GDC:

The GNU D Compiler

Iain Bucław

@ibuclaw

DConf 2013

Outline

Outline

Outline

1. History of Porting D Front End (DFE)
2. GDC Current Status
3. The Anatomy of a GCC Front End
4. GDC Extensions
5. Future Plans

Outline

1. History of Porting D Front End (DFE)
2. GDC Current Status
3. The Anatomy of a GCC Front End
4. GDC Extensions
5. Future Plans

Outline

What is GCC?

- Developed to be 100% free software.
- Solid support for multiple language.
- Ported to almost all architectures.

What is GCC?

- Developed to be 100% free software.
- Solid support for multiple language.
- Ported to almost all architectures.

What is GCC?

- Developed to be 100% free software.
- Solid support for multiple language.
- Ported to almost all architectures.

That and because the Clang/LLVM compiler was not to appear for another 5 years...

A Short History of Porting the D Front End.

History

January/2002:
Early discussions of wanting to port D to Linux began.

April/2002:
Walter Bright releases D Front End sources.

May/2002:
Birth of D.gnu Mailing List and BrightD Compiler Project.

June/2002:
OpenD Compiler Project announced.

History

January/2002:
: Early discussions of wanting to port D to Linux began.

April/2002:
: Walter Bright releases D Front End sources.

May/2002:
: Birth of D.gnu Mailing List and BrightD Compiler Project.

June/2002:
: OpenD Compiler Project announced.

History

January/2002:
: Early discussions of wanting to port D to Linux began.

April/2002:
: Walter Bright releases D Front End sources.

May/2002:
: Birth of D.gnu Mailing List and BrightD Compiler Project.

June/2002:
: OpenD Compiler Project announced.

History

January/2002:
: Early discussions of wanting to port D to Linux began.

April/2002:
: Walter Bright releases D Front End sources.

May/2002:
: Birth of D.gnu Mailing List and BrightD Compiler Project.

June/2002:
: OpenD Compiler Project announced.

History

August/2002:
: D Linux (DLI) released.

May/2003:
: Walter Ports DMD to Linux.

February/2004:
: GDMD Compiler Released.

March/2004:
: DGCC Compiler Released.

History

August/2002:
: D Linux (DLI) released.

May/2003:
: Walter Ports DMD to Linux.

February/2004:
: GDMD Compiler Released.

March/2004:
: DGCC Compiler Released.

History

August/2002:

D Linux (DLI) released.

May/2003:

Walter Ports DMD to Linux.

February/2004:

GDMD Compiler Released.

March/2004:

DGCC Compiler Released.

History

August/2002:
: D Linux (DLI) released.

May/2003:
: Walter Ports DMD to Linux.

February/2004:
: GDMD Compiler Released.

March/2004:
: DGCC Compiler Released.

History

September/2007:

New Development of an LLVM D Compiler.

June/2008:

DGCC Development Abandoned.

September/2009:

GDC Revival Project Kicks Off.

December/2009:

Enter Your Humble Speaker.

History

September/2007:

New Development of an LLVM D Compiler.

June/2008:

DGCC Development Abandoned.

September/2009:

GDC Revival Project Kicks Off.

December/2009:

Enter Your Humble Speaker.

History

September/2007:

New Development of an LLVM D Compiler.

June/2008:

DGCC Development Abandoned.

September/2009:

GDC Revival Project Kicks Off.

December/2009:

Enter Your Humble Speaker.

History

September/2007:
: New Development of an LLVM D Compiler.

June/2008:
: DGCC Development Abandoned.

September/2009:
: GDC Revival Project Kicks Off.

December/2009:
: Enter Your Humble Speaker.

End of Life for D1?

November/2011:
: LDC D2 Compiler becomes default version that is built.

December/2011:
: D1 discontinued support starting from 2013.

January/2012:
: GDC drops D1 Front End from development.

End of Life for D1?

November/2011:
: LDC D2 Compiler becomes default version that is built.

December/2011:
: D1 discontinued support starting from 2013.

January/2012:
: GDC drops D1 Front End from development.

End of Life for D1?

November/2011:
: LDC D2 Compiler becomes default version that is built.

December/2011:
: D1 discontinued support starting from 2013.

January/2012:
: GDC drops D1 Front End from development.

Current State of D2 Compiler

- Three main compilers based off the D2 Front End.
- Platform support for Linux, FreeBSD, OSX, Solaris, and Windows.
- Target support for ARM, PowerPC, x86, x86_64.
- D Runtime gaining support for more targets.
- Phobos becoming platform agnostic.

Current State of D2 Compiler

- Three main compilers based off the D2 Front End.
- Platform support for Linux, FreeBSD, OSX, Solaris, and Windows.
- Target support for ARM, PowerPC, x86, x86_64.
- D Runtime gaining support for more targets.
- Phobos becoming platform agnostic.

Current State of D2 Compiler

- Three main compilers based off the D2 Front End.
- Platform support for Linux, FreeBSD, OSX, Solaris, and Windows.
- Target support for ARM, PowerPC, x86, x86_64.
- D Runtime gaining support for more targets.
- Phobos becoming platform agnostic.

Current State of D2 Compiler

- Three main compilers based off the D2 Front End.
- Platform support for Linux, FreeBSD, OSX, Solaris, and Windows.
- Target support for ARM, PowerPC, x86, x86_64.
- D Runtime gaining support for more targets.
- Phobos becoming platform agnostic.

Current State of D2 Compiler

- Three main compilers based off the D2 Front End.
- Platform support for Linux, FreeBSD, OSX, Solaris, and Windows.
- Target support for ARM, PowerPC, x86, x86_64.
- D Runtime gaining support for more targets.
- Phobos becoming platform agnostic.

Current GDC Support Status.

GDC: Language Support

- D Front End 2.062.
- Passes 95% on D2 Testsuite.
- Work being done on passing D Runtime/Phobos Unittests.

GDC: Target Support

- **x86/x86_64:** Solid support.
- **ARM:** Partial support.
- **MIPS:** Partial support.
- **Others:** Untested / No runtime support.

GDC: Platform Support

- **GNU/Linux:** Main support platform.
- **FreeBSD/OpenBSD:** Support should be there.
- **OSX:** Lacks TLS Support.
- **Windows/MinGW:** Alpha quality release available.

GDC: To Hell With DMD Compatibility.

- GDC follows the D calling convention as per the spec.
  - Except for Win32, which defines the D calling convention.
  - Uses thiscall convention for methods.
- No D Inline Assembly implemented.
- No naked function support.
- Type va_list matches C ABI.

GDC: To Hell With DMD Compatibility.

- GDC follows the D calling convention as per the spec.
  - Except for Win32, which defines the D calling convention.
  - Uses thiscall convention for methods.
- No D Inline Assembly implemented.
- No naked function support.
- Type va_list matches C ABI.

GDC: To Hell With DMD Compatibility.

- GDC follows the D calling convention as per the spec.
  - Except for Win32, which defines the D calling convention.
  - Uses thiscall convention for methods.

- No D Inline Assembly implemented.

- No naked function support.

- Type va_list matches C ABI.

GDC: To Hell With DMD Compatibility.

- GDC follows the D calling convention as per the spec.
  - Except for Win32, which defines the D calling convention.
  - Uses thiscall convention for methods.

- No D Inline Assembly implemented.

- No naked function support.

- Type va_list matches C ABI.

GDC: To Hell With DMD Compatibility.

- GDC follows the D calling convention as per the spec.
  - Except for Win32, which defines the D calling convention.
  - Uses thiscall convention for methods.
- No D Inline Assembly implemented.
- No naked function support.
- Type va_list matches C ABI.

GDC: To Hell With DMD Compatibility.

- GDC follows the D calling convention as per the spec.
  - Except for Win32, which defines the D calling convention.
  - Uses thiscall convention for methods.
- No D Inline Assembly implemented.
- No naked function support.
- Type va_list matches C ABI.

GDC: To Hell With DMD Compatibility.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.

- **gcov** and **gprof** replace -cov and -profile.

- **gdmd** script maintained separately.

- No support for D DWARF extensions.

GDC: To Hell With DMD Compatibility.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.
- **gcov** and **gprof** replace -cov and -profile.
- **gdmd** script maintained separately.
- No support for D DWARF extensions.

GDC: To Hell With DMD Compatibility.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.

- **gcov** and **gprof** replace -cov and -profile.

- **gdmd** script maintained separately.

- No support for D DWARF extensions.

GDC: To Hell With DMD Compatibility.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.

- **gcov** and **gprof** replace -cov and -profile.

- **gdmd** script maintained separately.

- No support for D DWARF extensions.

GDC: To Hell With DMD Compatibility.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.
- **gcov** and **gprof** replace -cov and -profile.
- **gdmd** script maintained separately.
- No support for D DWARF extensions.

GDC: To Hell With DMD Compatibility.

- No __simd support.
  - Allow __vector sizes of 8, 16 or 32 bytes.
  - No current restrictions on what targets can use __vector.
- **gcov** and **gprof** replace -cov and -profile.
- **gdmd** script maintained separately.
- No support for D DWARF extensions.

The Anatomy of a GCC Front End.

Why GCC?

- The entry barrier to GCC development has gotten considerably lower during the last few years.

- With work on documentation and separation of internal modules, writing your own front end for GCC has become accessible to a wider community of developers.

Introduction to GCC

- Able to translate from a variety of source languages to assembly.

- Encapsulated into one command.

- Front end is made up of two main components.

Introduction to GCC

- Able to translate from a variety of source languages to assembly.

- Encapsulated into one command.

- Front end is made up of two main components.

Introduction to GCC

- Able to translate from a variety of source languages to assembly.
- Encapsulated into one command.
- Front end is made up of two main components.

Compilation Driver

- User interfacing application.
- Knows about all supported languages.
- Able to determine source language.
- Passes output between compiler and assembler.

Compilation Driver

- User interfacing application.
- Knows about all supported languages.
- Able to determine source language.
- Passes output between compiler and assembler.

Compilation Driver

- User interfacing application.
- Knows about all supported languages.
- Able to determine source language.
- Passes output between compiler and assembler.

Compilation Driver

- User interfacing application.
- Knows about all supported languages.
- Able to determine source language.
- Passes output between compiler and assembler.

Compiler Proper

- One compiler proper for each language.

- Composed from three components.

Compiler Proper

- One compiler proper for each language.
- Composed from three components.

Front End, Middle End and Back End

- The Front End contains all the language processing logic.
- The Middle End is the platform independent part of the compiler.
- The Back End is then the platform dependent part.

Front End, Middle End and Back End

- The Front End contains all the language processing logic.
- The Middle End is the platform independent part of the compiler.
- The Back End is then the platform dependent part.

Front End, Middle End and Back End

- The Front End contains all the language processing logic.
- The Middle End is the platform independent part of the compiler.
- The Back End is then the platform dependent part.

GENERIC

- GENERIC is a tree language.
- Mechanism to define own node types.
- Supports everything there is to represent in a typical C function.
- During the course of compilation, it is lowered into an intermediate code called GIMPLE.

GENERIC

- GENERIC is a tree language.
- Mechanism to define own node types.
- Supports everything there is to represent in a typical C function.
- During the course of compilation, it is lowered into an intermediate code called GIMPLE.

GENERIC

- GENERIC is a tree language.
- Mechanism to define own node types.
- Supports everything there is to represent in a typical C function.
- During the course of compilation, it is lowered into an intermediate code called GIMPLE.

GENERIC

- GENERIC is a tree language.
- Mechanism to define own node types.
- Supports everything there is to represent in a typical C function.
- During the course of compilation, it is lowered into an intermediate code called GIMPLE.

GIMPLE

- GIMPLE is a subset of GENERIC.
- Breaks down all expressions, using temporaries to store intermediate results.
- Further transforms all blocks into gotos and labels.
- Lowered down to RTL, or Register Transfer Language.

GIMPLE

- GIMPLE is a subset of GENERIC.
- Breaks down all expressions, using temporaries to store intermediate results.
- Further transforms all blocks into gotos and labels.
- Lowered down to RTL, or Register Transfer Language.

GIMPLE

- GIMPLE is a subset of GENERIC.
- Breaks down all expressions, using temporaries to store intermediate results.
- Further transforms all blocks into gotos and labels.
- Lowered down to RTL, or Register Transfer Language.

GIMPLE

- GIMPLE is a subset of GENERIC.
- Breaks down all expressions, using temporaries to store intermediate results.
- Further transforms all blocks into gotos and labels.
- Lowered down to RTL, or Register Transfer Language.

Interfacing with D Front-End

- GDC initialises the D Front-End, sets up all global parameters.
- D Front-End parses and runs semantic on the code.
- GDC generates GENERIC to be sent to backend.
- GCC backend compiles down to RTL.

Interfacing with D Front-End

- GDC initialises the D Front-End, sets up all global parameters.
- D Front-End parses and runs semantic on the code.
- GDC generates GENERIC to be sent to backend.
- GCC backend compiles down to RTL.

Interfacing with D Front-End

- GDC initialises the D Front-End, sets up all global parameters.
- D Front-End parses and runs semantic on the code.
- GDC generates GENERIC to be sent to backend.
- GCC backend compiles down to RTL.

Interfacing with D Front-End

- GDC initialises the D Front-End, sets up all global parameters.
- D Front-End parses and runs semantic on the code.
- GDC generates GENERIC to be sent to backend.
- GCC backend compiles down to RTL.

A Simple D Program

```
module demo;

int add(int a, int b)
{
    return a + b;
}
```

Code Generated in GENERIC

```
demo.add (int a, int b)
{
    return <retval> = a + b;
}
```

```
demo.add (int a, int b)
bind_expr (
    return_expr (
        init_expr (<retval>, plus_expr (a, b))
    )
)
```

Representation after Gimplification

```
demo.add (int a, int b)
{
    int vartmp0;
    vartmp0 = a + b;
    return vartmp0;
}
```

```
demo.add (int a, int b)
gimple_bind (
    int vartmp0;
    gimple_assign (plus_expr, vartmp0, a, b)
    gimple_return (vartmp0)
)
```

A More Interesting D Program

```
module demo;

long fib (uint m)
{
    return (m < 2) ? m : fib (m - 1) + fib (m - 2);
}
```

Code Generated in GENERIC

```
demo.fib(uint m)
{
    return <retval> = m <= 1 ? (long) m : demo.fib (m - 1) + demo.fib (m - 2);
}
```

```
demo.fib(uint m)
bind_expr (
    return_expr (
        init_expr (<retval>,
            cond_expr (le_expr, m, 1,
                nop_expr (m),
                plus_expr (call_expr (demo.fib, minus_expr (m, 1)),
                           call_expr (demo.fib, minus_expr (m, 2)))
            )
        )
    )
)
```

Representation after Gimplification

```
demo.fib (uint m)
{
    long vartmp0;
    long iftmp0;
    uint vartmp1;
    long vartmp2;
    uint vartmp3;
    long vartmp4;
    if (m <= 1) goto L1; else goto L2;
    L1:
    iftmp0 = (long) m;
    goto L3;
    L2:
    vartmp1 = m + 4294967295;
    vartmp2 = demo.fib (vartmp1);
    vartmp3 = m + 4294967294;
    vartmp4 = demo.fib (vartmp3);
    iftmp0 = vartmp2 + vartmp4;
    L3:
    vartmp0 = iftmp0;
    return vartmp0;
}
```

Notation Representation

```
demo.fib (uint m)
gimple_bind (
    long vartmp0;
    uint vartmp1;
    long vartmp2;
    uint vartmp3;
    long vartmp4;
    long iftmp0;
    gimple_cond (le_expr, m, 1, (L1), (L2))
    gimple_label (L1)
    gimple_assign (nop_expr, iftmp0, m)
    gimple_goto (L3)
    gimple_label (L2)
    gimple_assign (plus_expr, vartmp1, m, 4294967295)
    gimple_call (demo.fib, vartmp2, vartmp1)
    gimple_assign (plus_expr, vartmp3, m, 4294967294)
    gimple_call (demo.fib, vartmp4, vartmp3)
    gimple_assign (plus_expr, iftmp0, vartmp2, vartmp4)
    gimple_label (L3)
    gimple_assign (var_decl, vartmp0, iftmp0)
    gimple_return (vartmp0)
)
```

GDC Extensions

Custom Static Chains

- Generated for all nested functions
- Generated for toplevel functions with nested references.

```
int delegate() foo()
{
    int x = 7;

    int bar()
    {
        int baz()
        {
            return x + 3;
        }
        return baz();
    }
    return &bar;
}
```

Generated GENERIC Code

```
closure.foo.bar.baz (void *this)
{
    return <retval> = ((CLOSURE.closure.foo *) this)->x + 3;
}

closure.foo.bar (void *this)
{
    return <retval> = closure.foo.bar.baz ((CLOSURE.closure.foo *) this);
}

closure.foo (void *this)
{
    int x [value-expr: (__closptr)->x];
    struct CLOSURE.closure.foo *__closptr;

    __closptr = (CLOSURE.closure.foo *) _d_allocmemory (8);
    __closptr->__chain = 0B;
    __closptr->x = 7;
    return <retval> = {.object=__closptr, .func=closure.foo.bar};
}
```

Function Frames

- Where a closure is not required, a frame is instead generated.

```
void bar()
{
    int add = 2;
    scope dg = (int a) => a + add;
    assert(dg(5) == 7);
}
```

Generated GENERIC Code

```
frame.bar.__lambda1 (void *this)
{
    return <retval> = a + ((FRAME.frame.bar *) this)->add;
}

frame.bar ()
{
    struct  dg;
    int add [value-expr: (&__frame)->add];
    struct FRAME.frame.bar __frame;

    __frame.__chain = 0B;
    (&__frame)->add = 2;
    dg = {.object=&__frame, .func=frame.bar.__lambda1};
    if (dg.func (dg.object, 5) == 7)
    {
        0
    }
    else
    {
        _d_assert ({.length=6, .ptr="test.d"}, 7);
    }
}
```

GCC Built-in Functions and Types

- **gcc.builtins** gives access to built-ins provided by the GCC backend.

```
import gcc.builtins;

void test()
{
    real r = 0.5 * __builtin_sqrtl(real.min_normal);

    if (__builtin_expect (cast(long) r == 0, true))
        __builtin_printf("Hello World!\n");
}
```

Generated GENERIC Code

- Allows many C library calls to be optimised in certain cases.

```
builtins.test ()
{
    real r;

    r = 9.16801933777423582810706196024241582978182485679283618642e-2467;
    {
        if (__builtin_expect ((long) r == 0, 1) != 0)
        {
            __builtin_puts ("Hello World!");
        }
    }
}
```

Built-in Types

- Defines aliases to internal types.

```
__builtin_va_list;          // Target C va_list type.
__builtin_clong;            // Target C long int type.
__builtin_culong;           // Target C long unsigned int type.
__builtin_machine_byte;     // Signed type whose size is equal to sizeof(unit).
__builtin_machine_ubyte;    // Unsigned variant.
__builtin_machine_int;      // Signed type whose size is equal to sizeof(word).
__builtin_machine_uint;     // Unsigned variant.
__builtin_pointer_int;      // Signed type whose size is equal to sizeof(pointer).
__builtin_pointer_uint;     // Unsigned variant.
__builtin_unwind_int;       // Target C _Unwind_Sword type, for EH.
__builtin_unwind_uint;      // Target C _Unwind_Word type, for EH.
```

Implementing D Intrinsics

- DMD has several intrinsics to the compiler.

```
import core.bitop;
import core.math;

void main()
{
    long l;
    l = rndtol (4.5);

    size_t[2] a = [2, 256];
    btc(a.ptr, 35);
}
```

Generated GENERIC Code

- **core.math** intrinsics are mapped to GCC builtin-ins.
- **core.bitop** instrinsics are expanded with inlined generated code.

```
int D main()
{
    int D.2001;
    ulong a[2];
    long l;

    l = 0;
    l = (long) __builtin_llroundl (4.5e+0);

    a[0] = 2;
    a[1] = 256;
    D.2001 = (*(ulong *) &a & 34359738368) != 0 ? -1 : 0;
    *(ulong *) &a = *(ulong *) &a ^ 34359738368;

    return <retval> = 0;
}
```

Extending D Intrinsics

- Many functions defined in **core.stdc** are mapped to GCC built-ins.
- Functions recognised as a GCC built-in can be optimised.
- Can be turned off with -**fno-builtin** switch.

```
import core.stdc.stdio;
import core.stdc.math;

void test()
{
    real r = powl(3, 3);

    if (r == 27.0)
        printf("Match!\n");
}
```

```
intrinsic.test()
{
    real r;

    r = 2.7e+1;
    {
        if (r == 2.7e+1)
        {
            __builtin_puts ("Match!");
        }
    }
}
```

Variadic Functions

- The va_list type has an exclusive meaning in the compiler.
- Matches the C ABI, type is not a void*.
- Defined in **gcc.builtins**, then an alias to the type in **core.stdc.stdarg**.
- Special va functions expanded at compile-time.

Variadic Functions

```
import core.stdc.stdarg;

void variadic(...)
{
  auto a1 = va_arg!(int)(_argptr);
  auto a2 = va_arg!(double)(_argptr);
  auto a3 = va_arg!(int[2])(_argptr);
  auto a4 = va_arg!(string)(_argptr);
}
```

Generated GENERIC Code

```
valist.variadic (struct TypeInfo_Tuple & _arguments_typeinfo)
{
  struct  _argptr[1];
  struct  a4;
  int a3[2];
  double a2;
  int a1;
  struct  _arguments;

  __builtin_va_start (&_argptr, _arguments_typeinfo);
  try
    {
      _arguments = _arguments_typeinfo->elements;
      a1 = VA_ARG_EXPR <_argptr>;
      a2 = VA_ARG_EXPR <_argptr>;
      a3 = VA_ARG_EXPR <_argptr>;
      a4 = VA_ARG_EXPR <_argptr>;
    }
  finally
    {
      __builtin_va_end (&_argptr);
    }
}
```

GCC Attributes

- Used to be accessible via pragmas in the language.
- Now uses UDA syntax that gets handled by **gcc.attributes**.

```
import gcc.attributes;
import gcc.builtins;

@attribute("noreturn")
void die()
{
    __builtin_unreachable();
}
```

GCC Type Attributes

- Attributes can also be applied to types.

```
import gcc.attributes;

@attribute("aligned")
struct A
{
    char c;
    int i;
}

@attribute("unused") int unused_var;
```

- As of writing, none of these attributes are implemented in GDC.

GCC Extended Assembly

- GDC implements a variant of GCC Extended Assembly.
- Extended assembly allows you to optionally specify the operands.

```
asm {
    "rdtsc"
    : /* output operands */
    : /* input operands  */
    : /* list of clobbered registers */;
}
```

Benefits of Extended Assembly

- It is available on nearly all targets.
- Instruction templates can be generated through CTFE string constants.
- Does not prevent a function from being inlined.
- Can have some common optimisations applied to them, such as DCE.

Future Plans

Compiler: Short Term

- Removing last of DMD-backend facing code from DFE.
- Find a workable solution for TLS support.
- Better support for LTO.

Compiler: Short Term

- Removing last of DMD-backend facing code from DFE.
- Find a workable solution for TLS support.
- Better support for LTO.

Compiler: Short Term

- Removing last of DMD-backend facing code from DFE.
- Find a workable solution for TLS support.
- Better support for LTO.

Compiler: Long Term

- Kickstart testing of more targets with D2.

- Implement missing optimisation features of D.
  - Named return value optimisation.
  - POD struct types.

- Integration of DFE into GCC garbage collector.

Compiler: Long Term

- Kickstart testing of more targets with D2.

- Implement missing optimisation features of D.
  - Named return value optimisation.
  - POD struct types.

- Integration of DFE into GCC garbage collector.

Compiler: Long Term

- Kickstart testing of more targets with D2.
- Implement missing optimisation features of D.
  - Named return value optimisation.
  - POD struct types.
- Integration of DFE into GCC garbage collector.

Compiler: Long Term

- Kickstart testing of more targets with D2.
- Implement missing optimisation features of D.
  - Named return value optimisation.
  - POD struct types.
- Integration of DFE into GCC garbage collector.

Compiler: Long Term

- Kickstart testing of more targets with D2.
- Implement missing optimisation features of D.
  - Named return value optimisation.
  - POD struct types.
- Integration of DFE into GCC garbage collector.

Compiler: Wishlist

- Add support for label operands in Extended Assembly.

```
int frob(int x)
{
  int y;
  asm {
      "frob %%r5, %1;
       jc %l[Lerror];
       mov (%2), %%r5"
      :
      : "r"(x), "r"(&y)
      : "r5", "memory"
      : Lerror;
  }
  return y;

Lerror:
  return -1;
}
```

Library

- Implement Exception Chaining.
- Conversion of D IASM to Extended Assembly.
- Finish off port of ARM.
- Fix D GC runtime for TLS support.

Library

- Implement Exception Chaining.
- Conversion of D IASM to Extended Assembly.
- Finish off port of ARM.
- Fix D GC runtime for TLS support.

Library

- Implement Exception Chaining.
- Conversion of D IASM to Extended Assembly.
- Finish off port of ARM.
- Fix D GC runtime for TLS support.

Library

- Implement Exception Chaining.
- Conversion of D IASM to Extended Assembly.
- Finish off port of ARM.
- Fix D GC runtime for TLS support.

It is vital that we begin testing on, and gain support for more target architectures and platforms.

http://gdcproject.org

http://gdcproject.org/wiki

http://bugzilla.gdcproject.org

ibuclaw@gdcproject.org

Questions?