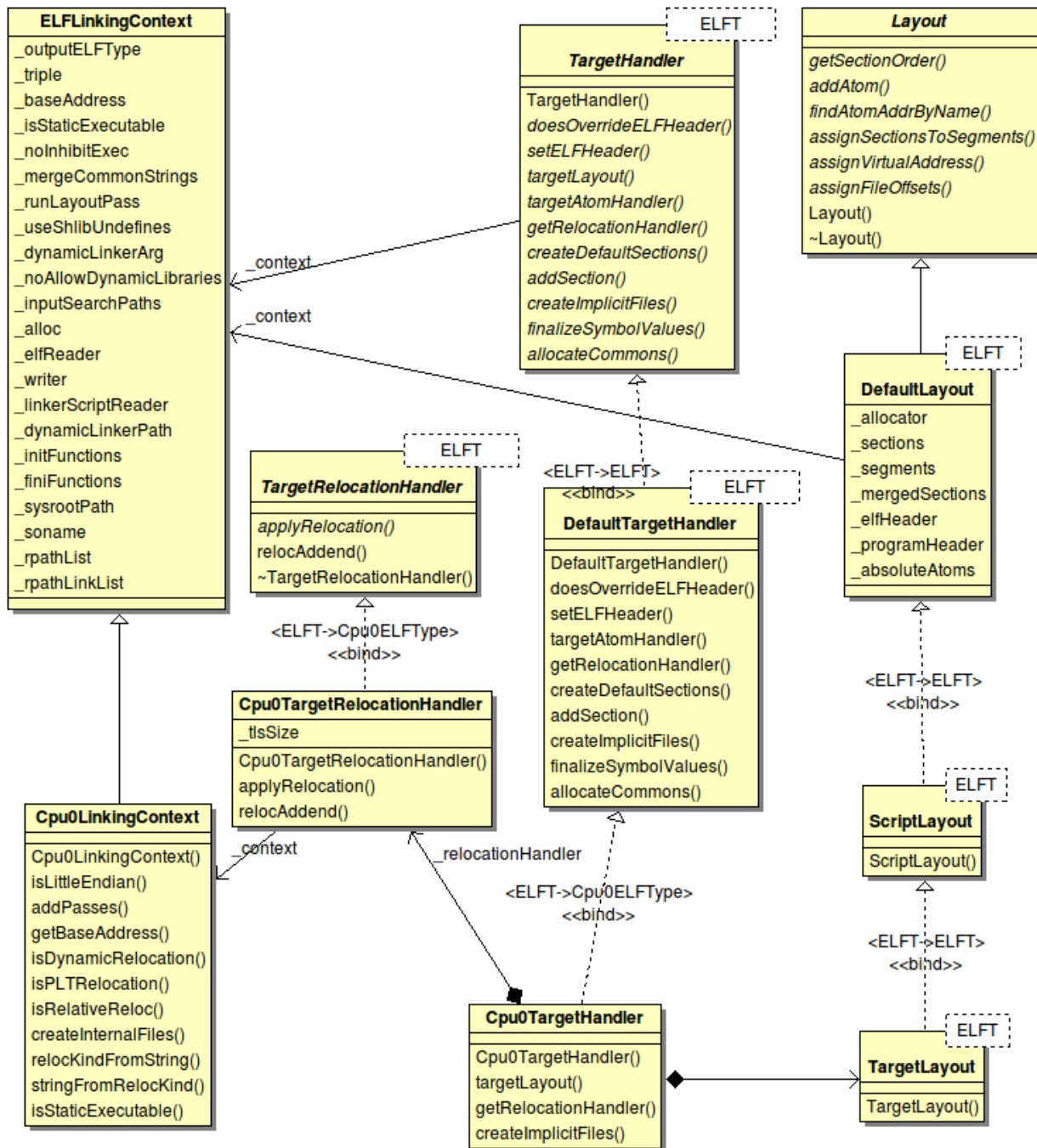


Figure 2.5: Cpu0 lld class relationship

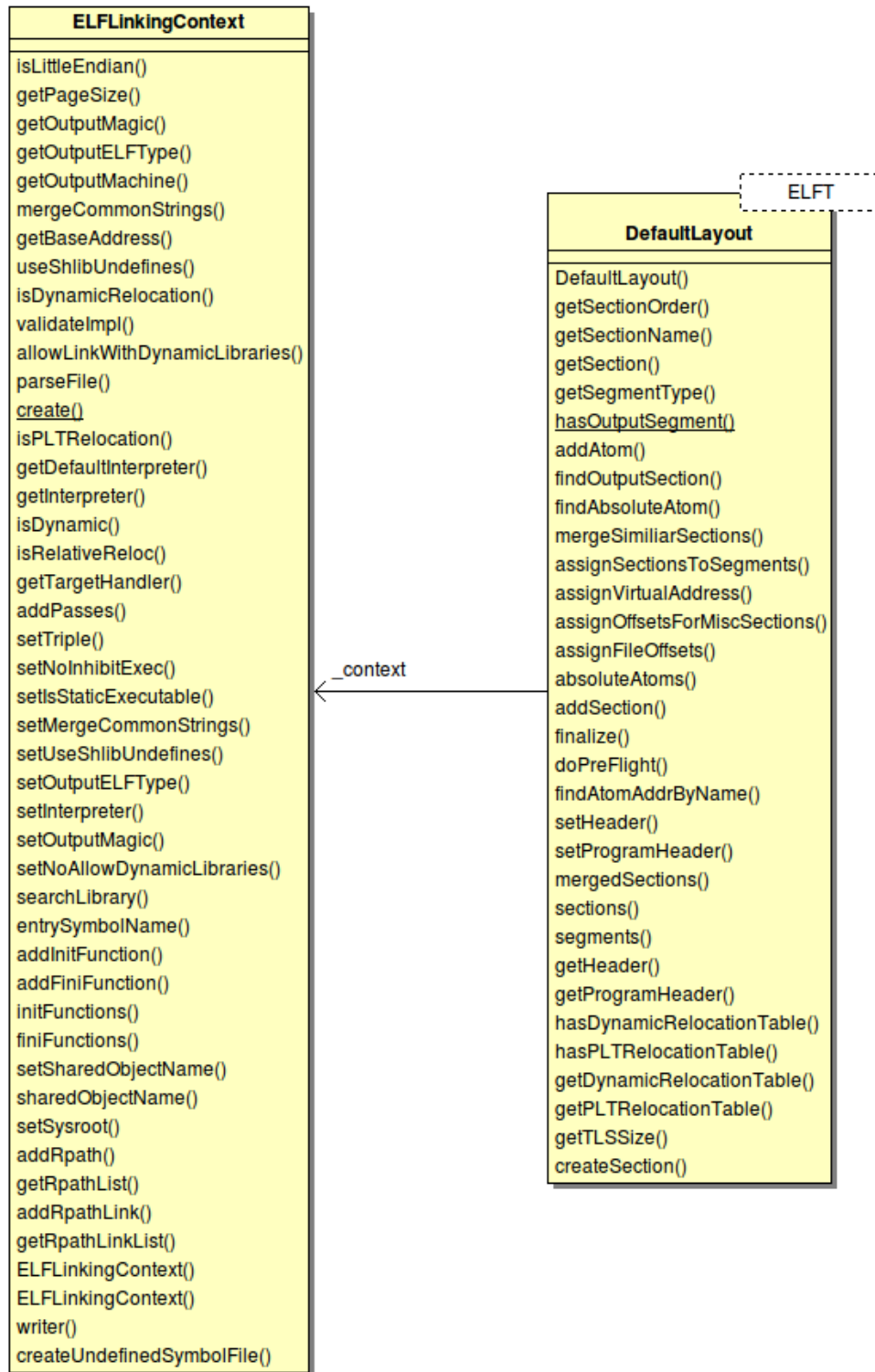


Figure 2.6: Cpu0 lld ELFLinkingContext and DefaultLayout member functions

```
// when the output file is shared object: e.g. foobar.so
    return llvm::make_unique<DynamicRelocationPass>(ctx);
#endif // DLINKER
case llvm::ELF::ET_REL:
    return nullptr;
default:
    llvm_unreachable("Unhandled output file type");
}
```

The “`#ifdef DLINKER`” part is for dynamic linker which will be used in next section. For static linker, a `StaticRelocationPass` object is created and return.

Now the following code of `Cpu0TargetRelocationHandler::applyRelocation()` will be called through `Cpu0TargetHandler` by `lld` ELF driver when it meets each relocation record.

#### `exlbt/lld/lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.cpp`

```
ErrorOr<void> Cpu0TargetRelocationHandler::applyRelocation(
    ELFWriter &writer, llvm::FileOutputBuffer &buf, const lld::AtomLayout &atom,
    const Reference &ref) const {

    switch (ref.kind()) {
    case R_CPU0_NONE:
        break;
    case R_CPU0_HI16:
        relocHI16(location, relocVAddress, targetVAddress, ref.addend());
        break;
    case R_CPU0_LO16:
        relocLO16(location, relocVAddress, targetVAddress, ref.addend());
        break;
    ...
    }
    return error_code::success();
}
```

#### `exlbt/lld/Cpu0/Cpu0TargetHandler.h`

```
class Cpu0TargetHandler LLVM_FINAL
    : public DefaultTargetHandler<Cpu0ELFType> {
public:
    ..
    virtual const Cpu0TargetRelocationHandler &getRelocationHandler() const {
        return _relocationHandler;
    }
}
```

Summary as Figure 2.7.

Remind, static `std::unique_ptr<ELFLinkingContext> ELFLinkingContext::create(llvm::Triple)` is called without an object of class `ELFLinkingContext` instance (because the static keyword). The `Cpu0LinkingContext` constructor will create it's `ELFLinkingContext` part. The `std::unique_ptr` came from c++11 standard. The `unique_ptr` objects automatically delete the object they manage (using a deleter) as soon as themselves are destroyed. Just like the `Singleton` pattern in *Design Pattern* book or *Smart Pointers* in *Effective C++* book.<sup>4</sup>

---

<sup>4</sup> [http://www.cplusplus.com/reference/memory/unique\\_ptr/](http://www.cplusplus.com/reference/memory/unique_ptr/)

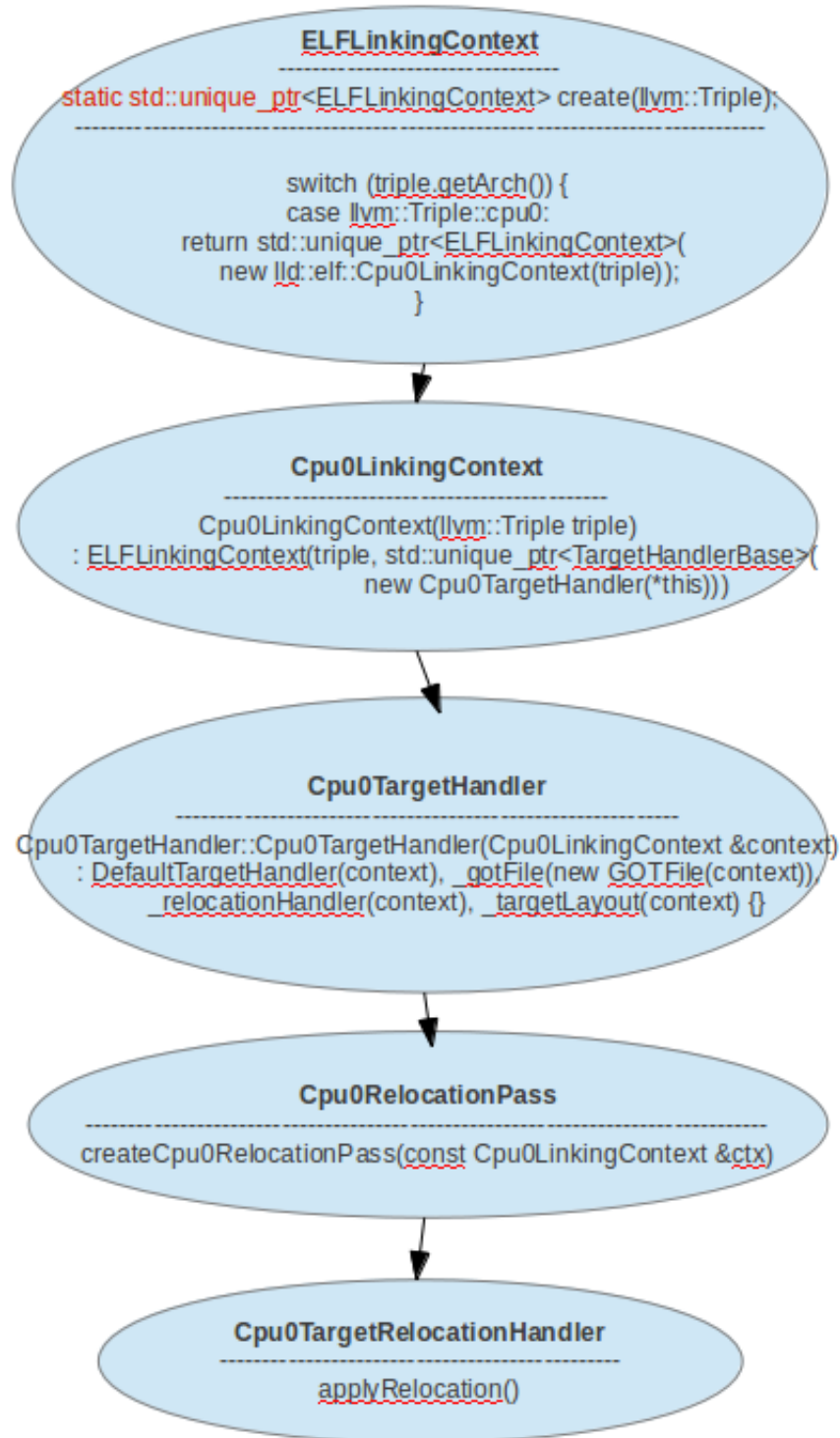


Figure 2.7: Cpu0 lld related objects created sequence

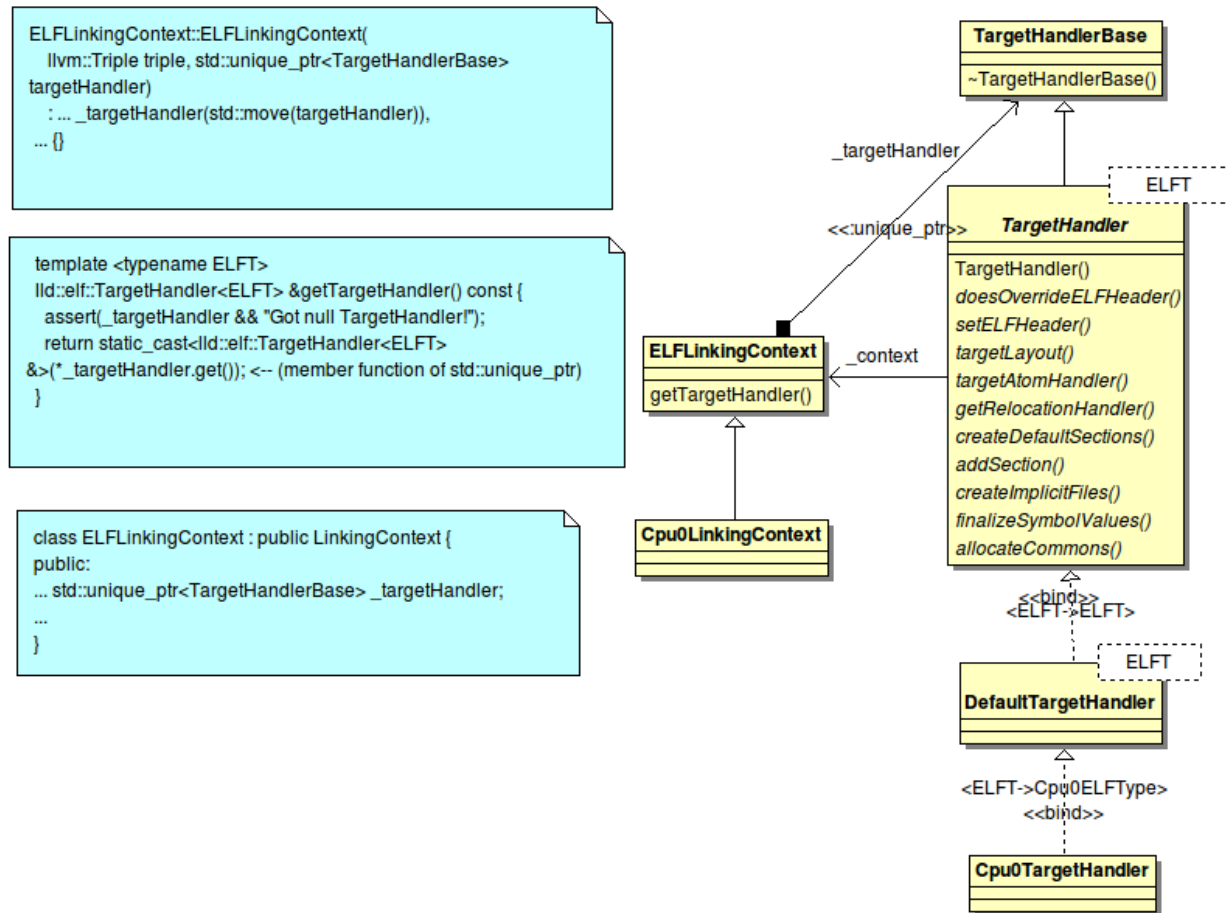


Figure 2.8: Cpu0LinkingContext get Cpu0TargetHandler through &amp;getTargetHandler()



As Figure 2.5 depicted, the Cpu0TargetHandler include the members or pointers which can access to other object. The way to access Cpu0TargetHandler object from Cpu0LinkingContext or Cpu0RelocationHandler rely on LinkingContext::getTargetHandler() function. As Figure 2.8 depicted, the unique\_ptr point to Cpu0TargetHandler will be saved in LinkingContext constructor function.

List the c++11 unique\_ptr::get() and move() which used in Figure 2.8 as follows.

---

**Note:** std::unique\_ptr::get() <sup>5</sup>

pointer get() const noexcept;

Get pointer Returns the stored pointer.

---

**Note:** std::move() <sup>6</sup>

**for example:** std::string bar = "bar-string"; std::move(bar);

bar is null after std::move(bar);

---

## 2.2.5 Dynamic linker

In addition to the lld code with #ifdef DLINKER. The following code in Verilog exists to support dynamic linker.

### libdex/verilog/dynlinker.v

```
'define DLINKER_INFO_ADDR 'h70000
'define GPADDR 'h7FFF0

`ifdef DLINKER
    task setDynLinkerInfo; begin
// below code set memory as follows,
//
//
// DLINKER_INFO_ADDR -----> | numDynEntry |
//
// DLINKER_INFO_ADDR+4 -----> | index of dynsym (0st row) |
// above is the 1st word of section .dynsym of libfoobar.cpu0.so.
// DLINKER_INFO_ADDR+8 -----> | index of dynsym (1st row) |
//
// | ... |
// DLINKER_INFO_ADDR+(numDynEntry-1)*4 -----> | index of dynsym (the last row) |
//
// DLINKER_INFO_ADDR+numDynEntry*4 -----> | 1st function (la()) offset in lib |
// DLINKER_INFO_ADDR+numDynEntry*4+4 -----> | 1st function (la()) name (48 bytes) |
//
// | ... |
// DLINKER_INFO_ADDR+numDynEntry+(numDynEntry-1)*4 ---> | last function (bar()) offset in lib |
// DLINKER_INFO_ADDR+numDynEntry+(numDynEntry-1)*4+4 -> | last function (bar()) name |
//
// DLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52 --> | .dynstr of lib |
//
// | ... |
//
// caculate number of dynamic entries
    numDynEntry = 0;
    j = 0;
```

<sup>5</sup> [http://www.cplusplus.com/reference/memory/unique\\_ptr/get/](http://www.cplusplus.com/reference/memory/unique_ptr/get/)

<sup>6</sup> <http://www.cplusplus.com/reference/utility/move/>

```
for (i=0; i < 384 && j == 0; i=i+52) begin
    if (so_func_offset[i] == 'MEMEMPTY && so_func_offset[i+1] == 'MEMEMPTY &&
        so_func_offset[i+2] == 'MEMEMPTY && so_func_offset[i+3] == 'MEMEMPTY) begin
        numDynEntry = i/52;
        j = 1;
        `ifdef DEBUG_DLINKER
            $display("numDynEntry = %8x", numDynEntry);
        `endif
    end
end
// save number of dynamic entries to memory address `DLINKER_INFO_ADDR
m[`DLINKER_INFO_ADDR] = numDynEntry[31:24];
m[`DLINKER_INFO_ADDR+1] = numDynEntry[23:16];
m[`DLINKER_INFO_ADDR+2] = numDynEntry[15:8];
m[`DLINKER_INFO_ADDR+3] = numDynEntry[7:0];
// copy section .dynsym of ELF to memory address `DLINKER_INFO_ADDR+4
i = `DLINKER_INFO_ADDR+4;
for (j=0; j < (4*numDynEntry); j=j+4) begin
    m[i] = dsym[j];
    m[i+1] = dsym[j+1];
    m[i+2] = dsym[j+2];
    m[i+3] = dsym[j+3];
    i = i + 4;
end
// copy the offset values of section .text of shared library .so of ELF to
// memory address `DLINKER_INFO_ADDR+4+numDynEntry*4
i = `DLINKER_INFO_ADDR+4+numDynEntry*4;
l = 0;
for (j=0; j < numDynEntry; j=j+1) begin
    for (k=0; k < 52; k=k+1) begin
        m[i] = so_func_offset[l];
        i = i + 1;
        l = l + 1;
    end
end
`ifdef DEBUG_DLINKER
    i = `DLINKER_INFO_ADDR+4+numDynEntry*4;
    for (j=0; j < (8*numDynEntry); j=j+8) begin
        $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
        i = i + 8;
    end
`endif
// copy section .dynstr of ELF to memory address
// `DLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52
i=`DLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52;
for (j=0; dstr[j] != 'MEMEMPTY; j=j+1) begin
    m[i] = dstr[j];
    i = i + 1;
end
`ifdef DEBUG_DLINKER
    $display("In setDynLinkerInfo()");
    for (i=`DLINKER_INFO_ADDR; i < `MEMSIZE; i=i+4) begin
        if (m[i] != 'MEMEMPTY || m[i+1] != 'MEMEMPTY ||
            m[i+2] != 'MEMEMPTY || m[i+3] != 'MEMEMPTY)
            $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
    end
    $display("global address %8x", {m[`GPADDR], m[`GPADDR+1],
        m[`GPADDR+2], m[`GPADDR+3]});
```

```

    $display("gp = %8x", gp);
`endif
// below code set memory as follows,
//
// gp -----> | all 0 | (16 bytes)
// gp+16 -----> | 0 |
// gp+16+1*4 -----> | 1st plt entry address | (4 bytes)
// | ... |
// gp+16+(numDynEntry-1)*4 -----> | the last plt entry address |
//
// gpPlt -----> | all 0 | (16 bytes)
// gpPlt+16+0*8'h10 -----> | 32'h10: pointer to plt0 |
// gpPlt+16+1*8'h10 -----> | 1st plt entry |
// gpPlt+16+2*8'h10 -----> | 2nd plt entry |
// | ... |
// gpPlt+16+(numDynEntry-1)*8'h10 --> | the last plt entry |
//
// note: gp point to the _GLOBAL_OFFSET_TABLE_,
// numDynEntry = actual number of functions + 1.
// gp+1*4..gp+numDynEntry*4 set to 8'h10 plt0 which will jump to dynamic
// linker.
// After dynamic linker load function to memory, it will set gp+index*4 to
// function memory address. For example, if the function index is 2, then the
// gp+2*4 is set to the memory address of this loaded function.
// Then the the caller call
// "ld $t9, 2*4($gp)" and "ret $t9" will jump to this loaded function directly.

gpPlt = gp+16+numDynEntry*4;
// set (gpPlt-16..gpPlt-1) to 0
for (j=16; j >= 1; j=j-1)
    m[gpPlt+j] = 8'h00;
// put plt in (gpPlt..gpPlt+numDynEntry*8'h10+1)
for (i=1; i < numDynEntry; i=i+1) begin
    // (gp+8'h10..gp+numDynEntry*8'h10+15) set to plt entry
    // addiu $t9, $zero, dynsym_idx
    m[gpPlt+i*8'h10] = 8'h09;
    m[gpPlt+i*8'h10+1] = 8'h60;
    m[gpPlt+i*8'h10+2] = i[15:8];
    m[gpPlt+i*8'h10+3] = i[7:0];
    // st $t9, 0($gp)
    m[gpPlt+i*8'h10+4] = 8'h02;
    m[gpPlt+i*8'h10+5] = 8'h6b;
    m[gpPlt+i*8'h10+6] = 0;
    m[gpPlt+i*8'h10+7] = 0;
    // ld $t9, ('16h0010)($gp)
    m[gpPlt+i*8'h10+8] = 8'h01;
    m[gpPlt+i*8'h10+9] = 8'h6b;
    m[gpPlt+i*8'h10+10] = 0;
    m[gpPlt+i*8'h10+11] = 8'h10;
    // ret $t9
    m[gpPlt+i*8'h10+12] = 8'h3c;
    m[gpPlt+i*8'h10+13] = 8'h60;
    m[gpPlt+i*8'h10+14] = 0;
    m[gpPlt+i*8'h10+15] = 0;
end

// .got.plt offset(0x00.0x03) has been set to 0 in elf already.
// Set .got.plt offset(8'h10..numDynEntry*8'h10) point to plt entry as above.

```

```
`ifdef DEBUG_DLINKER
    $display("numDynEntry = %8x", numDynEntry);
`endif
//      j32=32'h1fc0; // m[32'h1fc]="something" will hang. Very tricky
m[gp+16] = 8'h0;
m[gp+16+1] = 8'h0;
i=pltAddr[0]+16;
m[gp+16+2] = i[15:8];    // .plt section addr + 16
m[gp+16+3] = i[7:0];

j32=gpPlt+16;
for (i=1; i < numDynEntry; i=i+1) begin
    m[gp+16+i*4] = j32[31:24];
    m[gp+16+i*4+1] = j32[23:16];
    m[gp+16+i*4+2] = j32[15:8];
    m[gp+16+i*4+3] = j32[7:0];
    j32=j32+16;
end
`ifdef DEBUG_DLINKER
    // show (gp..gp+numDynEntry*4-1)
    for (i=0; i < numDynEntry; i=i+1) begin
        $display("%8x: %8x", gp+16+i*4, {m[gp+16+i*4], m[gp+16+i*4+1],
            m[gp+16+i*4+2], m[gp+16+i*4+3]});
    end
    // show (gpPlt..gpPlt+(numDynEntry+1)*8'h10-1)
    for (i=0; i < numDynEntry; i=i+1) begin
        for (j=0; j < 16; j=j+4)
            $display("%8x: %8x", gpPlt+i*8'h10+j,
                {m[gpPlt+i*8'h10+j],
                 m[gpPlt+i*8'h10+j+1],
                 m[gpPlt+i*8'h10+j+2],
                 m[gpPlt+i*8'h10+j+3]});
        end
    `endif
end endtask
`endif

`ifdef DLINKER
task loadToFlash; begin
    // erase memory
    for (i=0; i < `MEMSIZE; i=i+1) begin
        flash[i] = `MEMEMPTY;
    end
    $readmemh("dlconfig/libso.hex", flash);
`ifdef DEBUG_DLINKER
    for (i=0; i < `MEMSIZE && (flash[i] != `MEMEMPTY ||
        flash[i+1] != `MEMEMPTY || flash[i+2] != `MEMEMPTY ||
        flash[i+3] != `MEMEMPTY); i=i+4) begin
        $display("%8x: %8x", i, {flash[i], flash[i+1], flash[i+2], flash[i+3]});
    end
`endif
end endtask
`endif

`ifdef DLINKER
task createDynInfo; begin
    $readmemh("dlconfig/global_offset", globalAddr);
    m[`GPADDR] = globalAddr[0];
end
```

```

m['GPADDR+1] = globalAddr[1];
m['GPADDR+2] = globalAddr[2];
m['GPADDR+3] = globalAddr[3];
gp[31:24] = globalAddr[0];
gp[23:16] = globalAddr[1];
gp[15:8] = globalAddr[2];
gp[7:0] = globalAddr[3];
$readmemh("dlconfig/plt_offset", pltAddr);
`ifdef DEBUG_DLINKER
    $display("global address %8x", {m['GPADDR], m['GPADDR+1],
        m['GPADDR+2], m['GPADDR+3]});
    $display("gp = %8x", gp);
    $display("pltAddr = %8x", pltAddr[0]);
`endif
`endif
`endif
`ifdef DLINKER
    for (i=0; i < 192; i=i+1) begin
        dsym[i] = 'MEMEMPTY;
    end
    for (i=0; i < 96; i=i+1) begin
        dstr[i] = 'MEMEMPTY;
    end
    for (i=0; i < 384; i=i+1) begin
        so_func_offset[i] = 'MEMEMPTY;
    end
    $readmemh("dlconfig/dynsym", dsym);
    $readmemh("dlconfig/dynstr", dstr);
    $readmemh("dlconfig/so_func_offset", so_func_offset);
    setDynLinkerInfo();
end endtask
`endif

```

### lbldex/verilog/flashio.v

```

`define FLASHADDR 'ha0000

`ifdef DLINKER
    end else if (abus >= 'FLASHADDR && abus <= 'FLASHADDR+'MEMSIZE-4) begin
        fabus = abus-'FLASHADDR;
        if (en == 1 && rw == 0) begin // r_w==0:write
            data = dbus_in;
            case (m_size)
                `BYTE: {flash[fabus]} = dbus_in[7:0];
                `INT16: {flash[fabus], flash[fabus+1]} = dbus_in[15:0];
                `INT24: {flash[fabus], flash[fabus+1], flash[fabus+2]} = dbus_in[24:0];
                `INT32: {flash[fabus], flash[fabus+1], flash[fabus+2], flash[fabus+3]}
                    = dbus_in;
            endcase
        end else if (en == 1 && rw == 1) begin // r_w==1:read
            case (m_size)
                `BYTE: data = {8'h00, 8'h00, 8'h00, flash[fabus]};
                `INT16: data = {8'h00, 8'h00, flash[fabus], flash[fabus+1]};
                `INT24: data = {8'h00, flash[fabus], flash[fabus+1], flash[fabus+2]};
                `INT32: data = {flash[fabus], flash[fabus+1], flash[fabus+2],
                    flash[fabus+3]};
            endcase
        end else

```

```
        data = 32'hZZZZZZZZ;  
'endif
```

### lbdex/verilog/cpu0ld.v

```
'define DLINKER // Dynamic Linker Support  
// 'define DEBUG_DLINKER // Dynamic Linker Debug  
// TRACE: Display the memory contents of the loaded program and data  
// 'define TRACE  
  
'include "cpu0.v"
```

### lbdex/verilog/cpu0lld.v

```
'define CPU0II  
'define DLINKER // Dynamic Linker Support  
// 'define DEBUG_DLINKER // Dynamic Linker Debug  
// TRACE: Display the memory contents of the loaded program and data  
// 'define TRACE  
  
'include "cpu0.v"
```

The following code `ch_dynamiclinker.cpp` and `foobar.cpp` is the example for dynamic linker demonstration. File `dynamic_linker.cpp` is what our implementation to execute the dynamic linker function on Cpu0 Verilog machine.

### exlbt/input/debug.h

```
#ifndef _DEBUG_H_  
#define _DEBUG_H_  
  
#define STOP \  
    asm("lui $t9, 0xffff"); \  
    asm("addiu $t9, $zero, 0xffff"); \  
    asm("ret $t9");  
  
#define ENABLE_TRACE \  
    asm("ori $sw, $sw, 0x0020");  
  
#define DISABLE_TRACE \  
    asm("lui $at, 0xffff"); \  
    asm("ori $at, $at, 0xffdf"); \  
    asm("and $sw, $sw, $at"); // clear 'D  
  
#define SET_OVERFLOW \  
    asm("ori $sw, $sw, 0x008");  
  
#define CLEAR_OVERFLOW \  
    asm("lui $7, 0xffff"); \  
    asm("ori $7, $7, 0xffff7"); \  
    asm("and $sw, $sw, $7"); // clear 'V  
  
#define SET_SOFTWARE_INT \  
    asm("ori $sw, $sw, 0x4000");
```

```

#define CLEAR_SOFTWARE_INT \
    asm("lui $7, 0xffff"); \
    asm("ori $7, $7, 0xbfff"); \
    asm("and $sw, $sw, $7");

#define SAVE_REGISTERS \
    asm("lui $at, 7"); \
    asm("ori $at, $at, 0xff00"); \
    asm("st $2, 0($at)"); \
    asm("st $3, 4($at)"); \
    asm("st $4, 8($at)"); \
    asm("st $5, 12($at)"); \
    asm("st $t9, 16($at)"); \
    asm("st $7, 20($at)"); \
    asm("st $8, 24($at)"); \
    asm("st $9, 28($at)"); \
    asm("st $10, 32($at)"); \
    asm("st $gp, 36($at)"); \
    asm("st $12, 40($at)"); \
    asm("st $13, 44($at)");

#define RESTORE_REGISTERS \
    asm("lui $at, 7"); \
    asm("ori $at, $at, 0xff00"); \
    asm("ld $2, 0($at)"); \
    asm("ld $3, 4($at)"); \
    asm("ld $4, 8($at)"); \
    asm("ld $5, 12($at)"); \
    asm("ld $t9, 16($at)"); \
    asm("ld $7, 20($at)"); \
    asm("ld $8, 24($at)"); \
    asm("ld $9, 28($at)"); \
    asm("ld $10, 32($at)"); \
    asm("ld $gp, 36($at)"); \
    asm("ld $12, 40($at)"); \
    asm("ld $13, 44($at)");

#define OVERFLOW 0x8
#define INT 0x2000
#define SOFTWARE_INT 0x4000
#define INT1 0x8000
#define INT2 0x10000

extern void int_sim();

#endif

```

### exlbt/input/dynamic\_linker.h

```

#ifndef __DYNAMIC_LINKER_H_
#define __DYNAMIC_LINKER_H_

#define DYNLINKER_INFO_ADDR 0x70000
#define DYNENT_SIZE 4
#define DYNPROGSTART 0x40000
#define FLASHADDR 0xA0000

```

```
#include "dynamic_linker.h"

// #define DEBUG_DLINKER
#define PLTOADDR 0x10
#define REGADDR 0x7ff00

extern "C" int printf(const char *format, ...);

int got_plt_fill[0x80] = {
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
};

int progCounter = 0; // program counter, init to 0 in main()

ProgAddr prog[10];

void dynamic_linker()
{
    SAVE_REGISTERS;
    // static ProgAddr prog[10]; // has side effect (ProgAddr c
    // Virtual Box on iMac).

    int i = 0;
    int nextFreeAddr;
    int *src, *dest, *end;
    int numDynEntry = 0;
    int dynsym_idx = 0;
    int dynsym = 0;
    char *dynstr = 0;
    int libOffset = 0;
    int nextFunLibOffset = 0;
    volatile int memAddr = 0;
    numDynEntry = *((int*) (DYNLINKER_INFO_ADDR));
    int gp = *(int*) GPADDR;
#ifdef DEBUG_DLINKER
```



```

    printf("gp = %d\n", gp);
#endif
    dynsym_idx = *(int*)gp;
#ifdef DEBUG_DLINKER
    printf("numDynEntry = %d, dynsym_idx = %d\n", numDynEntry, dynsym_idx);
#endif
    dynsym = *(int*) ((DYNLINKER_INFO_ADDR+4)+(dynsym_idx*DYNENT_SIZE));
    dynstr = (char*) (DYNLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52+dynsym);
    libOffset = *((int*) (DYNLINKER_INFO_ADDR+4+numDynEntry*4+(dynsym_idx-1)*52));
    for (i = dynsym_idx; i < numDynEntry; i++) {
        nextFunLibOffset = *((int*) (DYNLINKER_INFO_ADDR+4+numDynEntry*4+i*52));
        if (libOffset != nextFunLibOffset)
            break;
    }
#ifdef DEBUG_DLINKER
    printf("address of dstr = %x, dynsym = %d, dstr = %s\n",
        (int)dynstr, dynsym, dynstr);
    printf("libOffset = %d, nextFunLibOffset = %d, progCounter = %d\n",
        libOffset, nextFunLibOffset, progCounter);
#endif
    if (progCounter == 0)
        nextFreeAddr = DYNPROGSTART;
    else
        nextFreeAddr = prog[progCounter-1].memAddr+prog[progCounter-1].size;
    prog[progCounter].memAddr = nextFreeAddr;
    prog[progCounter].size = (nextFunLibOffset - libOffset);

#ifdef DEBUG_DLINKER
    printf("prog[progCounter].memAddr = %d, prog[progCounter].size = %d\n",
        prog[progCounter].memAddr, (unsigned int) (prog[progCounter].size));
#endif
    // Load program from (FLASHADDR+libOffset..FLASHADDR+nextFunLibOffset-1) to
    // (nextFreeAddr..nextFreeAddr+prog[progCounter].size-1)
    src = (int*) (FLASHADDR+libOffset);
    end = (int*) (src+prog[progCounter].size/4);
#ifdef DEBUG_DLINKER
    printf("end = %x, src = %x, nextFreeAddr = %x\n",
        (unsigned int)end, (unsigned int)src, (unsigned int)nextFreeAddr);
    printf("*src = %x\n", (unsigned int)(*src));
#endif
    printf("loading %s...\n", dynstr);
    for (dest = (int*) (prog[progCounter].memAddr); src < end; src++, dest++) {
        *dest = *src;
#ifdef DEBUG_DLINKER
        printf("*dest = %08x\n", (unsigned int)(*dest));
#endif
    }
    progCounter++;

#ifdef DEBUG_DLINKER
    printf("progCounter-1 = %x, prog[progCounter-1].memAddr = %x, \
        *prog[progCounter-1].memAddr = %x\n",
        (unsigned int) (progCounter-1), (unsigned int) (prog[progCounter-1].memAddr),
        *(unsigned int*) (prog[progCounter-1].memAddr));
#endif
    // Change .got.plt for "ld      $t9, idx($gp)"
    *((int*) (gp+0x10+dynsym_idx*0x04)) = prog[progCounter-1].memAddr;
    *((int*) (0x7FFE0)) = prog[progCounter-1].memAddr;

```

```
#ifdef DEBUG_DLINKER
    printf("*((int*) (gp+0x10+dynsym_idx*0x10)) = %x, *((int*) (0x7FFE0) = %x\n",
        *((int*) (gp+0x10+dynsym_idx*0x10)), (unsigned int) (*((int*) (0x7FFE0))));
    printf("*((int*) (gp+0x04)) = %x, *((int*) (gp+0x08)) = %x, *((int*) (gp+0x0c)) = %x\n",
        *((int*) (gp+0x04)), *((int*) (gp+0x08)), *((int*) (gp+0x0c)));
#endif
printf("run %s...\n", dynstr);
RESTORE_REGISTERS;

// restore $lr. The next instruction of foo() of main.cpp for the main.cpp
// call foo() first time example.
// The $lr, $fp and $sp saved in cpu0Plt0AtomContent of Cpu0LinkingContext.cpp.
asm("ld $lr, 4($gp)"); // restore $lr
#ifdef DEBUG_DLINKER
    ENABLE_TRACE;
#endif
asm("ld $fp, 8($gp)"); // restore $fp
asm("ld $sp, 12($gp)"); // restore $sp
#ifdef DEBUG_DLINKER
    DISABLE_TRACE;
#endif
// jmp to the dynamic linked function. It's foo() for the
// caller, ch_dynamic_linker.cpp, call foo()
// first time example.
asm("lui $t9, 0x7");
asm("ori $t9, $t9, 0x7FFE0");
asm("ld $t9, 0($t9)");
asm("ret $t9");

return;
}
```

### exlbt/input/ch\_dynamiclinker.cpp

```
#include "dynamic_linker.h"
#include "print.h"

extern "C" int printf(const char *format, ...);

extern int la(int x1, int x2);
extern int foo(int x1, int x2);
extern int bar();

int main()
{
    // ENABLE_TRACE;
    int a = 0;

    #if 1
        a = foo(1, 2);
        printf("foo(1, 2) = %d\n", a);
    #endif
    #if 1
        a = bar();
        printf("bar() = %d\n", a);
    #endif
}
```

```
#if 0
    a = foo(1, 2);
    printf("foo(1, 2) = %d\n", a);
#endif

    return 0;
}
```

#### exlbt/input/foobar.cpp

```
#include "dynamic_linker.h"

int la(int x1, int x2)
{
    int sum = x1 + x1 + x2;

    return sum;
}

int foo(int x1, int x2)
{
    int sum = x1 + x2;

    return sum;
}

#if 0
int factorial(int x)
{
    if (x > 0)
        return x*factorial(x-1);
    else
        return 1;
}
#endif

int bar()
{
    int a;
    // ENABLE_TRACE;
    a = foo(2, 2);
    a += la(2, 3); // 4+7=11
    // a += factorial(4); // 11+24=35

    return a;
}
```

#### exlbt/input/build-dlinker.sh

```
#!/usr/bin/env bash

source functions.sh

sh_name=build-dlinker.sh
argNum=$#
```

```
arg1=$1
arg2=

prologue;

rm -rf dlconfig
mkdir dlconfig

${CLANGDIR}/clang -target mips-unknown-linux-gnu -c start.cpp -emit-llvm -o start.bc
${CLANGDIR}/clang -target mips-unknown-linux-gnu -c debug.cpp -emit-llvm -o debug.bc
${CLANGDIR}/clang -target mips-unknown-linux-gnu -c dynamic_linker.cpp -emit-llvm \
-o dynamic_linker.cpu0.bc
${CLANGDIR}/clang -target mips-unknown-linux-gnu -c printf-stdarg-def.c -emit-llvm \
-o printf-stdarg-def.bc
${CLANGDIR}/clang -target mips-unknown-linux-gnu -c printf-stdarg.c -emit-llvm \
-o printf-stdarg.bc
${CLANGDIR}/clang -target mips-unknown-linux-gnu -c ch_dynamiclinker.cpp -emit-llvm \
-o ch_dynamiclinker.cpu0.bc
${CLANGDIR}/clang -target mips-unknown-linux-gnu -c foobar.cpp -emit-llvm -o foobar.cpu0.bc
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static -filetype=obj \
-cpu0-reserve-gp=true dynamic_linker.cpu0.bc -o dynamic_linker.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static -filetype=obj \
printf-stdarg-def.bc -o printf-stdarg-def.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static -filetype=obj \
-cpu0-reserve-gp=true printf-stdarg.bc -o printf-stdarg.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=pic -filetype=obj \
-cpu0-reserve-gp=true -cpu0-no-cpload=true foobar.cpu0.bc -o foobar.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static -filetype=obj \
lib_cpu0.ll -o lib_cpu0.o
${TOOLDIR}/lld -flavor gnu -target cpu0-unknown-linux-gnu -shared -o \
libfoobar.cpu0.so foobar.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static -filetype=obj \
-cpu0-reserve-gp=true start.bc -o start.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static \
-filetype=obj debug.bc -o debug.cpu0.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${CPU} -relocation-model=static -filetype=obj \
-cpu0-reserve-gp=true ch_dynamiclinker.cpu0.bc -o ch_dynamiclinker.cpu0.o
${TOOLDIR}/lld -flavor gnu -target cpu0-unknown-linux-gnu -o a.out \
start.cpu0.o \
printf-stdarg-def.cpu0.o printf-stdarg.cpu0.o dynamic_linker.cpu0.o \
ch_dynamiclinker.cpu0.o libfoobar.cpu0.so lib_cpu0.o debug.cpu0.o
${TOOLDIR}/llvm-objdump -elf2hex -le=false -cpu0dumpso libfoobar.cpu0.so \
> dlconfig/libso.hex
${TOOLDIR}/llvm-objdump -elf2hex -le=false -cpu0linkso a.out > cpu0.hex
cp -rf dlconfig cpu0.hex ../../lbdex/verilog/
echo "0 /* 0: big endian, 1: little endian */" > ../../lbdex/verilog/cpu0.config
cat ../../lbdex/verilog/cpu0.config
```

## Run

```
1-160-136-173 input Jonathan$ pwd
/Users/Jonathan/test/lbt/exlbt/input
1-160-136-173:input Jonathan$ bash build-dlinker.sh cpu032II
1-160-136-173:input Jonathan$ cd ../../lbdex/verilog/
1-160-136-173:verilog Jonathan$ pwd
/Users/Jonathan/test/lbt/lbdex/verilog
1-160-136-173:verilog Jonathan$ iverilog -o cpu0IIId cpu0IIId.v
```

```

1-160-136-173:verilog Jonathan$ ls
clean.sh  cpu0IId  cpu0Id.v  cpu0IId.v  cpu0IIs.v  cpu0Is.v  cpu0.v  dynlinker.v
flashio.v
1-160-136-173:verilog Jonathan$ ./cpu0IId
WARNING: ./cpu0.v:371: $readmemh(cpu0.hex): Not enough words in the file for
the requested range [0:524287].
WARNING: ./dynlinker.v:185: $readmemh(libso.hex): Not enough words in the
file for the requested range [0:524287].
WARNING: ./dynlinker.v:223: $readmemh(dynsym): Not enough words in the file
for the requested range [0:191].
WARNING: ./dynlinker.v:224: $readmemh(dynstr): Not enough words in the file
for the requested range [0:95].
WARNING: ./dynlinker.v:225: $readmemh(so_func_offset): Not enough words in
the file for the requested range [0:383].
numDynEntry = 00000005
taskInterrupt(001)
loading _Z3fooii...
run _Z3fooii...
foo(1, 2) = 3
loading _Z3barv...
run _Z3barv...
loading _Z2laii...
run _Z2laii...
bar() = 11
...

```

The “`#ifdef DEBUG_DLINKER`” part of code in `dynamic_linker.cpp` is for debugging purpose (since we coding it and take time to debug). After skip these debug code, the `dynamic_linker.cpp` is short and not difficult to read.

The run result is under expectation. The `main()` call `foo()` function first. Function `foo()` is loaded by dynamic linker (`dynamic_linker.cpp`) from flash address `FLASHADDR` (defined in `dynamic_linker.h`) to memory. The `flashio.v` implement the simulation read from flash address. After loaded `foo()` body from flash, `dynamic_linker.cpp` jump to this loaded address by “`ret $t9`” instruction.

Same as static linker, you can generate `slt` instruction instead of `cmp` by change from `cpu=cpu0I` to `cpu0=cpu0II` in `build-dlinker.sh` and run it again to get the same result. Finally, since the dynamic linker is a demo implemenation, it supports big endian only.

## How to work

After run `build-dlinker.sh`, the following files are created.

### lbdex/verilog/cpu0.hex

```

/*Disassembly of section .plt:*/
/*..PLT0:*/
/*      0:*/ 36 00 00 3c          /* jmp      60*/
/*      4:*/ 36 00 00 04          /* jmp      4*/
/*      8:*/ 36 00 00 04          /* jmp      4*/
/*      c:*/ 36 ff ff fc          /* jmp      -4*/

/*..PLT0:*/
/*     10:*/ 02 eb 00 04          /* st       $lr, 4($gp)*/
/*     14:*/ 02 cb 00 08          /* st       $fp, 8($gp)*/
/*     18:*/ 02 db 00 0c          /* st       $sp, 12($gp)*/
/*     1c:*/ 36 00 09 b8          /* jmp      2488*/

```

```

/*__plt__Z3barv:*/
/*      20:*/ 01 6b 00 24          /* ld      $t9, 36($gp)
/*      24:*/ 3c 60 00 00          /* ret     $t9*/
/*      28:*/ 00 00 00 00          /* nop*/
/*      2c:*/ 00 00 00 00          /* nop*/

/*__plt__Z3fooi:*/
/*      30:*/ 01 6b 00 1c          /* ld      $t9, 28($gp)
/*      34:*/ 3c 60 00 00          /* ret     $t9*/
/*      38:*/ 00 00 00 00          /* nop*/
/*      3c:*/ 00 00 00 00          /* nop*/
...

/*main:*/
...
/*      d68:*/ 3b ff f2 b4          /* jsub    16773812*/ // call foo()
...
/*      d80:*/ 3b ff f3 28          /* jsub    16773928*/ // call printf()
/*      d84:*/ 3b ff f2 a8          /* jsub    16773800*/ // call bar()
...
/*      d9c:*/ 3b ff f3 0c          /* jsub    16773900*/ // call printf()
...
/*      db8:*/ 3c e0 00 00          /* ret     $lr*/
...
/*Contents of section .data:*/
/*20a8 */00 00 00 01 00 00 00 01 00 00 00 01 00 00 00 01 /* .....*/
...

```

## lbdt/verilog/dynstr

```

00 5f 5f 74 6c 73 5f 67 65 74 5f 61 64 64 72 00 5f 5a 32 6c 61 69 69 00 5f 5a
35 70 6f 77 65 72 69 00 5f 5a 33 66 6f 6f 69 69 00 5f 5a 33 62 61 72 76 00 5f
47 4c 4f 42 41 4c 5f 4f 46 46 53 45 54 5f 54 41 42 4c 45 5f 00 5f 44 59 4e 41
4d 49 43 00

```

## exlbt/verilog/dynsym

```

00 00 00 00 00 00 00 01 00 00 00 10 00 00 00 18 00 00 00 22 00 00 00 2b 00 00
00 33 00 00 00 49

```

## lbdt/verilog/global\_offset

```

00 00 20 68

```

## exlbt/input/num\_dyn\_entry

6

**exlbt/input/libfoobar.cpu0.so**

```
1-160-136-173:input Jonathan$ ~/llvm/test/cmake_debug_build/Debug/bin/
llvm-objdump -s libfoobar.cpu0.so
```

```
libfoobar.cpu0.so:      file format ELF32-CPU0

Contents of section :
...
Contents of section .dynsym:
 00e4 00000000 00000000 00000000 00000000 .....
 00f4 00000001 0000019c 00000000 12000004 .....
 0104 00000010 0000019c 0000003c 12000004 .....<....
 0114 00000018 000001d8 00000038 12000004 .....8....
 0124 00000021 00000210 00000070 12000004 ...!.....p....
 0134 00000029 00001040 00000000 10000006 ...)...@.....
 0144 0000003f 00001040 00000000 11000005 ...?...@.....

Contents of section .dynstr:
 0154 005f5f74 6c735f67 65745f61 64647200 .__tls_get_addr.
 0164 5f5a326c 61696900 5f5a3366 6f6f6969 _Z21aii._Z3fooi
 0174 005f5a33 62617276 005f474c 4f42414c ._Z3barv._GLOBAL
 0184 5f4f4646 5345545f 5441424c 455f005f _OFFSET_TABLE._
 0194 44594e41 4d494300                DYNAMIC.
```

**exlbt/input/a.out**

```
1-160-136-173:input Jonathan$ ~/llvm/test/cmake_debug_build/Debug/bin/
llvm-objdump -s a.out
```

```
a.out:      file format ELF32-CPU0

Contents of section :
...
Contents of section .dynsym:
 013c 00000000 00000000 00000000 00000000 .....
 014c 00000001 00000000 00000000 12000000 .....
 015c 0000000a 00000000 00000000 12000000 .....

Contents of section .dynstr:
 016c 005f5a33 666f6f69 69005f5a 33626172 ._Z3fooi._Z3bar
 017c 76006c69 62666f6f 6261722e 63707530 v.libfoobar.cpu0
 018c 2e736f00                .so.
...
Contents of section .got.plt:
 2068 00000000 00000000 00000000 00000000 .....
 2078 00000000 00000000 00000000 00000000 .....
 2088 000001d0 00000000 00000000 00000000 .....
 2098 000001e0 00000000 00000000 00000000 .....

Contents of section .data:
 20a8 00000001 00000001 00000001 00000001 .....
```

File `dynstr` is section `.dynstr` of `libfoobar.cpu0.so`. File `dynsym` is the first 4 bytes of every entry of `.dynsym`. File `global_offset` contains the start address of section `.got.plt`.

The code of `dylinker.v` will set the memory as follows after program is loaded. (`gp` value below is 2068 came from file `global_offset`).

## memory contents

```
// -----
// gp -----> | all 0 | (16 bytes)
// gp+16 -----> | 0 |
// gp+16+1*4 -----> | 1st plt entry address | (4 bytes)
// | ... |
// gp+16+(numDynEntry-1)*4 -----> | the last plt entry address |
// -----
// gpPlt -----> | all 0 | (16 bytes)
// gpPlt+16+0*8'h10 -----> | 32'h10: pointer to plt0 |
// gpPlt+16+1*8'h10 -----> | 1st plt entry |
// gpPlt+16+2*8'h10 -----> | 2nd plt entry |
// | ... |
// gpPlt+16+(numDynEntry-1)*8'h10 --> | the last plt entry |
// -----
```

For example code of `ch_dynamiclinker.cpp` and `foobar.cpp`, `gp` is 2068, `numDynEntry` is the contents of file `num_dyn_entry` which is 6. Every plt entry above (memory address `gp+16+1*8'h10..gp+16+(numDynEntry-1)*8'h10`) is initialized to “`addiu $t9, $zero, 4($gp); st $t9, 0($gp); ld $t9, 16($gp); ret $t9`” as follows,

## memory contents

```
// -----
// gp -----> | all 0 | (16 bytes)
// gp+16 -----> | 0 |
// gp+16+1*4 -----> | 1st plt entry address | (4 bytes)
// | ... |
// gp+16+(numDynEntry-1)*4 -----> | the last plt entry address |
// -----
// gpPlt -----> | all 0 | (16 bytes)
// gpPlt+16+0*8'h10 -----> | 32'h10: pointer to plt0 |
// gpPlt+16+1*8'h10 -----> | addiu $t9, $zero, 4 |
// | st $t9, 0($gp) |
// | ld $t9, 16($gp) |
// | ret $t9 |
// gpPlt+16+2*8'h10 -----> | addiu $t9, $zero, 4 |
// | st $t9, 0($gp) |
// | ld $t9, 16($gp) |
// | ret $t9 |
// ... | ... |
// gpPlt+16+(6-1)*8'h10 -----> | addiu $t9, $zero, 4 |
// | st $t9, 0($gp) |
// | ld $t9, 16($gp) |
// | ret $t9 |
// -----
```

Figure 2.9 is the memory contents after the example program is loaded.

Figure 2.10 is the Control flow transfer from `call foo()` of `main()` to dynamic linker. After the first time of `ch_dynamiclinker.cpp` call `foo()`, it jump to `__plt_Z3fooi` plt entry. In `__plt_Z3fooi`, “`ld $t9, 1c($gp)`” and “`ret $t9`” will jump to “Plt foo:”. Since `foo` is the 3rd plt entry in “Plt foo:”, it save 3 to `0($gp)` memory address then jump to `PLT0`. The purpose of `PLT0` is to save `$lr`, `$fp`, `$sp` and jump to dynamic linker. Now, the control flow transfers to dynamic linker. Dynamic linker will get the loaded function name and function offset of shared library by the value of `0($gp)` which is 3 now (set in “Plt foo:”). The value 3 tells dynamic linker loading `foo()` (3rd string in `.dynstr`) from offset of shared library, `0x3c` (3rd value of Function offset area in Figure). Now, dynamic linker can load `foo()`



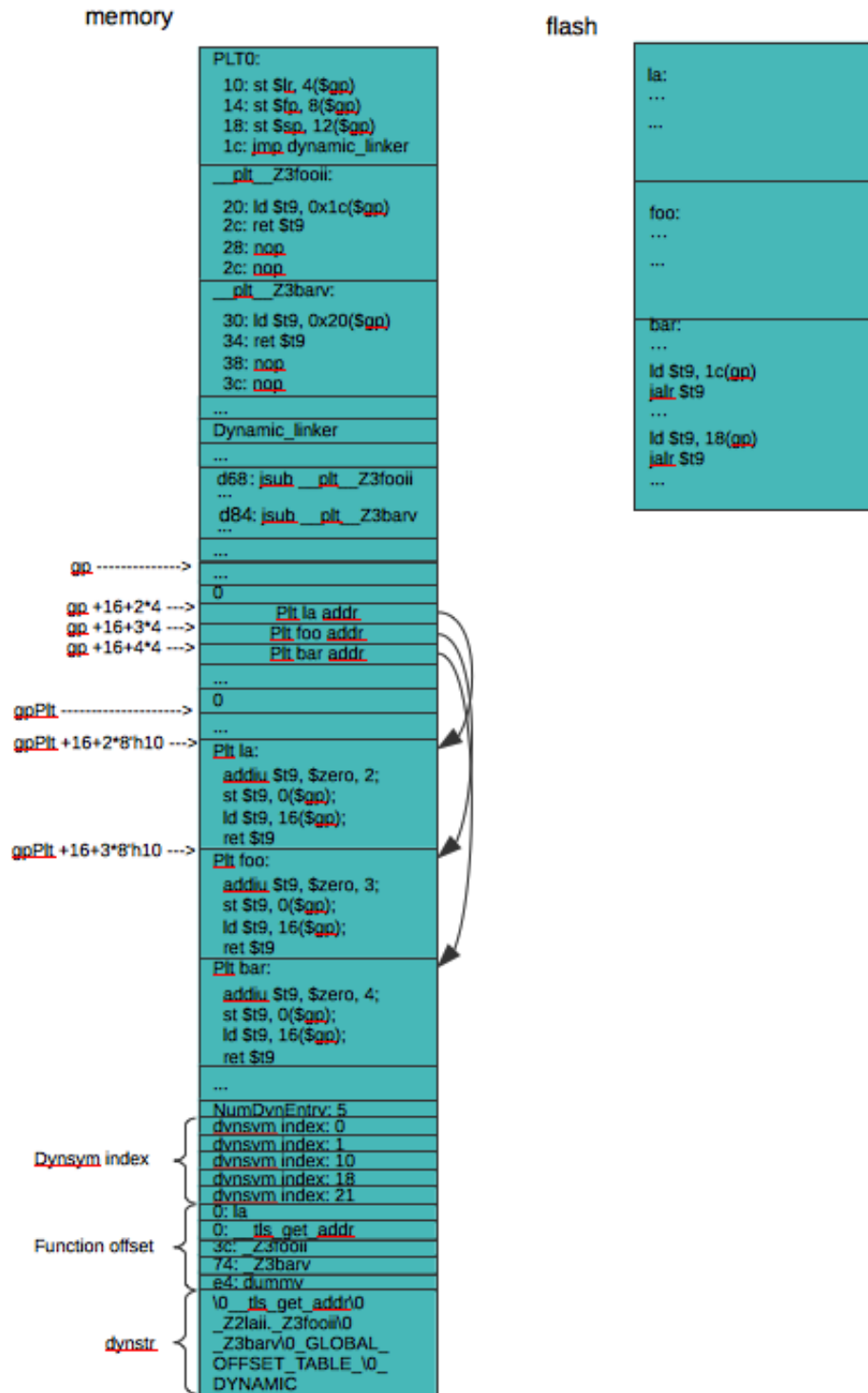


Figure 2.9: Memory contents after the program is loaded

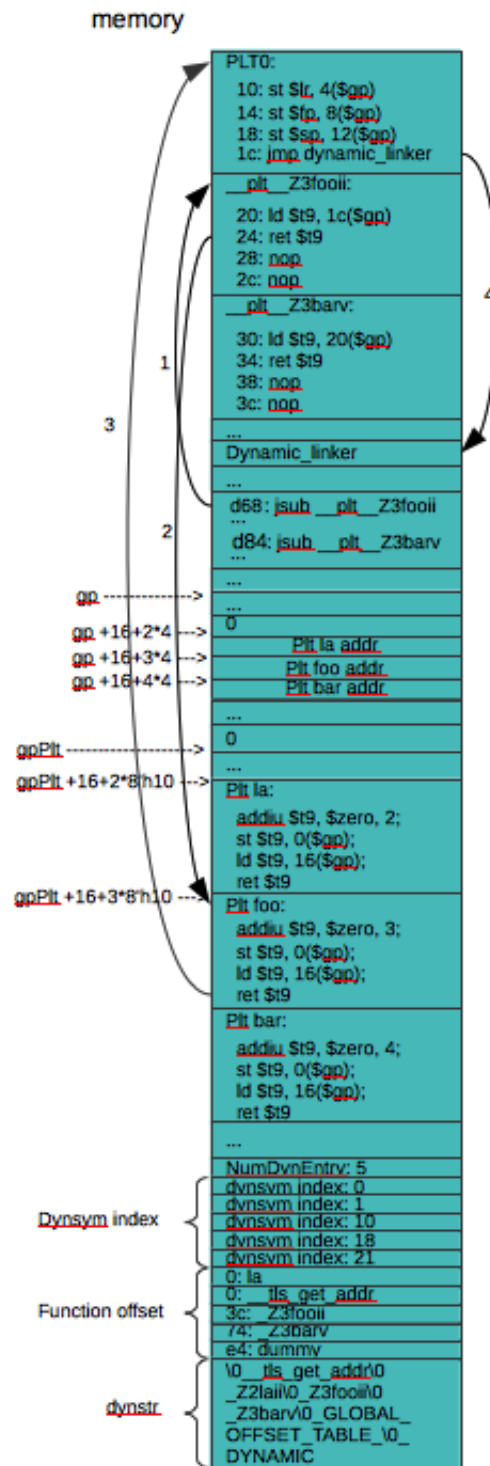


Figure 2.10: Control flow transfer from calling `foo()` instruction of `main()` to dynamic linker

function from flash to memory, set the address  $gp+3*4$  to 0x40000 where the address 0x40000 is the memory address of `foo()` function loaded to, and then prepare jump to the `foo()` memory address. Remind we say the prepare jump to `foo()`. Because before jump to `foo()`, dynamic linker needs to restore the `$lr`, `$fp`, `$sp` to the value of just before caller calling `foo()` (they are saved in 4, 8, 12 of `$gp` offset in PLT0, so them can be restored from that address).

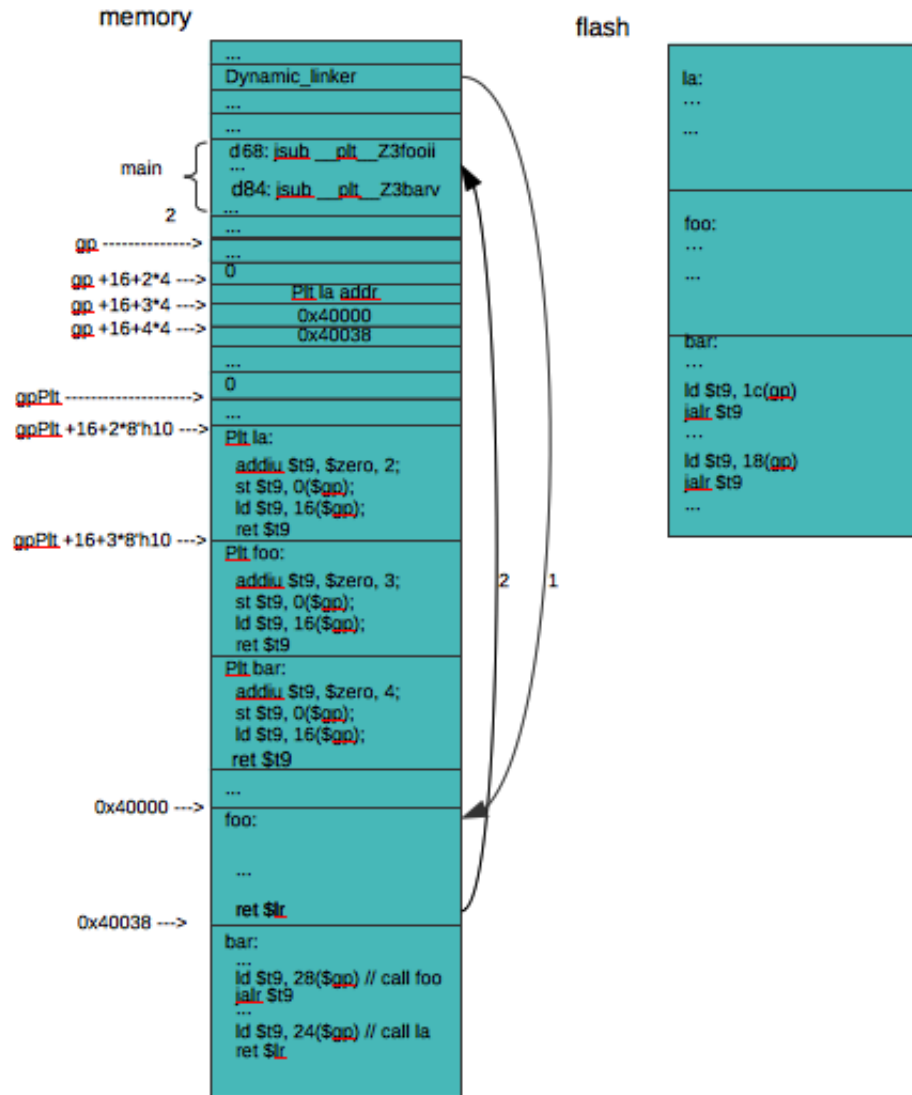


Figure 2.11: Transfer from dynamic linker to `foo()` and back to `main()`

As Figure 2.11 depicted, control flow from dynamic linker to `foo()` and back to caller `main()` when it meets the instruction “`ret $lr`” in `foo()`.

Now the program run at the next instruction of call `foo()` in `main()` as Figure 2.12 depicted. When it runs to address 0xd8 “`jsub __plt_Z3barv`”, the control flow will transfer from `main` through `__plt_Z3barv`, “`Plt bar:`” and PLT0 to dynamic linker as Figure 2.12 depicted. Then load and run `bar()` from flash to memory just like the calling `__plt_Z3fooi` as Figure 2.13 depicted. The difference is `bar()` will call `foo()` first and call `la()` next. The call `foo()` in `bar()` will jump to `foo()` directly as Figure 2.12 because the content of `gp+28` is the address of 0x40000 which set in dynamic linker when the first time of `foo()` function is called.

Finally when `bar()` call `la()` function it will jump to “`Plt la:`” since the content of `$gp+24` point to “`Plt la:`”. The “`Plt la:`” code will call dynamic linker to load `la()` function, run `la()` and back to `bar()` as Figure 2.14.

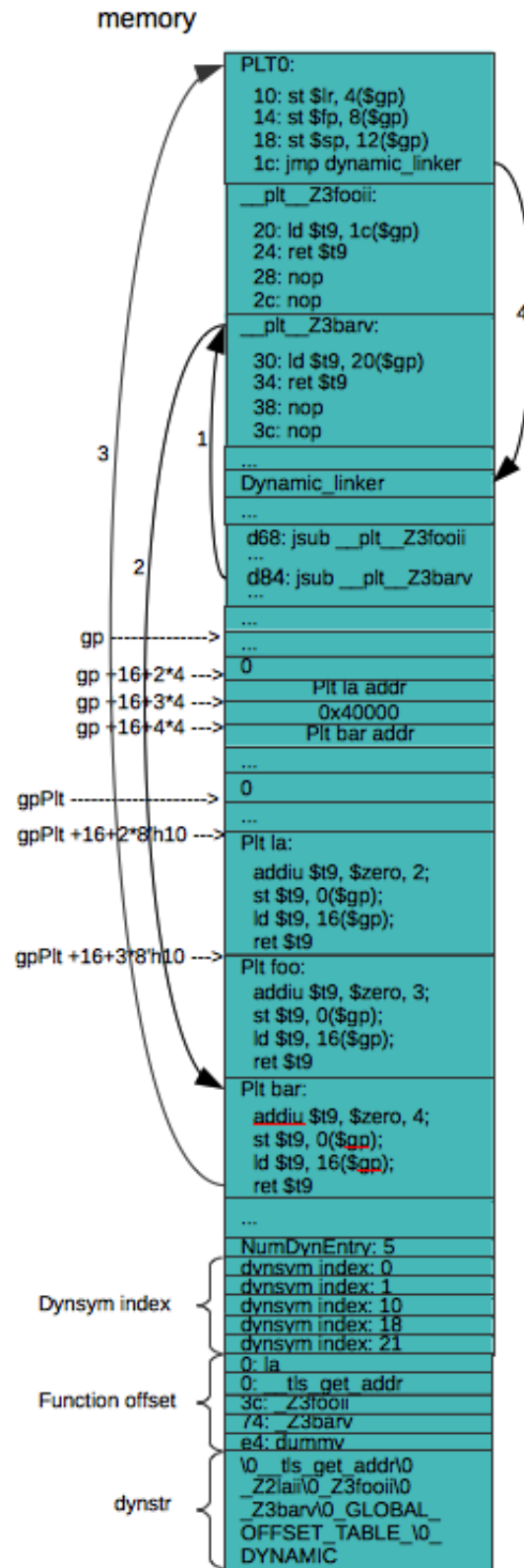


Figure 2.12: Control flow transfer from calling bar() instruction of main() to dynamic linker

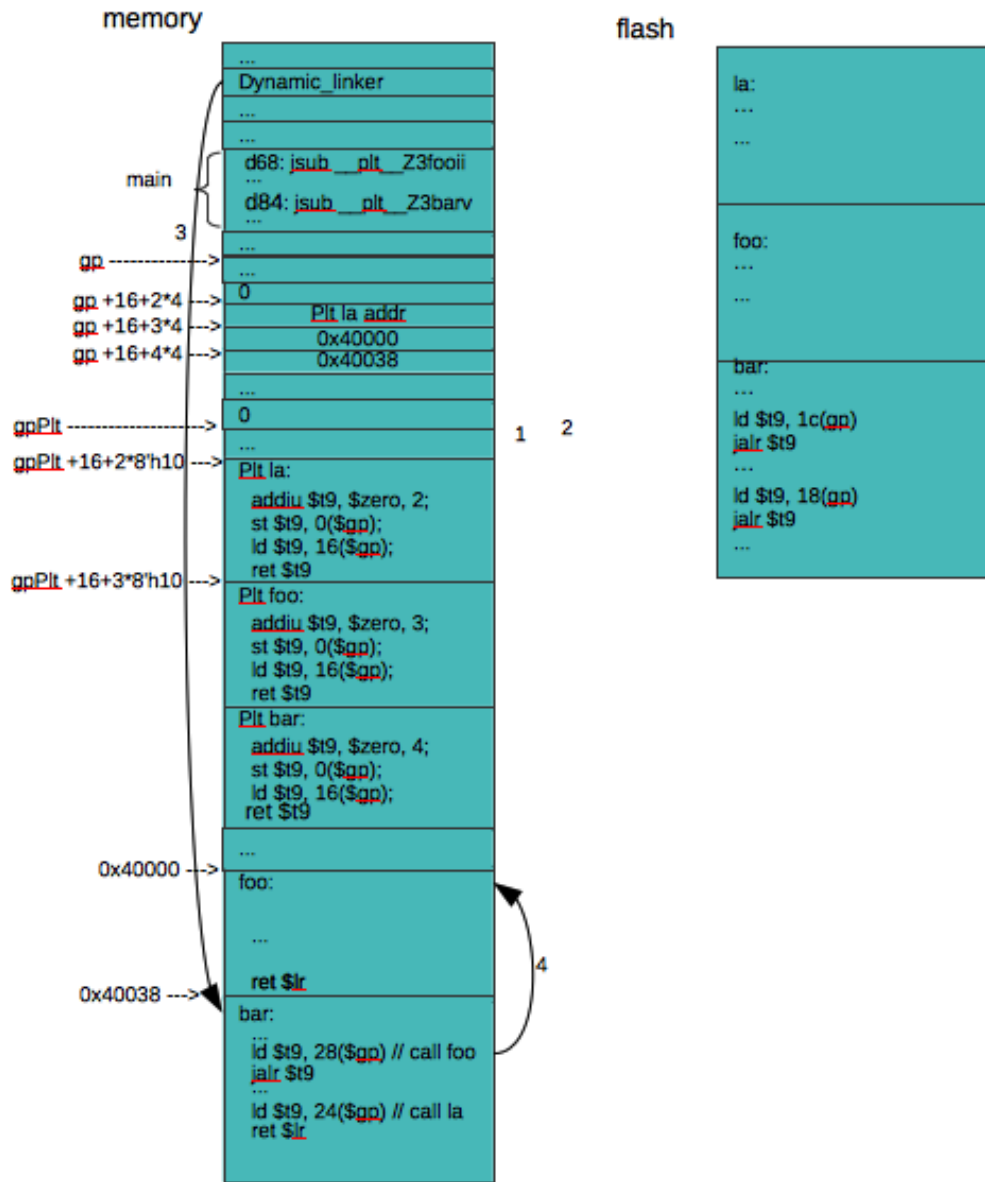


Figure 2.13: Dynamic linker load bar() from flash to memory

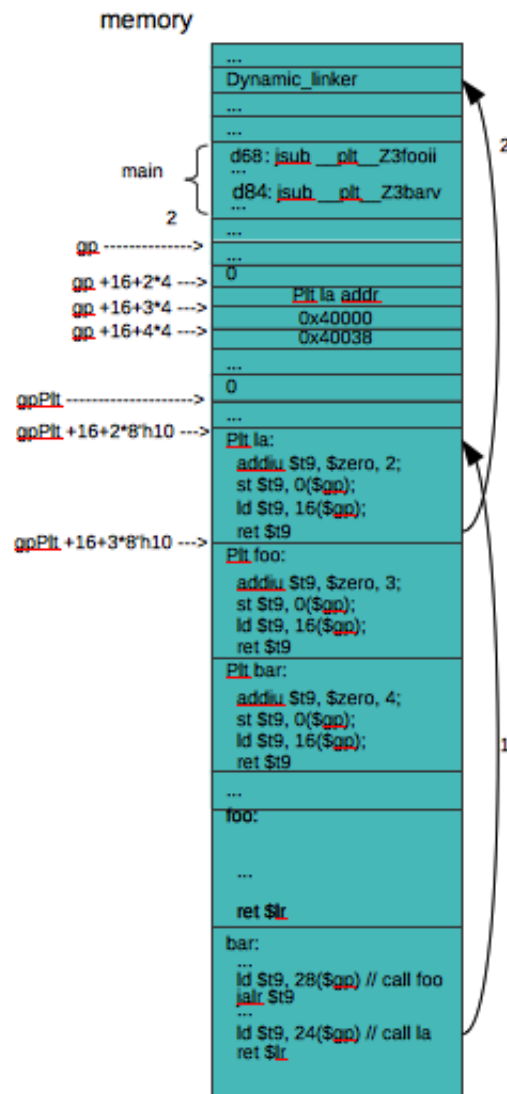


Figure 2.14: Call la through “Plt la:” in bar()

The dynamic linker implementation usually is not specified in ABI. It needs the co-work between linker and dynamic linker/loader. It uses the pointers (the area from  $gp+16+1*4$  to  $gp+16+(numDynEntry-1)*4$ ). When the code is loaded, this corresponding pointer in this area points to the loaded memory. Otherwise, it points to dynamic linker. The `Plt` or `__plt_Z3fooi`, `__pltZ3barv` are coding in our `cpu0PltAtomContent[]` of `Cpu0RelocationPass.cpp`. It is called linkage editor implementation.

## 2.3 Summary

### 2.3.1 Create a new backend base on LLVM

Thanks the llvm open source project. To write a linker and ELF to Hex tools for a new CPU architecture is easy and reliable. Combined with the llvm Cpu0 backend code and Verilog language code programmed in previous chapters, we design a software toolchain to compile C/C++ code, link and run it on Verilog Cpu0 simulator without any real hardware investment. If you buy the FPGA development hardware, we believe these code can run on FPGA CPU even though we didn't do it. Extend system program toolchain to support a new CPU instruction set can be finished just like we have shown you at this point. School knowledges of system program, compiler, linker, loader, computer architecture and CPU design has been translated into a real work and see how it is running. Now, these school books knowledge is not limited on paper. We design it, program it, and run it on real world.

The total code size of llvm Cpu0 backend compiler, Cpu0 lld linker, llvm-objdump with elf2hex Cpu0 support and Cpu0 Verilog Language is around 10 thousands lines of source code include comments. The total code size of clang, llvm and lld has 1000 thousands lines exclude the test and documents parts. It is only 1 % of the llvm size. More over, the llvm Cpu0 backend and lld Cpu0 backend are 70% of same with llvm Mips and lld X86\_64. Based on this truth, we believe llvm is a well defined structure in compiler architecture.

### 2.3.2 Contribute back to Open Source through working and learning

Finally, 10 thousands lines of source code in Cpu0 backend is very small in UI program. But it's quite complex in system program which based on llvm. We spent 600 pages of pdf to explain these code. Open source code give programmers best opportunity to understand the code and enhance/extend the code function. But it can be better, we believe the documentation is the next most important thing to improve the open source code development. The Open Source Organization recognized this point and set Open Source Document Project years ago<sup>7 8 9 10 11</sup>. Open Source grows up and becomes a giant software infrastructure with the forces of company<sup>12 13</sup>, school research team and countless talent engineers passion. It terminated the situation of everyone trying to re-invent wheels during 10 years ago. Extend your software from the re-usable source code is the right way. Of course you should consider an open source license if you are working with business. Actually anyone can contribute back to open source through the learning process. This book is written through the process of learning llvm backend and contribute back to llvm open source project. We think this book cannot exists in traditional paper book form since only few number of readers interested in study llvm backend even though there are many paper published books in concept of compiler. So, this book is published via electric media form and try to match the Open Document License Expection<sup>14</sup>. There are distance between the concept and the realistic program implemenation. Keep note through learning a large complicate software such as this llvm backend is not enough. We all learned the knowledge through books during school and after school. So, if you cannot find a good way to produce documents, you can consider to write documents like this book. This book document uses sphinx tool just like the llvm development team. Sphinx uses restructured text format

---

<sup>7</sup> [http://en.wikipedia.org/wiki/BSD\\_Documentation\\_License](http://en.wikipedia.org/wiki/BSD_Documentation_License)

<sup>8</sup> <http://www.freebsd.org/docproj/>

<sup>9</sup> <http://www.freebsd.org/copyright/freebsd-doc-license.html>

<sup>10</sup> [http://en.wikipedia.org/wiki/GNU\\_Free\\_Documentation\\_License](http://en.wikipedia.org/wiki/GNU_Free_Documentation_License)

<sup>11</sup> <http://www.gnu.org/copyleft/fdl.html>

<sup>12</sup> <http://www.apple.com/opensource/>

<sup>13</sup> <https://www.ibm.com/developerworks/opensource/>

<sup>14</sup> <http://www.gnu.org/philosophy/free-doc.en.html>

here <sup>15</sup> <sup>16</sup> <sup>17</sup>. Appendix A of lbd book tell you how to install sphinx tool. Documentation work will help yourself to re-examine your software and make your program better in structure, reliability and more important “Extend your code to somewhere you didn’t expect”.

---

<sup>15</sup> <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>

<sup>16</sup> <http://docutils.sourceforge.net/docs/ref/rst/directives.html>

<sup>17</sup> <http://docutils.sourceforge.net/rst.html>



## OPTIMIZATION

- LLVM IR optimization
- Project
  - LLVM-VPO

This chapter introduce llvm optimization.

### 3.1 LLVM IR optimization

The llvm-link provide optimizatoin in IR level which can apply in different programs developed by more than one language. Of course, it can apply in the same language which support seperate compile.

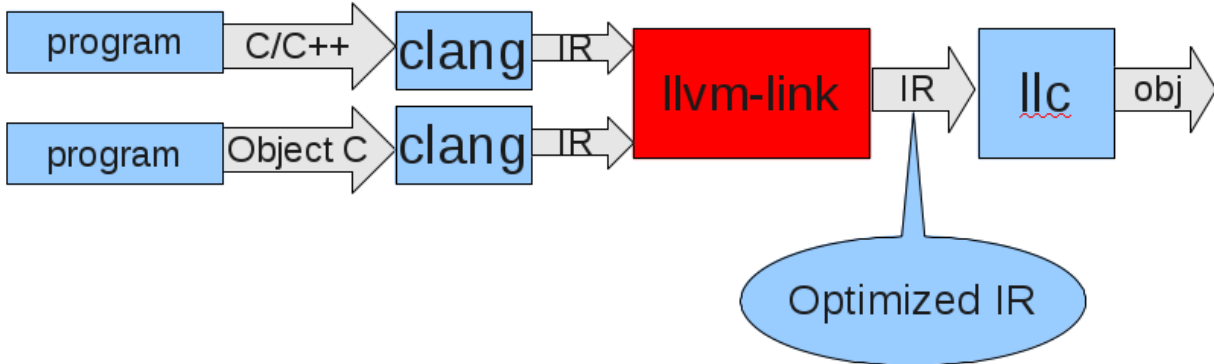


Figure 3.1: llvm-link flow

Clang provide optimization options to do optimatoin from high level language to IR. But since many languages like C/C++ support seperate compilation, it meaning there is no chance to do inter-procedure optimization if the functions come from different source files. To solve this problem, llvm provide **llvm-link** to link all \*.bc into a single IR file, and through **opt** to finish the inter-procedure optimization<sup>1</sup>. Beyond the DAG local optimization mentioned in Chapter 2, there are global optimization based on inter-procedure analysis<sup>2</sup>. The following steps and examples show this optimization solution in llvm.

---

<sup>1</sup> <http://www.cs.cmu.edu/afs/cs/academic/class/15745-s12/public/lectures/L3-LLVM-Part1.pdf>

<sup>2</sup> Refer chapter 9 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

**exlbt/input/optimizen/1.cpp**

```
int callee(const int *a) {
    return *a+1;
}
```

**exlbt/input/optimize/2.cpp**

```
extern int callee(const int *X);

int caller() {
    int T;

    T = 4;

    return callee(&T);
}
```

```
JonathantekiiMac:input Jonathan$ clang -O3 -target mips-unknown-linux-gnu
-c 1.cpp -emit-llvm -o 1.bc
JonathantekiiMac:input Jonathan$ clang -O3 -target mips-unknown-linux-gnu
-c 2.cpp -emit-llvm -o 2.bc
JonathantekiiMac:input Jonathan$ llvm-link -o=a.bc 1.bc 2.bc
JonathantekiiMac:input Jonathan$ opt -O3 -o=a1.bc a.bc
JonathantekiiMac:input Jonathan$ llvm-dis a.bc -o -
```

```
...
; Function Attrs: nounwind readonly
define i32 @_Z6calleePKi(i32* nocapture readonly %a) #0 {
    %1 = load i32* %a, align 4, !tbaa !1
    %2 = add nsw i32 %1, 1
    ret i32 %2
}
```

```
define i32 @_Z6callerv() #1 {
    %T = alloca i32, align 4
    store i32 4, i32* %T, align 4, !tbaa !1
    %1 = call i32 @_Z6calleePKi(i32* %T)
    ret i32 %1
}
...
```

```
JonathantekiiMac:input Jonathan$ llvm-dis a1.bc -o -
...
; Function Attrs: nounwind readonly
define i32 @_Z6calleePKi(i32* nocapture readonly %a) #0 {
    %1 = load i32* %a, align 4, !tbaa !1
    %2 = add nsw i32 %1, 1
    ret i32 %2
}

; Function Attrs: nounwind readnone
define i32 @_Z6callerv() #1 {
    ret i32 5
}
...
```

From the result as above, the **opt** output has lesser number of IR instructions. Of course, the backend code will be

more effective as follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm a.bc -o -
.section .mdebug.abi32
.previous
.file "a.bc"
.text
.globl      _Z6calleePKi
.align      2
.type       _Z6calleePKi,@function
.ent        _Z6calleePKi          # @_Z6calleePKi
_Z6calleePKi:
.frame      $sp,0,$lr
.mask       0x00000000,0
.set        noreorder
.set        nomacro
# BB#0:
ld          $2, 0($sp)
ld          $2, 0($2)
addiu       $2, $2, 1
ret         $lr
.set        macro
.set        reorder
.end        _Z6calleePKi
$tmp0:
.size       _Z6calleePKi, ($tmp0)-_Z6calleePKi

.globl      _Z6callerv
.align      2
.type       _Z6callerv,@function
.ent        _Z6callerv           # @_Z6callerv
_Z6callerv:
.cfi_startproc
.frame      $sp,32,$lr
.mask       0x00004000,-4
.set        noreorder
.cpload     $t9
.set        nomacro
# BB#0:
addiu       $sp, $sp, -32
$tmp3:
.cfi_def_cfa_offset 32
st          $lr, 28($sp)         # 4-byte Folded Spill
$tmp4:
.cfi_offset 14, -4
.cprestore  8
addiu       $2, $zero, 4
st          $2, 24($sp)
addiu       $2, $sp, 24
st          $2, 0($sp)
ld          $t9, %call16(_Z6calleePKi)($gp)
jalr        $t9
ld          $gp, 8($sp)
ld          $lr, 28($sp)         # 4-byte Folded Reload
addiu       $sp, $sp, 32
ret         $lr
.set        macro
.set        reorder
```

```
.end _Z6callerv
$tmp5:
.size _Z6callerv, ($tmp5)-_Z6callerv
.cfi_endproc

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm a1.bc -o -
.section .mdebug.abi32
.previous
.file "a1.bc"
.text
.globl _Z6calleePKi
.align 2
.type _Z6calleePKi,@function
.ent _Z6calleePKi # @_Z6calleePKi
_Z6calleePKi:
.frame $sp,0,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
ld $2, 0($sp)
ld $2, 0($2)
addiu $2, $2, 1
ret $lr
.set macro
.set reorder
.end _Z6calleePKi
$tmp0:
.size _Z6calleePKi, ($tmp0)-_Z6calleePKi

.globl _Z6callerv
.align 2
.type _Z6callerv,@function
.ent _Z6callerv # @_Z6callerv
_Z6callerv:
.frame $sp,0,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $2, $zero, 5
ret $lr
.set macro
.set reorder
.end _Z6callerv
$tmp1:
.size _Z6callerv, ($tmp1)-_Z6callerv
```

Though llvm-link provide optimization in IR level to support separete compile, it come with the cost in compile time. As you can imagine, any one statement change will change the output IR of llvm-link. And the obj binary code have to re-compile. Compare to the separete compile for each \*.c file, it only need to re-compile the corresponding \*.o file only.

## 3.2 Project

### 3.2.1 LLVM-VPO

Friend Gang-Ryung Uh replace LLC compiler by llvm on Very Portable Optimizer (VPO) compiler toolchain. VPO performs optimizations on a single intermediate representation called Register Transfer Lists (RTLs). In other word, the system generate RTLs from llvm IR and it do further optimization on RTLs.

The LLVM-VPO is illustrated at his home page. Click **“6. LLVM-VPO Compiler Development - 2012 Google Faculty Research Award”** at this home page<sup>3</sup> will get the information.

---

<sup>3</sup> <http://cs.boisestate.edu/~uh/>



## LIBRARY

- Compiler-rt
- Avr libc
- Software Float Point Support

Since Cpu0 has not hardware float point instructions, it needs soft float point library to finish the floating point operation. LLVM compiler-rt project include the software floating point library implementation, so we choose it as the implementation.

Since compiler-rt uses unix/linux rootfs structure, we fill the gap by porting avr libc.

Both the compiler-rt and avr libc porting is under going, it's not finished. The flow as follows,

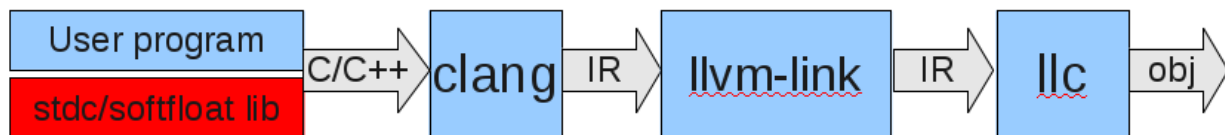


Figure 4.1: libc/softfloat library flow

The llvm-link which introduced at last chapter can be hired for optimization.

### 4.1 Compiler-rt

Directory libex/libsoftfloat/compiler-rt include the floating point library support for Cpu0 backend. The compiler-rt<sup>1</sup> version we use is llvm 3.5 release.

### 4.2 Avr libc

Directory libex/libc/avr-libc-1.8.1 include the libc porting.

AVR Libc is a Free Software project whose goal is to provide a high quality C library for use with GCC on Atmel AVR microcontrollers. AVR Libc is licensed under a single unified license. This so-called modified Berkeley license is intended to be compatible with most Free Software licenses like the GPL, yet impose as little restrictions for the use of the library in closed-source commercial applications as possible<sup>2</sup>.

---

<sup>1</sup> <http://compiler-rt.llvm.org/>

<sup>2</sup> <http://www.nongnu.org/avr-libc/>

The source code can be download from here <sup>3</sup>. Document are here <sup>4 5</sup>.

## 4.3 Software Float Point Support

exlbt/input/ch\_float.cpp

```
#include "debug.h"

#define SINGLE_PRECISION
#include "fp_lib.h"

extern "C" int __fixsfsi(fp_t a);
extern "C" int __fixdfsi(double a);

extern long long test_longlong_shift1();
extern long long test_longlong_shift2();

#include "print.cpp"

int test_float_to_int()
{
    float a = -2.2;
    si_int c = __fixsfsi(a);

    return c;
}

// work
int test_double_to_int()
{
    double a = -4.2;
    si_int c = __fixdfsi(a);

    return c;
}

#include "int_lib.h"

extern "C" int printf(const char *format, ...);
extern "C" int sprintf(char *out, const char *format, ...);

int test_float_add()
{
    float a = -2.2;
    float b = 3.3;

    int c = (int) (a + b); // 1

    return c;
}

int test_float_mul()
```

---

<sup>3</sup> <http://download.savannah.gnu.org/releases/avr-libc/>

<sup>4</sup> <http://www.atmel.com/webdoc/AVRLibcReferenceManual/index.html>

<sup>5</sup> <http://courses.cs.washington.edu/courses/csep567/04sp/pdfs/avr-libc-user-manual.pdf>



```
{
    float a = -2.2;
    float b = 3.3;
    float c = -4.3;

    int d = (int)(a * b * c); // 31.218=31

    return d;
}

int test_float_div()
{
    float a = -2.2;
    float b = 3.3;
    float c = 1.2;

    int d = (int)((a * b) / c); // -6

    return d;
}

int test_double_to_int_2()
{
    double b = 3.3;

    int c = (int)(b);

    return c;
}

int test_double_add()
{
    double a = 2.2;
    double b = 3.3;

    int c = (int)(a + b);

    return c;
}

int test_double_mul()
{
    double a = -2.2;
    double b = 3.3;
    double c = 4.3;

    int d = (int)(a * b * c); // -31.218=-31

    return d;
}

int test_double_div()
{
    double a = -2.2;
    double b = 3.3;
    double c = 1.2;

    int d = (int)((a * b) / c); // -6
```

```
    return d;
}

#if 0
//extern "C" di_int __addvdi3(di_int a, di_int b);
extern "C" di_int __addvdi3(long long a, long long b);

//int test__addvdi3(di_int a, di_int b)
int test__addvdi3(long long a, long long b)
{
    di_int x = __addvdi3(a, b);
    di_int expected = a + b;
    if (x != expected)
        printf("error in test__addvdi3(0x%llx, 0x%llx) = %lld, expected %lld\n",
               a, b, x, expected);
    return x != expected;
}

int test_addvdi3()
{
    //    test__addvdi3(0x8000000000000000LL, -1); // should abort
    //    test__addvdi3(-1, 0x8000000000000000LL); // should abort
    //    test__addvdi3(1, 0x7FFFFFFFFFFFFFFFLL); // should abort
    //    test__addvdi3(0x7FFFFFFFFFFFFFFFLL, 1); // should abort

    if (test__addvdi3(0x8000000000000000LL, 1))
        return 1;
    if (test__addvdi3(1, 0x8000000000000000LL))
        return 1;
    if (test__addvdi3(0x8000000000000000LL, 0))
        return 1;
    if (test__addvdi3(0, 0x8000000000000000LL))
        return 1;
    if (test__addvdi3(0x7FFFFFFFFFFFFFFFLL, -1))
        return 1;
    if (test__addvdi3(-1, 0x7FFFFFFFFFFFFFFFLL))
        return 1;
    if (test__addvdi3(0x7FFFFFFFFFFFFFFFLL, 0))
        return 1;
    if (test__addvdi3(0, 0x7FFFFFFFFFFFFFFFLL))
        return 1;

    return 0;
}
#endif

extern "C" di_int __absvdi2(di_int a);

int test__absvdi2(di_int a)
{
    di_int x = __absvdi2(a);
    di_int expected = a;
    if (expected < 0)
        expected = -expected;
    if (x != expected || expected < 0)
        printf("error in __absvdi2(0x%08X%08X) = %08d%08d, expected positive %08d%08d\n",
               (int)(a>>32), (int)a, (int)(x>>32), (int)x, int(expected>>32), (int)expected);
}
```

```

    return x != expected;
}

int test_absvdi2()
{
    //    if (test__absvdi2(0x8000000000000000LL))    // should abort
    //        return 1;
    test__absvdi2(0x0000000000000000LL);
    test__absvdi2(0x0000000000000001LL);
    test__absvdi2(0x0000000000000002LL);
    test__absvdi2(0x7FFFFFFFFFFFFFFELL);
    test__absvdi2(0x7FFFFFFFFFFFFFFELL);
    test__absvdi2(0x8000000000000001LL);
    test__absvdi2(0x8000000000000002LL);
    test__absvdi2(0xFFFFFFFFFFFFFFELL);
    test__absvdi2(0xFFFFFFFFFFFFFFELL);

    int i;
    for (i = 0; i < 100; ++i)
        if (test__absvdi2(((di_int)i << 32) | 1))
            return 1;

    return 0;
}

extern "C" si_int __absvsi2(si_int a);

int test__absvsi2(si_int a)
{
    si_int x = __absvsi2(a);
    si_int expected = a;
    if (expected < 0)
        expected = -expected;
    if (x != expected || expected < 0)
        printf("error in __absvsi2(0x%X) = %d, expected positive %d\n",
              a, x, expected);
    return x != expected;
}

int test_absvsi2()
{
    //    if (test__absvsi2(0x80000000))    // should abort
    //        return 1;
    test__absvsi2(0x00000000);
    test__absvsi2(0x00000001);
    test__absvsi2(0x00000002);
    test__absvsi2(0x7FFFFFFE);
    test__absvsi2(0x7FFFFFFF);
    test__absvsi2(0x80000001);
    test__absvsi2(0x80000002);
    test__absvsi2(0xFFFFFFFF);
    test__absvsi2(0xFFFFFFFF);

    int i;
    for (i = 0; i < 100; ++i)
        if (test__absvsi2(i))
            return 1;
}

```

```
    return 0;
}

#if 0
#define CRT_HAS_128BIT

#ifndef CRT_HAS_128BIT

// Returns: absolute value

// Effects: aborts if abs(x) < 0

extern "C" ti_int __absvti2(ti_int a);

int test__absvti2(ti_int a)
{
    ti_int x = __absvti2(a);
    ti_int expected = a;
    if (expected < 0)
        expected = -expected;
    if (x != expected || expected < 0)
    {
        twords at;
        at.all = a;
        twords xt;
        xt.all = x;
        twords expectedt;
        expectedt.all = expected;
        printf("error in __absvti2(0x%8X%8X.%8X%8X) = "
               "0x%8X%8X.%8X%8X, expected positive 0x%8X%8X.%8X%8X\n",
               (int)(at.s.high>>32), (int)(at.s.high), (int)(at.s.low>>32),
               (int)(at.s.low), (int)(xt.s.high>>32), (int)(xt.s.high),
               (int)(xt.s.low>>32), (int)(xt.s.low),
               (int)(expectedt.s.high>>32), (int)(expectedt.s.high),
               (int)(expectedt.s.low>>32), (int)(expectedt.s.low));
    }
    return x != expected;
}

#endif

int test_absvti2()
{
#ifndef CRT_HAS_128BIT

//      if (test__absvti2(make_ti(0x8000000000000000LL, 0))) // should abort
//          return 1;
    if (test__absvti2(0x0000000000000000LL))
        return 1;
    if (test__absvti2(0x0000000000000001LL))
        return 1;
    if (test__absvti2(0x0000000000000002LL))
        return 1;
    if (test__absvti2(make_ti(0x7FFFFFFFFFFFFFFFFFLL, 0xFFFFFFFFFFFFFFFFFELL)))
        return 1;
    if (test__absvti2(make_ti(0x7FFFFFFFFFFFFFFFFFLL, 0xFFFFFFFFFFFFFFFFFELL)))
        return 1;
    if (test__absvti2(make_ti(0x8000000000000000LL, 0x0000000000000001LL)))
```

```

        return 1;
    if (test__absvti2(make_ti(0x8000000000000000LL, 0x0000000000000002LL))
        return 1;
    if (test__absvti2(make_ti(0xFFFFFFFFFFFFFFFFLL, 0xFFFFFFFFFFFFFFFFLL))
        return 1;
    if (test__absvti2(make_ti(0xFFFFFFFFFFFFFFFFLL, 0xFFFFFFFFFFFFFFFFLL))
        return 1;

    int i;
    for (i = 0; i < 10000; ++i)
        if (test__absvti2(make_ti(((ti_int)i << 32) | i,
                                ((ti_int)i << 32) | i)))
            return 1;
#else
    printf("skipped\n");
#endif
    return 0;
}
#endif

int main() {
    int a;

    a = test_longlong_shift1(); // 0x121 = 289
    printf("test_longlong_shift1() = %d\n", a);
    a = test_longlong_shift2(); // 0x16 = 22
    printf("test_longlong_shift2() = %d\n", a);
    a = test_float_to_int(); // -2
    printf("test_float_to_int() = %d\n", a);
    a = test_double_to_int(); // -4
    printf("test_double_to_int() = %d\n", a);
    a = test_double_to_int_2(); // 3
    printf("test_double_to_int_2() = %d\n", a);
    a = test_float_add(); // 1
    printf("test_float_add() = %d\n", a);
    a = test_float_mul(); // 31
    printf("test_float_mul() = %d\n", a);
    a = test_float_div(); // -6
    printf("test_float_div() = %d\n", a);
    a = test_double_add(); // 5
    printf("test_double_add() = %d\n", a);
    a = test_double_mul(); // -31
    printf("test_double_mul() = %d\n", a);
    a = test_double_div(); // -6
    printf("test_double_div() = %d\n", a);

    #if 0
        test_addvdi3();
    #endif
    test_absvdi2();
    test_absvsi2();

    return 0;
}

```

**exlbt/input/build-float.sh**

```
#!/usr/bin/env bash

INCFLAG="-I../libsoftfloat/compiler-rt/builtins"

source functions.sh

sh_name=build-float.sh
argNum=$#
arg1=$1
arg2=$2

prologue;

libsf=../libsoftfloat/compiler-rt
pushd ${libsf}
bash build.sh
popd
olibs=${libsf}/obj

clang -target mips-unknown-linux-gnu -c start.cpp -emit-llvm -o start.bc
clang -target mips-unknown-linux-gnu -c debug.cpp -emit-llvm -o debug.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg-def.c -emit-llvm \
-o printf-stdarg-def.bc
clang -target mips-unknown-linux-gnu -c printf-stdarg.c -emit-llvm \
-o printf-stdarg.bc
clang $INCFLAG -c ch_float.cpp -emit-llvm -o ch_float.bc
clang $INCFLAG -c ${LBDEXDIR}/input/ch9_7.cpp -emit-llvm -o ch9_7.bc
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj start.bc -o start.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj debug.bc -o debug.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg-def.bc -o printf-stdarg-def.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj printf-stdarg.bc -o printf-stdarg.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch9_7.bc -o ch9_7.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch_float.bc -o ch_float.cpu0.o
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
-filetype=obj lib_cpu0.ll -o lib_cpu0.o
${TOOLDIR}/lld -flavor gnu -target cpu0${endian}-unknown-linux-gnu \
-o a.out start.cpu0.o debug.cpu0.o printf-stdarg-def.cpu0.o printf-stdarg.cpu0.o \
ch_float.cpu0.o ch9_7.cpu0.o lib_cpu0.o \
${olibs}/libFloat.o
# ${olibs}/fixsfsi.o ${olibs}/fixsfdi.o ${olibs}/fixdfsi.o \
# ${olibs}/addsf3.o ${olibs}/mulsf3.o ${olibs}/divsf3.o \
# ${olibs}/adddf3.o ${olibs}/muldf3.o ${olibs}/divdf3.o \
# ${olibs}/ashrdi3.o ${olibs}/ashldi3.o ${olibs}/lshrdi3.o \
# ${olibs}/extendsfdf2.o ${olibs}/truncdfsf2.o

epilogue;
```

Run as follows,

```
JonathantekiiMac:input Jonathan$ bash build-float.sh cpu032II be
...
endian = BigEndian
0 /* 0: big endian, 1: little endian */

JonathantekiiMac:input Jonathan$ iverilog -o cpu0IIs cpu0IIs.v
JonathantekiiMac:input Jonathan$ ./cpu0IIs
114-43-184-210:verilog Jonathan$ ./cpu0IIs
WARNING: ./cpu0.v:458: $readmemh(cpu0.hex): Not enough words in the file for
the requested range [0:524287].
taskInterrupt(001)
test_longlong_shift1() = 289
test_longlong_shift2() = 22
test_float_to_int() = -2
test_double_to_int() = -4
test_double_to_int_2() = 3
test_float_add() = 1
test_float_mul() = 31
test_float_div() = -6
test_double_add() = 5
test_double_mul() = -31
test_double_div() = -6
total cpu cycles = 104105
RET to PC < 0, finished!
```





## BOOK EXAMPLE CODE

The example code `exlbt.tar.gz` is available in:

<http://jonathan2251.github.io/lbt/exlbt.tar.gz>



## **ALTERNATE FORMATS**

The book is also available in the following formats:



**PRESENTATION FILES**