How The Flang Frontend Works

Introduction to the interior of the Open–Source Fortran frontend for LLVM

Paul Osmialowski*
Arm
Development Solutions Group
Manchester, UK
pawel.osmialowski@arm.com

ABSTRACT

In May 2017, PGI® publicized Flang [16][7], an Open–Source Fortran frontend for LLVM along with a complementary runtime library. The ultimate goal set for Flang is to make it part of the whole LLVM ecosystem with level of support and attention equal to that experienced by the Clang frontend. To come closer to this goal it is important to make Flang widely known and more visible. A good introduction to the frontend interior could serve such a purpose and the intention of this paper is to describe how Flang works and how its source code is structured.

ACM Reference Format:
Paul Osmialowski. 2017. How The Flang Frontend Works. In *Proceedings of LLVM-HPC'17: Fourth Workshop on the LLVM Compiler Infrastructure in HPC, Denver, CO, USA, November 12–17, 2017 (LLVM-HPC'17),* 14 pages.
https://doi.org/10.1145/3148173.3148183

1 INTRODUCTION AND MOTIVATION

Through the decades of its existence, the Fortran programming language has proven to be an important language for numerical and scientific computing [2][8]. Among its advantages the most important are:

- A large code base created during decades of investment in scientific software.
- Wide acceptance in scientific environment resulting in a large number of available experts.
- Good expressiveness for describing numerical algorithms.
- Good efficiency of compiled code.
- Portability across different platforms.

These features guarantee further development of the Fortran applications. Utilising the LLVM compiler family in the High Performance Computing (HPC) world demands the existence of a Fortran compiler frontend for it.

*Osmialowski has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 671697.

LLVM-HPC'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5565-0/17/11…$15.00
https://doi.org/10.1145/3148173.3148183

There were a few different attempts to introduce Fortran frontend into the LLVM compilers infrastructure [18][9][4]. The most complete is the Flang project developed by NVIDIA's PGI® [6][12]. After it was made publicly available [16][7], the next logical step would be to make it a part of the LLVM project on equal rights with the Clang compiler. This appears to be uneasy task since both of these projects were developed apart from each other for most of the time with totally different design goals in mind. The purpose of this article is to reveal the extent of the differences – this could help the wider audience to decide about the possibility of Flang being accepted as a member of LLVM family.

Being a part of the LLVM ecosystem and using its backend, Flang can target multiple hardware architectures. Nowadays support is present for AArch64, 64-bit Power and x86_64 architectures, but it should be fairly easy to add any other 64-bit hardware architecture supported by LLVM to this list.

The Flang project is open for communication with the wider public. The GitHub *flang-compiler* account [12] holds all of the source code and documentation, it is also used for bug tracking – this proved to be a good communication channel with the developers. To improve the communication even further, The *flang-compiler* channel was opened on the Slack on–line service [13]. This is an invitation only service, but anyone can join [14].

Since it was made public, Flang gained lots of attention and became tested for conformance with Fortran standards and performance of generated code. With its source code based on PGI®'s commercial Fortran compiler, Flang inherited most of the interior concepts: two–pass parser, semantic analyzer, two levels of internal representation (ILM, ILI) and internal optimizer. Having the code base developed for more than two decades, such heritage offers maturity and good Fortran standards conformance [1].

Figures 1 and 2 depict how the choice of Fortran compiler (between GFortran and Flang) affects performance of compiled workloads. Note that in case of Flang, most of the optimization is done within LLVM passes and this arrangement (Fortran frontend combined with LLVM middle–end and backend) is still very young, there is room for further improvement.

The rest of the article is organized as follows. Section 2 describes the Flang pipeline operation between its invocation by the frontend driver (Clang) and the LLVM IR code generation. Section 3 focuses more on the source code and its structure. Section 4 provides a *Hello World* example and describes how it is processed. Section 5 guides through the command line parameters accepted while invoking the two stages of the Flang frontend. The article is concluded in Section 6.

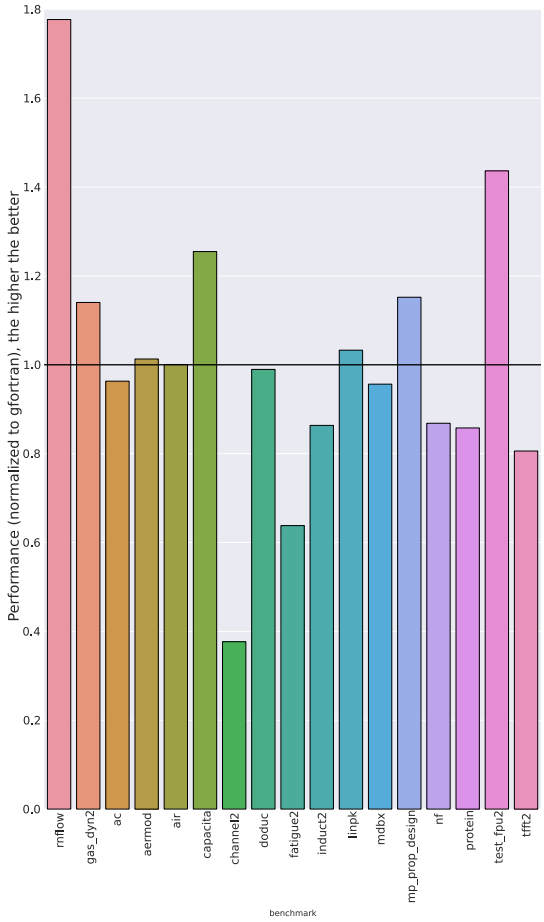

Figure 1: Execution time differences between workloads compiled by Flang and workloads compiled by GFortran. Workloads were from the Polyhedron [17] suite of benchmarks (downloaded from http://www.fortran.uk) executed on single core of an AArch64 machine equipped with Cortex–A57 cores. Both compilers were invoked with flags *-O3 -ffp–contract=fast -funroll–loops*.

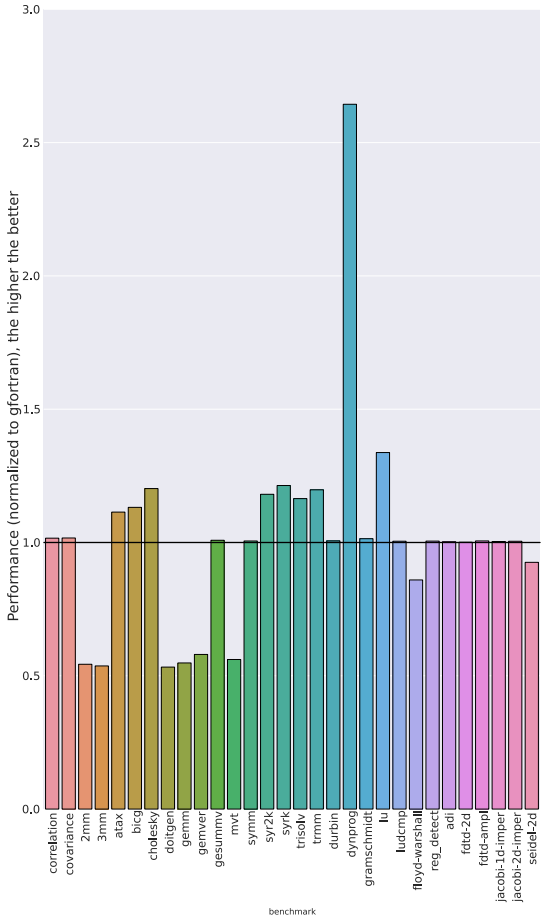

Figure 2: Execution time differences between workloads compiled by Flang and workloads compiled by GFortran. Worlkoads were from the PolyBench/Fortran [15] suite of benchmarks executed on single core of an AArch64 machine equipped with Cortex–A57 cores. Both compilers were invoked with flags *-O3 -ffp–contract=fast -funroll–loops*.

2 THE PIPELINE – HOW FLANG FITS INTO LLVM

The Flang project is split into two major parts. The first part is the frontend driver which comprises of a series of patches held on top of Clang's git repository fork on GitHub [3]. This part is responsible for parsing Flang and Fortran specific command line options and establishing the whole of the pipeline.

The second part is the frontend itself and its sources are held in a separate git repository on GitHub [11]. When compiled from the sources, it builds two executables (*flang1* and *flang2*), the first reads Fortran code and produces ILM (the first Internal Representation), the second reads ILM and produces ILI internally (the second Internal Representation which also undergoes optimization steps) from which LLVM IR is finally generated.

Figure 3 presents a broad view of the Fortran program compilation process. A more detailed list of functional blocks is as follows:

- Flang1 phases:
  scanner, turns Fortran code into tokens.
  parser, turns tokens into an AST and a symbol table.
  transformer, turns the AST into a canonical AST.
  output, turns the canonical AST into ILM.
- Flang2 phases:
  expander, turns ILM into ILI.
  optimizer, turns ILI into optimized ILI.

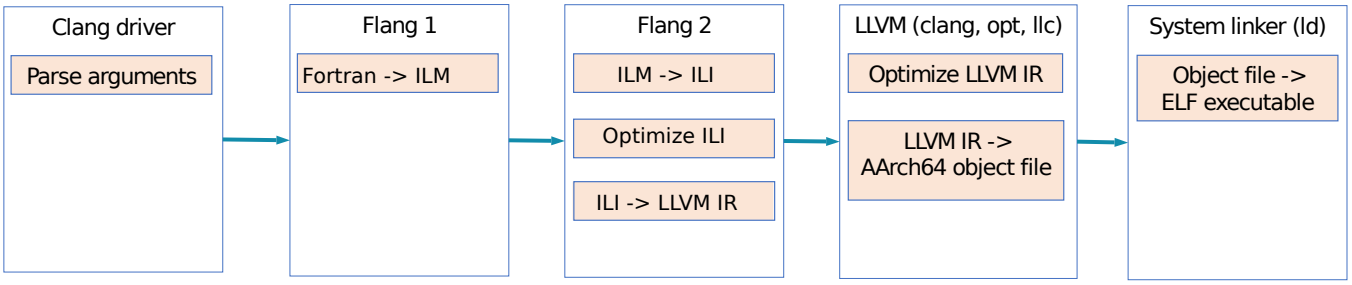

Figure 3: The Flang's pipeline.

the bridge, turns optimized ILI into LLVM IR.

The Fortran program is read from an input file by the *flang1* executable started by frontend driver. Within *flang1*, the scanner turns Fortran program into tokens, then the parser turns those tokens into an Abstract Syntax Tree (AST) and symbol table (symtab). The AST is then transformed into canonical form which is used to generate temporary a text file with ILM code – the first Intermediate Representation used by Flang.

The ILM code is read from the temporary input text file by the *flang2* executable. During the import stage its text content is fed into data structures held in the process memory. This form is turned into ILI, the second Intermediate Representation used by Flang, which is then optimized by internal optimizer. The optimized ILI is then used during the LLVM bridge phase in which final output text file with LLVM IR is generated.

2.1 Fortran 90 Modules

For each module defined in a given Fortran 90 source file, a *.mod* file is created along with the binary *.o* file produced for a given Fortran source file. If a source file defines more than one module, a single *.o* file is created accompanied by several *.mod* files. The role of Fortran 90 modules is similar to C/C++ header files (except that no preprocessor is involved and they are imported by Fortran 90 programs with a *use* statement) and they are typically installed into *include* directories along with C/C++ header files (while compiled library binaries containing actual executable code are held in *lib* directories). In the case of GFortran, *.mod* files are binaries. In the case of Flang, *.mod* files are text files containing exported symbols definitions (yet still not very human readable) produced by *flang1* (and not processed any further by *flang2*). Modules compiled with one compiler cannot be used by code compiled with the other compiler.

Listing 1 contains example module code while the listing 2 contains generated *.mod* file for this example module code.

```
module somemod
  implicit none
  real(kind = 8) :: pi
  save
contains
  integer(kind = 4) function somefun(a, b, c)
    integer(kind = 4), intent(in) :: a, b, c
    somefun = a * b * c
  end function somefun
end module somemod
```

Listing 1: Example of Fortran 90 module code.

```
V30 :0 x4 somemod
11 somemod.f90 S624 0
07/16/2017  11:05:13
enduse
S 624 24 0 0 0 8 1 0 5015 10005 0 A 0 0 0 0 B 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 somemod
S 626 6 4 0 0 9 1 624 5023 4 0 A 0 0 0 0 B 0 0 0 0 0 0 0
      0 0 0 0 0 627 0 0 0 0 0 0 0 0 0 0 0 624 0 0 0 0 pi
S 627 11 0 0 0 8 1 624 5026 40800000 805000 A 0 0 0 0 B 0
      0 0 0 0 8 0 0 626 626 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 _somemod$2
S 628 23 5 0 0 6 632 624 5037 4 0 A 0 0 0 0 B 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 somefun
S 629 1 3 1 0 6 1 628 5045 4 3000 A 0 0 0 0 B 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 a
S 630 1 3 1 0 6 1 628 5047 4 3000 A 0 0 0 0 B 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 b
S 631 1 3 1 0 6 1 628 5049 4 3000 A 0 0 0 0 B 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 c
S 632 14 5 0 0 6 1 628 5037 4 400000 A 0 0 0 0 B 0 0 0 0
      0 0 0 2 3 0 0 633 0 0 0 0 0 0 0 0 0 6 0 624 0 0 0 0
      somefun
F 632 3 629 630 631
S 633 1 3 0 0 6 1 628 5037 4 1003000 A 0 0 0 0 B 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      somefun
Z
Z
```

Listing 2: Example of *.mod* file generated by *flang1* for Fortran 90 module code.

3 THE CODE AND SOURCES DIRECTORY TREE

3.1 Directory tree

The directory tree holding Flang's source code is depicted on Figure 4. Since the building process is organized by CMake [5], every source code directory contains the usual *CMakeLists.txt* file. The out–of–source building process is recommended for projects utilizing CMake infrastructure [10].

The code is mostly written in ANSI C98, some C++ files do exist and they are all compiled with *–std=c++11* compiler flag. There are also Fortran routines (e.g. *ftn_transpose_real*, *ftn_transpose_cmplx*, *vmmul_real*, *vmmul_cmplx*) in the runtime library. Various definition files are provided in *nroff* format (*.n* files) from which certain C headers (*.h* files) are generated. Some of the build–time utilities serve that purpose (*astutil*, *machar*, *fen2rst*, *n2rst*, *ilmtp*, *ilitp*, *fsymutil*, *fesymutil*, *fsymini*, *fesymini*, *fmachar*). The documentation source files are also provided in *nroff* format (*.n* files).

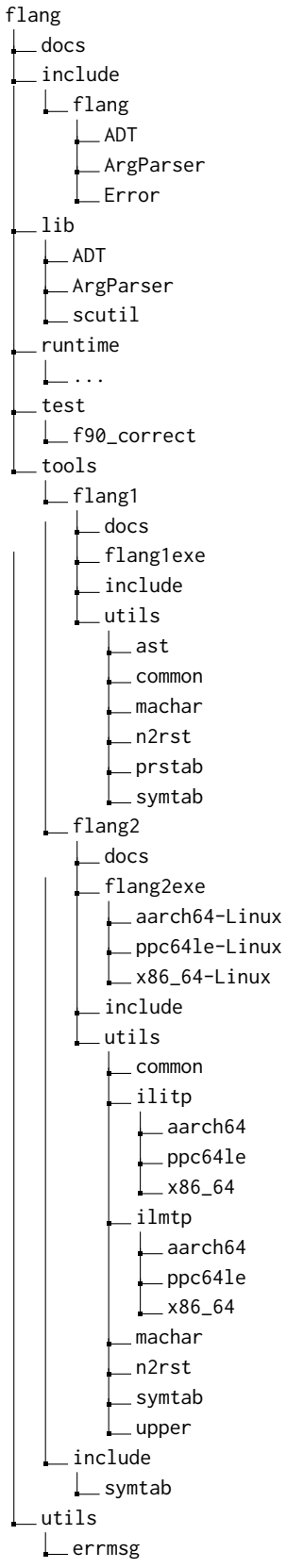

Figure 4: Flang frontend directory structure.

Flang project source code directory structure can be described as follows:

- *lib*, *include* – auxiliary libraries shared by *flang1*, *flang2* and the Fortran runtime library.
- *runtime* – the Fortran runtime library.
- *tools/flang1* – *flang1*, Fortran to ILM translator.
- *tools/flang2* – *flang2*, ILM to LLVM IR translator.
- *tools/include/symtab* – *gbldefs.h* (one of three in the tree) and *global.h* (one of four in the tree).
- *utils* – the *errmsggen* (*errmsg.cpp*) utility, used to generate two *errmsgdf.h* files (for *flang1* and *flang2*) – error message definitions.

The *lib* directory structure presents as follows:

- *ADT* (Abstract Data Types) – contains *hash.c* – open addressing, quadratically probed hash tables.
- *ArgParser* – command line arguments parser routines, common to *flang1* and *flang2*.
- *scutil* – various routines, mostly sophisticated file I/O handling.

The *tools/flang1* directory structure presents as follows:

- *flang1exe* – the frontend, stage 1.
- *include/platform.h.in* – template for *platform.h* header file generated by CMake (platform specific definitions).
- *utils* – utilities required during *flang1* building process:
  ast – *astutil*, generates *ast.h* and *astdf.h* from *ast.in.h*.
  common – *utils.h*, *utils.cpp*, nroff–to–C converter used by *astutil*, *machar* and *fen2rst*.
  machar – *machar.n*, *machar.cpp*, machine characteristics utility definitions, used for generating *machar.h* and *machardf.h*.
  n2rst – *n2rst.cpp*, nroff (.n) to Shpinx (.rst) format conversion utility.
  prstb – parse table generator; grammar definition and parser tables.
  symtab – symbol table generator and utilities.

The *tools/flang2* directory structure presents as follows:

- *flang2exe* – the frontend, stage 2.
- *include/platform.h.in* – template for *platform.h* header file generated by CMake (platform specific definitions).
- *utils* – utilities required during *flang2* building process:
  common – *utils.h*, *utils.cpp*, nroff–to–C converter used by *ilitp*, *ilmtp*, *fsymutil*, *fesymutil*, *fsymini*, *fesymini* and *fmachar*.
  ilmtp – *ilmtp.cpp*, ILM template utility; reads ILM definition file (*ilmtp.n*) and generates *ilmtp.h* and *ilmtpdf.h*; contains architecture specific *.n* files for aarch64, ppc64le and x86_64.
  ilitp – *ilitp.cpp*, ILI template utility; reads ILI definition file (*ilitp.n*) and generates *iliatt.h* and *ilinofodf.h*; contains architecture specific *.n* files for aarch64, ppc64le and x86_64.
  machar – see previous.
  n2rst – see previous.
  symtab – see previous.
  upper – *upperl* utility, turns *upperilm.in* into *upperilm.h*; included by *flang2exe/upper.c* which imports the lowered Fortran code.

The directory structure contains some duplication. We can observe some utilities (*machar*, *n2rst*, *symtab*) repeated in both *tools/flang1* and *tools/flang2* directories. It is expected that more general versions of these tools should emerge in time and they will be moved to the top–level *utils* directory (a similar move was already done to the *errmsggen* utility). Detailed lists of files in *tools/flang1/flang1exe* and *tools/flang2/flang2exe* also contain some duplication. From an LLVM developers point of view, the most important difference is the presence of *ll*–prefixed filenames in the *tools/flang2/flang2exe* directory, these files are parts of the LLVM IR bridge code: *ll_builder.c*, *ll_builder.h*, *ll_dbgutl.c*, *ll_ftn.c*, *ll_structure.c*, *ll_structure.h*, *ll_write.c*, *ll_write.h*, *llassem.c*, *llassem.h*, *llassem_common.c*, *lldebug.c*, *lldebug.h*, *llmputil.c*, *llmputil.h*, *llopt.c llsched.c llutil.c* and *llutil.h*.

The LLVM IR generation is performed in terms of explicit operations on text strings. Listing 3 presents an example of code translating an operation identifier into its textual representation.

```
static const char *
get_op_name(enum LL_Op op)
{
  switch (op) {
  case LL_FPTRUNC:
    return "fptrunc";
  case LL_FPEXT:
    return "fpext";
  case LL_SEXT:
    return "sext";
  case LL_ZEXT:
    return "zext";
  case LL_TRUNC:
    return "trunc";
  case LL_BITCAST:
    return "bitcast";
  case LL_SITOFP:
    return "sitofp";
  case LL_UITOFP:
    return "uitofp";
  case LL_FPTOSI:
    return "fptosi";
  case LL_FPTOUI:
    return "fptoui";
  case LL_ADD:
    return "add";
  case LL_FADD:
    return "fadd";
  case LL_SUB:
    return "sub";
  case LL_FSUB:
    return "fsub";
  case LL_MUL:
    return "mul";
  case LL_FMUL:
    return "fmul";
  case LL_UDIV:
    return "udiv";
  case LL_SDIV:
    return "sdiv";
  case LL_SREM:
    return "srem";
  case LL_UREM:
    return "urem";
  case LL_FDIV:
    return "fdiv";
  case LL_OR:
    return "or";
  case LL_ASHR:
    return "ashr";
  case LL_LSHR:
    return "lshr";
  case LL_AND:
    return "and";
  case LL_XOR:
    return "xor";
  case LL_SHL:
    return "shl";
  case LL_INTTOPTR:
    return "inttoptr";
  case LL_PTRTOINT:
    return "ptrtoint";
  case LL_ICMP:
    return "icmp";
  case LL_FCMP:
    return "fcmp";
  default:
    return "thisisnotacceptable";
  }
}
```

Listing 3: Example of code translating operation identifier into its textual representation.

3.2 Documentation

The documentation source files are also provided in *nroff* format (*.n* files). Various command–line utilities (e.g. *groff* or *a2ps*) can be used to generate documentation pages from these source files. The same format is used for various definition files (*.n* files) from which certain C headers (*.h* files) are generated. This ensures that all generated entities are also well documented.

Directory tree contains various *.n* files (*nroff* format) with meaningful names (one chapter - one *.n* file), see Listing 4 for full list of provided documentation source files.

```
tools/flang1/docs/ast.n
tools/flang1/docs/commopt.n
tools/flang1/docs/comms.n
tools/flang1/docs/controller.n
tools/flang1/docs/dinit.n
tools/flang1/docs/grammar.n
tools/flang1/docs/intro.n
tools/flang1/docs/outconv.n
tools/flang1/docs/output.n
tools/flang1/docs/parser.n
tools/flang1/docs/scanner.n
tools/flang1/docs/semant.n
tools/flang1/docs/titlep.n
tools/flang1/docs/transform.n
tools/flang1/utils/ast/ast.n
tools/flang1/utils/machar/machar.n
tools/flang1/utils/symtab/symini_ftn.n
tools/flang1/utils/symtab/symtab.n
tools/flang2/docs/coding.n
tools/flang2/docs/controller.n
tools/flang2/docs/dinit.n
tools/flang2/docs/error.n
tools/flang2/docs/expander.n
tools/flang2/docs/fin.n
tools/flang2/docs/ili.n
tools/flang2/docs/ilm.n
tools/flang2/docs/intro.n
tools/flang2/docs/register.n
tools/flang2/docs/xflag.n
tools/flang2/docs/xref.n
tools/flang2/utils/ilitp/aarch64/ilitp.n
tools/flang2/utils/ilitp/aarch64/ilitp_longdouble.n
tools/flang2/utils/ilitp/ilitp_atomic.n
tools/flang2/utils/ilitp/ppc64le/ilitp.n
tools/flang2/utils/ilitp/ppc64le/ilitp_float128.n
tools/flang2/utils/ilitp/ppc64le/ilitp_longdouble.n
tools/flang2/utils/ilitp/x86_64/ilitp.n
tools/flang2/utils/ilitp/x86_64/ilitp_longdouble.n
tools/flang2/utils/ilmtp/aarch64/ilmtp.n
tools/flang2/utils/ilmtp/aarch64/ilmtp_atomic.n
tools/flang2/utils/ilmtp/aarch64/ilmtp_longdouble.n
```

```
tools/flang2/utils/ilmtp/ppc64le/ilmtp.n
tools/flang2/utils/ilmtp/ppc64le/ilmtp_atomic.n
tools/flang2/utils/ilmtp/ppc64le/ilmtp_longdouble.n
tools/flang2/utils/ilmtp/x86_64/ilmtp.n
tools/flang2/utils/ilmtp/x86_64/ilmtp_atomic.n
tools/flang2/utils/ilmtp/x86_64/ilmtp_longdouble.n
tools/flang2/utils/machar/machar.n
tools/flang2/utils/symtab/symini_ftn.n
tools/flang2/utils/symtab/symtab.n
```

Listing 4: List of provided documentation source files.

3.3 Coding convention

As declared in point 1.3 of *tools/flang2/docs/coding.n* document, the established coding convention is as follows:

- ANSI C98 with some K&R code that will eventually be removed.
- Declared proper use of static keyword for limiting visibility of file globals and file functions.
- Naming convention for files: files that define external symbols have *df* prefix, headers that declare external symbols also have *df* prefix, e.g. *ilidf.c*, *flgdf.h*.
- Naming convention for code: variable names are lower case, macros and *typedefs* are typically upper case, *enums* begin with a capital letter, then are lower case (no explicit rule stated for function names though!).
- The first non-comment line of each *.c* file is *#include "gbldefs.h"* (note that there are three *gbldefs.h* files in the directory tree!).
- All dynamic storage allocation and freeing is done through the macros *NEW*, *NEED* and *FREE* defined in *gbldefs.h*.

Although the listed items seem to be reasonable, they do not prevent surprising constructs from appearing in the Flang code. One example of those surprises is the re–defined *assert* macro (see Listing 5). It does not only reuse usual *assert* macro name, it also changes its interface by introducing additional parameters.

```
/** \brief Assert that cond is true, and emit an internal
    compiler error
 * otherwise.
 *
 * Note that unlike the C standard <assert.h> assert()
    macro, this version is
 * active in both debug and release builds, and expands
    to a statement,
 * not an expression.
 */
#define assert(cond, txt, val, sev) \
  if (cond)                         \
    ;                               \
  else                              \
  interr((txt), (val), (sev))
```

Listing 5: Re–defined assert macro.

3.4 Multi–platform compatibility

In order to achieve portability across hardware architectures, the *platform.h* file is generated by the CMake from *platform.h.in* template file (listing 6 presents example *platform.h* file generated by CMake for a GNU/Linux system running on AArch64 hardware).

```
#include "define.h"

#define VHOST "Linux"
#define TARGET "aarch64"

/* FIXME:
 *  - filtered for pop
 *  - true for all arch/arch
 */
#define BIGOBJ 1
#define PTRCOMMON 1
#define PGF90 1
#define NATIVE 1
#define NOLZ 1

#define HOST_ARM 1
#define HOST_LINUX 1
#define HOST_LINUX_ARM 1
#define HOST_LINUX_ARM64 1
#define TARGET_ARM 1
#define TARGET_LLVM 1
#define TARGET_LLVM_ARM 1
#define TARGET_LLVM_ARM64 1
#define TARGET_LLVM_64 1
#define TARGET_LINUX 1
#define TARGET_LINUX_ARM 1
#define TARGET_LINUX_ARM64 1
```

Listing 6: Example *platform.h* file generated for an AArch64 system.

It should be noted that despite workable support for three hardware architectures (AArch64, 64-bit Power and x86_64), the original Flang code evolved with strict ties to x86_64 and recently added support for other architectures is implemented with reuse of x86_64 features. Listing 7 shows example of x86_64 definitions reused for AArch64 features. Somewhat detached from the list of supported architectures is the array of known target triples, see Listing 8.

```
/***** ARM -- recycle FEATURE_ x64/x86 manifests *****/
#if defined(TARGET_LLVM_ARM)
#define FEATURE_SCALAR_NEON FEATURE_SCALAR_SSE
#define FEATURE_NEON FEATURE_SSE
#define FEATURE_FMA FEATURE_FMA3
#endif
```

Listing 7: Example of x86_64 features reuse.

```
static const struct triple_info known_triples[] = {
    /* These prefixes are tried in order against
     target_triple. */
    {"nvptx64-", "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
     i32:32:32-i64:64:64"
                 "-f32:32:32-f64:64:64-v16:16:16-v32
     :32:32-v64:64:64-v128:128:"
                 "128-n16:32:64"},
    {"spir64-", "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32
     :32:32-i64:64:64"
                 "-f32:32:32-f64:64:64-v16:16:16-v24
     :32:32-v32:32:32-v48:64:64"
                 "-v64:64:64-v96:128:128-v128:128:128-v192
     :256:256"
                 "-v256:256:256-v512:512:512-v1024
     :1024:1024"},
    {"i386", "e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"},
    {"x86_64-", "e-p:64:64-i64:64-f80:128-n8:16:32:64-
     S128"},
    {"armv7-", "e-p:32:32-i64:64-v128:64:128-n32-S64"},
    {"aarch64-", "e-m:e-i64:64-i128:128-n32:64-S128"},
    {"powerpc64le", "e-p:64:64-i64:64-n32:64"},
    {"", ""}
};
```

Listing 8: Array of target triples.

4 HELLO WORLD EXAMPLE

In order to demonstrate how example Fortran code is processed by Flang, a short program can be used as seen in Listing 9.

```
program hello
  print *, 'hello world ', 12.34
end
```

Listing 9: *hello.f90* Example Fortran code.

Using the *-v* parameter we can instruct the frontend driver to print out all the commands with their arguments that it is about to execute in order to proceed with the compilation (see example in Listing 10). The meanings of particular *flang1* and *flang2* arguments are described in Section 5.

```
$ flang -v -c hello.f90
clang version 4.0.1 (https://github.com/flang-compiler/
    clang.git bcdf99e52b47e13a64504a5783ce4eed40833835)
    (http://llvm.org/git/llvm.git
    f3d3277bb713bb8aced9a7ac2e9b05c52d2844ee)
Target: aarch64-unknown-linux-gnu
Thread model: posix
Candidate multilib: .;@m64
Selected multilib: .;@m64

"flang1" hello.f90 -opt 0 -terse 1 -inform warn -nohpf -
    nostatic -y 129 2 -inform warn -x 19 0x400000 -quad
    -x 59 4 -x 15 2 -x 49 0x400004 -x 51 0x20 -x 57 0x4c
     -x 58 0x10000 -x 124 0x1000 -tp px -x 57 0xfb0000 -
    x 58 0x78031040 -x 47 0x08 -x 48 4608 -x 49 0x100 -
    stdinc $(dirname  which flang )/../include:/usr/
    local/include:$(dirname  which flang )/../lib/clang
    /4.0.1/include:/usr/include/aarch64-linux-gnu:/
    include:/usr/include -def unix -def __unix -def
    __unix__ -def linux -def __linux -def __linux__ -def
     __NO_MATH_INLINES -def __LP64__ -def __x86_64 -def
    __x86_64__ -def __LONG_MAX__=9223372036854775807L -
    def __SIZE_TYPE__=unsigned long int -def
    __PTRDIFF_TYPE__=long int -def __THROW= -def
    __extension__= -def __amd_64__amd64__ -def __k8 -def
     __k8__ -def __PGLLVM__ -freeform -vect 48 -y 54 1 -
    x 70 0x40000000 -y 163 0xc0000000 -x 189 0x10 -
    stbfile /tmp/hello-f2fb4a.stb -modexport /tmp/hello-
    f2fb4a.cmod -modindex /tmp/hello-f2fb4a.cmdx -output
     /tmp/hello-f2fb4a.ilm

"flang2" /tmp/hello-f2fb4a.ilm -ieee 1 -x 6 0x100 -x 42 0
    x400000 -y 129 4 -x 129 0x400 -fn hello.f90 -opt 0 -
    terse 1 -inform warn -y 129 2 -inform warn -x 51 0
    x20 -x 119 0xa10000 -x 122 0x40 -x 123 0x1000 -x 127
     4 -x 127 17 -x 19 0x400000 -x 28 0x40000 -x 120 0
    x10000000 -x 70 0x8000 -x 122 1 -x 125 0x20000 -quad
     -x 59 4 -tp px -x 120 0x1000 -x 124 0x1400 -y 15 2
    -x 57 0x3b0000 -x 58 0x48000000 -x 49 0x100 -astype
    0 -x 183 4 -x 121 0x800 -x 54 0x10 -x 70 0x40000000
    -x 249 40 -x 124 1 -y 163 0xc0000000 -x 189 0x10 -y
    189 0x4000000 -x 183 0x10 -stbfile /tmp/hello-f2fb4a
    .stb -asm /tmp/hello-f2fb4a.ll

"clang-4.0" -cc1 -triple aarch64-unknown-linux-gnu -emit-
    obj -mrelax-all -disable-free -disable-llvm-verifier
     -discard-value-names -main-file-name hello.f90 -
    mrelocation-model static -mthread-model posix -
    mdisable-fp-elim -fmath-errno -masm-verbose -
    mconstructor-aliases -fuse-init-array -target-cpu
    generic -target-feature +neon -target-abi aapcs -v -
    dwarf-column-info -debugger-tuning=gdb -coverage-
    notes-file ./hello.gcno -resource-dir $(dirname
    which flang )/../lib/clang/4.0.1 -fdebug-compilation
    -dir . -ferror-limit 19 -fmessage-length 316 -fallow
    -half-arguments-and-returns -fno-signed-char -fobjc-
    runtime=gcc -fdiagnostics-show-option -fcolor-
    diagnostics -o hello.o -x ir /tmp/hello-f2fb4a.ll

clang -cc1 version 4.0.1 based upon LLVM 4.0.1 default
    target aarch64-unknown-linux-gnu
```

Listing 10: Example execution of Flang invoked with *-v* parameter.

Figure 5 depicts how the most significant line of this example program is processed by Flang; it presents relevant fragments of AST, ILM, ILI and LLVM IR.

Listing 11 shows how this example program is turned into tokens (Section 5 explains how to instruct *flang1* and *flang2* to print human–readable representation of their internal structures).

```
tkntyp: PROGRAM tknval: 0 lineno: 1
tkntyp: <id name> tknval: 0 (hello) lineno: 1
tkntyp: END tknval: 0 lineno: 1
tkntyp: PRINT tknval: 0 lineno: 2
tkntyp: * tknval: 0 lineno: 2
tkntyp: , tknval: 0 lineno: 2
tkntyp: <quoted string> tknval: 626 lineno: 2
tkntyp: , tknval: 0 lineno: 2
tkntyp: <real> tknval: 1095069860 lineno: 2
tkntyp: END tknval: 0 lineno: 2
tkntyp: <END stmt> tknval: 0 lineno: 3
tkntyp: END tknval: 0 lineno: 3
```

Listing 11: Tokens extracted from example Fortran program.

The Fortran *print* instruction from the example program is turned during compilation into the following sequence of runtime library calls:

- Subroutine *f90io_src_info03* – initialize internal globals; starts every use of the *print* instruction.
- Function *f90io_print_init* – initialize output buffer.
- Function *f90io_sc_ch_ldw* – place character string into output buffer.
- Function *f90io_sc_f_ldw* – place floating point number into output buffer.
- Function *f90io_ldw_end* – end adding to output buffer and print its content; ends every use of the *print* instruction.

The sequence of calls above is reflected in the AST, Symbol Table and Intermediate Representation. Listing 12 shows the relevant part of the AST (calls to the *f90io_sc_ch_ldw* and the *f90io_sc_f_ldw* functions) as printed in human–readable form by *flang1*.

```
ident       hshlk/std:    0  type:integer  alias:    0
    callfg:0 opt=(0,0)
aptr:   20  sptr:  631 (z__io)

assign      hshlk/std:    2  type:integer opt=(0,0)
aptr:   21  dest:   20  src:   18

constant    hshlk/std:    0  type:integer opt=(0,0)
aptr:   22  sptr:  632 (12)

constant    hshlk/std:    0  type:character*12 opt=(0,0)
aptr:   23  sptr:  633 ("hello world ")

constant    hshlk/std:    0  type:real opt=(0,0)
aptr:   24  sptr:  634 (-9.2615258789E+02)

constant    hshlk/std:    0  type:integer opt=(0,0)
aptr:   25  sptr:  637 (14)

unaryop     hshlk/std:    0  type:integer  alias:    0
    callfg:0 opt=(0,0)
aptr:   26  lop :   25  optype:28

func-call   hshlk/std:    0  type:  alias:    0  callfg:1
    opt=(0,0)
```

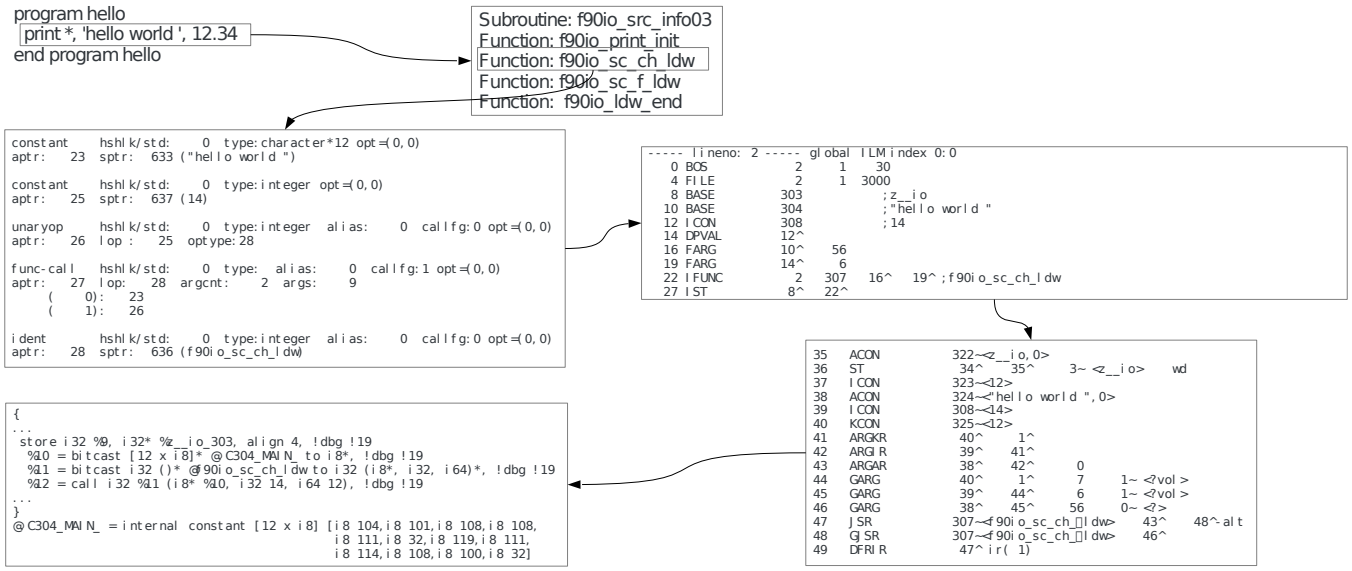

Figure 5: *Hello World* example.

```
aptr:  27  lop:  28  argcnt:  2  args:  9
     (  0):  23
     (  1):  26

ident      hshlk/std:  0  type:integer  alias:  0
     callfg:0 opt=(0,0)
aptr:  28  sptr:  636 (f90io_sc_ch_ldw)

assign     hshlk/std:  3  type:integer opt=(0,0)
aptr:  29  dest:  20  src:  27

unaryop    hshlk/std:  0  type:real  alias:  0
     callfg:0 opt=(0,0)
aptr:  30  lop :  24  optype:28

constant   hshlk/std:  0  type:integer opt=(0,0)
aptr:  31  sptr:  639 (27)

unaryop    hshlk/std:  0  type:integer  alias:  0
     callfg:0 opt=(0,0)
aptr:  32  lop :  31  optype:28

func-call  hshlk/std:  0  type:  alias:  0  callfg:1
      opt=(0,0)
aptr:  33  lop:  34  argcnt:  2  args:  12
     (  0):  30
     (  1):  32

ident      hshlk/std:  0  type:integer  alias:  0
     callfg:0 opt=(0,0)
aptr:  34  sptr:  638 (f90io_sc_f_ldw)
```

Listing 12: Relevant part of Abstract Syntax Tree of example Fortran program.

The *sptr* values are references to Symbol Table, the relevant parts of which are presented in human–readable form in Listing 13.

```
z__io integer
sptr:631 dtype:6 sc:1=LOCAL stype:6=variable
enclfunc:0 hashlk:0 scope:624=hello symlk:1=NOSYM
address:0
autobj:0 cmblk:0 cvlen:0 adjstrlk:0 descr:0 midnum:0
     newarg:0 devcopy:0 nmcnst:0 ptroff:0 sdsc:0 slnk:0
assn dcld hccsym seq
*** b4: 15 0xf

12 integer
sptr:632 dtype:6 sc:0=NONE stype:3=constant
enclfunc:0 hashlk:0 scope:1=NOSYM symlk:0
address:0 conval1g:0 conval2g:12 conval3g:0 conval4g:0

"hello world " character*12
sptr:633 dtype:58 sc:0=NONE stype:3=constant
enclfunc:0 hashlk:0 scope:1=NOSYM symlk:0
address:0 conval1g:5071 conval2g:0 conval3g:0 conval4g:0

-9.7322831928E+18 real
sptr:634 dtype:8 sc:0=NONE stype:3=constant
enclfunc:0 hashlk:0 scope:1=NOSYM symlk:0
address:0 conval1g:0 conval2g:1095069860 conval3g:0
     conval4g:0

f90io_ldw integer
sptr:635 dtype:6 sc:5=EXTERN stype:14=subroutine
enclfunc:0 hashlk:0 scope:0 symlk:1=NOSYM
dpdsc:0 intent:in paramct:0
gsame:0 fval:0 altname:0 slnk:0
dcld func hccsym indep nodesc sdscsafe sequent typd

f90io_sc_ch_ldw integer
sptr:636 dtype:6 sc:5=EXTERN stype:14=subroutine
enclfunc:0 hashlk:0 scope:0 symlk:1=NOSYM
dpdsc:0 paramct:0
gsame:0 fval:0 altname:0 slnk:0
dcld func hccsym nodesc pure

14 integer
sptr:637 dtype:6 sc:0=NONE stype:3=constant
enclfunc:0 hashlk:0 scope:1=NOSYM symlk:0
address:0 conval1g:0 conval2g:14 conval3g:0 conval4g:0

f90io_sc_f_ldw integer
sptr:638 dtype:6 sc:5=EXTERN stype:14=subroutine
enclfunc:0 hashlk:636=f90io_sc_ch_ldw scope:0 symlk:1=
     NOSYM
dpdsc:0 paramct:0
gsame:0 fval:0 altname:0 slnk:0
dcld func hccsym nodesc pure

27 integer
sptr:639 dtype:6 sc:0=NONE stype:3=constant
```

```
enclfunc:0 hashlk:370=ibset scope:1=NOSYM symlk:0
address:0 conval1g:0 conval2g:27 conval3g:0 conval4g:0
```

Listing 13: Relevant part of Symbol Table of example Fortran program.

As can be seen, Symbol Table contains character strings, numeric constants and runtime library function names.

Listing 14 shows the part of ILM (as generated by *flang1* for the *print* instruction in the example Fortran program) responsible for emitting the "hello world" string into a print buffer. In this example, runtime function *f90io_sc_ch_ldw* (*s636*) is called by the *IUFUNC* instruction. We can observe that the string "hello world" (*s633*) and data type 14 (*s637*; *__STR*, *Fortran character*, see definition in Listing 15) should be passed to this function.

```
i40: ----------------------
i0: BOS l2 n1 n0
i4: FILE n2 n1 n3000
i8: BASE s631
i10: BASE s633
i12: ICON s637
i14: DPVAL i12
i16: FARG i10 t58
i19: FARG i14 t6
i22: IUFUNC n2 s636 i16 t58 i19 t6
i27: IST i8 i22
i30: ----------------------
```

Listing 14: Part of ILM emitting "hello word" into a print buffer as generated for example Fortran program.

```
typedef enum {
...
  __STR = 14,        /*   F character    */
...
  __REAL4 = 27,      /*   F real*4, real */
...
} _DIST_TYPE;
```

Listing 15: Definitions of *Fortran character* and *Fortran real*4* data types in the runtime library.

Similarly, listing 16 shows the part of ILM (as generated by *flang1* for *print* instruction of example Fortran program) responsible for emitting the floating point numeric constant into a print buffer. In this example, the runtime function *f90_io_sc_f_ldw* (*s638*) is called by the *IUFUNC* instruction. We can observe that the encoded numeric constant (*s634*) and data type 27 (*s639*; *__REAL4*, *Fortran real*4*, see definition in Listing 15) should be passed to this function.

```
i30: ----------------------
i0: BOS l2 n1 n0
i4: FILE n2 n1 n4000
i8: BASE s631
i10: RCON s634
i12: DPVAL i10
i14: ICON s639
i16: DPVAL i14
i18: FARG i12 t8
i21: FARG i16 t6
i24: IUFUNC n2 s638 i18 t8 i21 t6
i29: IST i8 i24
i32: ----------------------
```

Listing 16: Part of ILM emitting the floating point numeric constant into a print buffer as generated for example Fortran program.

Listing 17 shows the part of ILI (as generated by the *flang2* for *print* instruction of example Fortran program) responsible for emitting "hello world" and the floating point numeric constant. Two calls to runtime functions (*f90io_sc_ch_ldw* and *f90io_sc_f_ldw*) can be easily noticed. The code preparing arguments for those functions is also similar. Note that the Symbol Table identifiers which previously were integer numbers around 600 now are integer numbers around 300 (the Symbol Table was regenerated during import step of *flang2*). This listing presents a human–readable dump of ILI code (see Section 5 for a guidance on dumping internal data structures in a human–readable form).

```
35    ACON          322~<z__io,0>
36    ST             34^     35^      3~ <z__io>     wd
37    ICON          323~<12>
38    ACON          324~<"hello world ",0>
39    ICON          308~<14>
40    KCON          325~<12>
41    ARGKR          40^      1^
42    ARGIR          39^     41^
43    ARGAR          38^     42^       0
44    GARG           40^      1^       7      1~ <?vol>
45    GARG           39^     44^       6      1~ <?vol>
46    GARG           38^     45^      56      0~ <?>
47    JSR           307~<f90io_sc_ch_ldw>     43^     48^-
      alt
48    GJSR          307~<f90io_sc_ch_ldw>     46^
49    DFRIR          47^ ir( 1)
50    ST             49^     35^      3~ <z__io>     wd
51    FCON          305~< 1.2340000153E+01>
52    ICON          310~<27>
53    ARGIR          52^      1^
54    ARGSP          51^     53^
55    GARG           52^      1^       6      1~ <?vol>
56    GARG           51^     55^       8      1~ <?vol>
57    JSR           309~<f90io_sc_f_ldw>     54^     58^-alt
58    GJSR          309~<f90io_sc_f_ldw>     56^
59    DFRIR          57^ ir( 1)
```

Listing 17: Part of ILI emitting "hello word" and a floating point numeric constant into a print buffer as generated for the example Fortran program.

Finally, Listing 18 shows a part of LLVM IR (without metadata) as generated by *flang2* for the example Fortran program.

```
; ModuleID = 'hello.f90'
target datalayout = "e-m:e-i64:64-i128:128-n32:64-S128"
target triple = "aarch64-unknown-linux-gnu"
define void @MAIN_() !dbg !14 {
L.entry:
    %z__io_303 = alloca i32, align 4

  %0 = bitcast i32* @.C283_MAIN_ to i8*, !dbg !18
  %1 = bitcast void ()* @fort_init to void (i8*)*, !dbg
     !18
  call void %1 (i8* %0), !dbg !18
  br label %L.LB1_316

L.LB1_316:
  %2 = bitcast i32* @.C300_MAIN_ to i8*, !dbg !19
  %3 = bitcast [9 x i8]* @.C298_MAIN_ to i8*, !dbg !19
  %4 = bitcast void ()* @f90io_src_info03 to void (i8*,
     i8*, i64)*, !dbg !19
  call void %4 (i8* %2, i8* %3, i64 9), !dbg !19
  %5 = bitcast i32* @.C301_MAIN_ to i8*, !dbg !19
  %6 = bitcast i32* @.C283_MAIN_ to i8*, !dbg !19
  %7 = bitcast i32* @.C283_MAIN_ to i8*, !dbg !19
  %8 = bitcast i32 ()* @f90io_print_init to i32 (i8*, i8
     *, i8*, i8*)*, !dbg !19
  %9 = call i32 %8 (i8* %5, i8* null, i8* %6, i8* %7), !
     dbg !19
  store i32 %9, i32* %z__io_303, align 4, !dbg !19
```

```
  %10 = bitcast [12 x i8]* @.C304_MAIN_ to i8*, !dbg !19
  %11 = bitcast i32 ()* @f90io_sc_ch_ldw to i32 (i8*, i32
     , i64)*, !dbg !19
  %12 = call i32 %11 (i8* %10, i32 14, i64 12), !dbg !19
  store i32 %12, i32* %z__io_303, align 4, !dbg !19
  %13 = bitcast i32 ()* @f90io_sc_f_ldw to i32 (float,
     i32)*, !dbg !19
  %14 = call i32 %13 (float 0x4028AE1480000000, i32 27),
     !dbg !19
  store i32 %14, i32* %z__io_303, align 4, !dbg !19
  %15 = call i32  @f90io_ldw_end (), !dbg !19
  store i32 %15, i32* %z__io_303, align 4, !dbg !19
  ret void, !dbg !20
}

@.C310_MAIN_ = internal constant i32 27
@.C305_MAIN_ = internal constant float 0x4028AE1480000000
@.C308_MAIN_ = internal constant i32 14
@.C304_MAIN_ = internal constant [12 x i8] [i8 104,i8
     101,i8 108,i8 108,i8 111,i8 32,i8 119,i8 111,i8 114,
     i8 108,i8 100,i8 32]
@.C284_MAIN_ = internal constant i64 0
@.C301_MAIN_ = internal constant i32 6
@.C298_MAIN_ = internal constant [9 x i8] [i8 104,i8 101,
     i8 108,i8 108,i8 111,i8 46,i8 102,i8 57,i8 48]
@.C300_MAIN_ = internal constant i32 2
@.C283_MAIN_ = internal constant i32 0
```

Listing 18: Part of LLVM IR (without metadata) as generated by *flang2* for the example Fortran program.

As we can observe, the global scope of a Fortran program becomes wrapped by a compiler–generated *MAIN_* function.

5 FRONTEND COMMAND LINE OPTIONS (FLANG1 AND FLANG2)

Although both *flang1* and *flang2* executables can be invoked separately without being controlled by frontend driver, no help is provided describing command line arguments accepted by either of them. The best way to find the full list of accepted arguments is to examine the source code.

Listing 19 shows what arguments are registered for *flang1* using the *ArgParser* library API.

```
arg_parser_t *arg_parser;

create_arg_parser(&arg_parser, true);

/* Register two ways for supplying source file argument
   */
register_filename_arg(arg_parser, &sourcefile);
register_string_arg(arg_parser, "src", &sourcefile, NULL
   );
/* Output file (.ilm) */
register_combined_bool_string_arg(arg_parser, "output",
   (bool *)&(flg.output),
                                  &outfile_name);
/* Other files to input or output */
register_string_arg(arg_parser, "stbfile", &stboutfile,
   NULL);
register_string_arg(arg_parser, "modexport", &
   modexport_val, NULL);
register_string_arg(arg_parser, "modindex", &
   modindex_val, NULL);
register_string_arg(arg_parser, "qfile", &dbgfile, NULL)
   ;

/* Optimization level */
register_integer_arg(arg_parser, "opt", &(flg.opt), 1);

/* Allocate space for command line macro definitions */
flg.def = (char **)getitem(8, argc * sizeof(char *));
flg.undef = (char **)getitem(8, argc * sizeof(char *));
flg.idir = (char **)getitem(8, argc * sizeof(char *));
module_dirs = (char **)getitem(8, argc * sizeof(char *))
   ;
/* Command line macro definitions */
register_string_list_arg(arg_parser, "def", flg.def);
register_string_list_arg(arg_parser, "undef", flg.undef)
   ;
register_string_list_arg(arg_parser, "idir", flg.idir);
register_string_list_arg(arg_parser, "moddir",
   module_dirs);

/* x flags */
register_xflag_arg(arg_parser, "x", flg.x,
                   (sizeof(flg.x) / sizeof(flg.x[0])));
register_yflag_arg(arg_parser, "y", flg.x);
/* Debug flags */
register_qflag_arg(arg_parser, "q", flg.dbg,
                   (sizeof(flg.dbg) / sizeof(flg.dbg[0])
   ));
register_action_map_arg(arg_parser, "qq", phase_dump_map
   , dump_map);

/* Other flags */
register_boolean_arg(arg_parser, "mp", (bool *)&(flg.smp
   ), false);
register_boolean_arg(arg_parser, "preprocess", &
   arg_preproc, true);
register_boolean_arg(arg_parser, "reentrant", &
   arg_reentrant, false);
register_integer_arg(arg_parser, "terse", &(flg.terse),
   1);
register_inform_level_arg(arg_parser, "inform",
                          (inform_level_t *)&(flg.inform
   ), LV_Inform);
register_boolean_arg(arg_parser, "hpf", (bool *)&(flg.
   hpf), true);
register_boolean_arg(arg_parser, "static", (bool *)&(flg
   .doprelink), true);
register_boolean_arg(arg_parser, "quad", (bool *)&(flg.
   quad), true);
register_boolean_arg(arg_parser, "freeform", &
   arg_freeform, false);
register_string_arg(arg_parser, "tp", &tp, NULL);
register_string_arg(arg_parser, "stdinc", &(flg.stdinc),
    (char *)1);
register_integer_arg(arg_parser, "vect", &(vect_val), 0)
   ;
register_boolean_arg(arg_parser, "standard", (bool *)&(
   flg.standard), false);
register_boolean_arg(arg_parser, "save", (bool *)&(flg.
   save), false);
register_boolean_arg(arg_parser, "extend", &arg_extend,
   false);
register_boolean_arg(arg_parser, "recursive", (bool *)&(
   flg.recursive),
                     false);
register_string_arg(arg_parser, "cmdline", &(flg.cmdline
   ), NULL);

/* Set values form command line arguments */
parse_arguments(arg_parser, argc, argv);
```

Listing 19: Registration of *flang1* command line arguments.

Closer examination of *flang1* source code reveals the meaning of these arguments:

- *-filename* – source file name (method 1 of 2).
- *-src* – source file name (method 2 of 2).
- *-output* – output *.ilm* text file name (will contain STB + ILM).
- *-stbfile* – output *.stb* symbol table text file name (will contain STB only).
- *-modexport* – output *.cmod* text file name, collective content of all modules exported by given Fortran source file (note that separate *.mod* files will be also generated by *flang1*!).

- *-modindex* – output *.cmdx* text file containing index of exported modules (including offsets in *.cmod* file where each of the modules begins).
- *-qfile* – output debug *.qdbg* text file name.
- *-opt* – optimization level.
- *-def* – same as clang's -D.
- *-undef* – same as clang's -U.
- *-idir* – same as clang's -I.
- *-moddir* – same as flang's -J.
- *-x* – the *-x* parameter (see below).
- *-y* – the *-y* parameter (see below).
- *-q* – the *debug* parameter (see further below).
- *-qq* – the *dump* parameter (see further below).
- *-mp/-nomp* – process OpenMP directives.
- *-preprocess/-nopreprocess* – process C preprocessor directives in Fortran code.
- *-reentrant/-noreentrant* – local variables should have *RECURSIVE* semantics instead of *SAVE* semantics (same as *recursive/norecursive* below); also inhibit special treatment of terminus functions.
- *-terse* – reduce verbosity of error messages (where 0 = do not reduce).
- *-inform* – increase verbosity level of error messages.
- *-hpf/-nohpf* – handle HPF (High Performance Fortran) – off by default, does not seem to be fully implemented!.
- *-static/-nostatic* – enable IPA (inter-procedural analysis), does not seem to do anything, always set to *OFF*.
- *-quad/-noquad* – quad align (round-up size) *unconstrained objects* if size is greater or equal to 16.
- *-freeform/-nofreeform* – allow free form code.
- *-tp* – x86–specific target architecture name, needs to be dropped.
- *-stdinc* – colon–separated list of include directories.
- *-vect* – vectorizer settings (bitmask), does not seem to be examined anywhere in the frontend.
- *-standard/-nostandard* – be strict about standard, e.g. disallow use of <> instead of /= or *.ne.*.
- *-save/-nosave* – local variables should have *SAVE* semantics instead of *RECURSIVE* semantics (typically used for overriding OpenMP's default using *-Msave*).
- *-extend/-noextend* – allow lines up to 132 characters length (instead of 72 characters).
- *-recursive/-norecursive* – local variables should have *RECURSIVE* semantics instead of *SAVE* semantics (OpenMP's default, can be forced by *-Mrecursive*).
- *-cmdline* – override command line used to invoke the compiler.

Listing 20 shows what arguments are registered for *flang2* using the *ArgParser* library API.

```
/* Argument parser */
arg_parser_t *arg_parser;

/* Create argument parser, force errors on unknown flags
    (last argument is
 * true) */
create_arg_parser(&arg_parser, true);

/* Input file name */
register_filename_arg(arg_parser, &sourcefile);
/* Fortran source file */
register_string_arg(arg_parser, "fn", &(gbl.file_name),
    NULL);
/* Other files to input or output */
register_string_arg(arg_parser, "stbfile", &stboutfile,
    NULL);
register_combined_bool_string_arg(arg_parser, "asm", (
    bool *)&(flg.asmcode),
                                  &asmfile);

/* Register version arguments */
register_string_arg(arg_parser, "vh", &(version.host), "
    ");

/* x flags */
register_xflag_arg(arg_parser, "x", flg.x,
                   (sizeof(flg.x) / sizeof(flg.x[0])));
register_yflag_arg(arg_parser, "y", flg.x);
/* Debug flags */
register_qflag_arg(arg_parser, "q", flg.dbg,
                   (sizeof(flg.dbg) / sizeof(flg.dbg[0])
    ));
register_action_map_arg(arg_parser, "qq", phase_dump_map
    , dump_map);

/* Other flags */
register_integer_arg(arg_parser, "opt", &(flg.opt), 1);
register_integer_arg(arg_parser, "ieee", &(flg.ieee), 0)
    ;
register_inform_level_arg(arg_parser, "inform",
                          (inform_level_t *)&(flg.inform
    ), LV_Inform);
register_integer_arg(arg_parser, "endian", &(flg.endian)
    , 0);
register_boolean_arg(arg_parser, "mp", (bool *)&(flg.smp
    ), false);
register_boolean_arg(arg_parser, "reentrant", &
    arg_reentrant, false);
register_integer_arg(arg_parser, "terse", &(flg.terse),
    1);
register_boolean_arg(arg_parser, "quad", (bool *)&(flg.
    quad), false);
register_boolean_arg(arg_parser, "save", (bool *)&(flg.
    save), false);
register_string_arg(arg_parser, "tp", &tp, NULL);
register_integer_arg(arg_parser, "astype", &(flg.astype)
    , 0);
register_boolean_arg(arg_parser, "recursive", (bool *)&(
    flg.recursive),
                     false);
register_integer_arg(arg_parser, "vect", &(vect_val), 0)
    ;
register_string_arg(arg_parser, "cmdline", &(cmdline),
    NULL);
register_boolean_arg(arg_parser, "debug", (bool *)&(flg.
    debug), false);

/* Run argument parser */
parse_arguments(arg_parser, argc, argv);
```

Listing 20: Registration of *flang2* command line arguments.

Closer examination of *flang2* source code reveals the meaning of these arguments:

- *-filename* – source *.ilm* file name.
- *-fn* - (the original) Fortran source file name.
- *-stbfile* – input *.stb* symbol table file name.
- *-asm* – output *.ll* LLVM IR text file name.
- *-vh* – build host; does not seem to have any use currently.
- *-x* – the *-x* parameter (see below).
- *-y* – the *-y* parameter (see below).
- *-q* – the *debug* parameter (see further below).
- *-qq* – the *dump* parameter (see further below).

- *-opt* – optimization level.
- *-ieee* – enable or disable IEEE division.
- *-inform* – increase verbosity level of error messages.
- *-endian* – force endianness, where 0 = little, 1 = big.
- *-mp/-nomp* – process OpenMP pragmas.
- *-reentrant/-noreentrant* – local variables should have *RECURSIVE* semantics instead of *SAVE* semantics (same as *recursive/norecursive* below); also inhibit special treatment of terminus functions.
- *-terse* – reduce verbosity of error messages (where 0 = do not reduce).
- *-quad/-noquad* – quad align (round–up size) *unconstrained objects* if size is greater or equal to 16.
- *-save/-nosave* – local variables should have *SAVE* semantics instead of *RECURSIVE* semantics (typically used for overriding OpenMP's default using *-Msave*).
- *-tp* – x86–specific target architecture name, needs to be dropped.
- *-astype* – assembler syntax, where 0 = ELF, 1 = COFF.
- *-recursive/-norecursive* – local variables should have *RECURSIVE* semantics instead of *SAVE* semantics (OpenMP's default, can be forced by *-Mrecursive*).
- *-vect* – vectorizer settings (bitmask), does not seem to be examined anywhere in the frontend.
- *-cmdline* – override command line used to invoke the compiler.
- *-debug/-nodebug* – debug *ON*/*OFF* flag.

5.1 -x and -y arguments

These flags (also called *xflags*) were added to allow more detailed control (than flags listed above) of the compiler's behavior. The *-y* flag is the inversion of *-x* flag (whenever given *-x* flag instructs to do some action, the *-y* flag with the same arguments instructs not to do it). Multiple *-x* and *-y* arguments can be passed to *flang1* or *flang2* during their invocation. In order to ease the use of this flag in code, the *XBIT* macro was defined as shown in Listing 21. Example of use can be observed in Listing 22.

General usage of these flags when invoking *flang1* or *flang2* directly is: *-x <number ><value >* or -y <number ><value >. Numbers and corresponding values are described in documentation that could be generated from file *xflag.n*. For example, number 124 and value 0x10 for which the *if* condition is checking in Listing 22 is described as follows:

> 124: F77 implementation-defined behavior modifications.
> 0x10: treat INTEGER as INTEGER*8 and LOGICAL as LOGICAL*8

```
#define XBIT(n, m) (flg.x[n] & m)
```

Listing 21: *XBIT* macro definition.

```
if (XBIT(124, 0x10)) {
  dt_int = DT_INT8; /* -i8 */
} else {
  dt_int = DT_INT;
}
```

Listing 22: *XBIT* macro in use.

Any of these *-x* and *-y* flags can be passed to *flang1* and *flang2* from flang command line options: *-Hx,<number >,<value >* passes *-x <number ><value >* to *flang1*, *-Mx,<number >,<value >* passes *-x <number ><value >* to *flang2* (e.g. *-Mx,183,0x20000*). The same is valid for *-y*. Multiple *-x*, *-y* (and *-Hx*, *-Hy*, *-Mx*, *-My* for the compiler invocation) arguments can be specified within one command line.

Most of the *xflags* values are bit vectors, except those that should hold actual (plain) values (e.g. *xflags* 9 and 10). Listing 23 reveals how these two classes are distinguished.

```
/** Does this index correspond to a bitvector (true) or
    plain value (false) */
bool
is_xflag_bitvector(int index)
{
  switch (index) {
  case 9:   /* max cnt for unroller */
  case 10:  /* # of times to unroll */
  case 16:  /* lower bound loop iter count for vectorizer
     */
  case 27:  /* default/max overlap size */
  case 30:  /* lower limit on iteration count */
  case 31:  /* lower limit on strip size */
  case 32:  /* cache size */
  case 33:  /* upper limit on strip size in complex-
    vector loops */
  case 35:  /* max iteration count */
  case 38:  /* MAXVPUS */
  case 40:  /* # array loads/stores */
  case 41:  /* # fp operations */
  case 43:  /* # CPUs */
  case 44:  /* min parallel loop count */
  case 64:  /* code straightening optimizations */
  case 79:  /* CSE of DP loads: distance threshold */
  case 100: /* break blocks */
  case 101: /* ST processor SI information */
  case 105: /* upper limit on unrolling with unroll & jam
     */
  case 106: /* set unroll factor for scalar  unroll & jam
     */
  case 130: /* Levels of VLIW scheduling */
  case 131: /* Levels of predication */
  case 133: /* VLIW density */
  case 134: /* register stall limit */
  case 138: /* Limit of the #of prefetches for a loop */
  case 139: /* Limit of #vili for vectorizing single
    precision loop */
  case 140: /* Limit of #vili for vectorizing double
    precision loop */
  case 157: /* count for unroller for multiblock loops */
  case 160: /* nest level for intensity count */
  case 181: /* 3D tile size */
  case 188: /* openacc default vector length */
  case 199: /* max blocks in loop to be fused */
  case 249: /* LLVM version number */
    return false;
  default:
    break;
  }
  return true;
}
```

Listing 23: *xflags*: bit vectors versus plain values.

5.2 -q and -qq arguments

The *-q* flag configures compile–time debugging behavior. Multiple *-q* (and *-Hq* and/or *-Mq* can be specified for the compiler invocation, similar to *-x* and *-y* shown above) parameters can be specified within one command line.

They share the same syntax as with *-x* parameters: *-q <number ><value >*, for example: *-q 0 1* is a very useful setting, which in documentation generated from *coding.n* file is described as follows:

> 0:
> 1 Use stderr for debug file (instead of ".qdbg").

The *-q* parameter not only allows tracing of frontend activity (due to verbose debug output) but also allows to display contents of important data structures. For example, *-Mq,0,1 -Mq,4,1* causes *flang2* to print human–readable ILM representation right after it is imported from *flang1*'s output file. In order to print human–readable ILI representation, one can type: *-Mq,0,1 -Mq,10,2 -Mq,10,1*.

Additionally to the above, *-qq* parameter causes debug routines to dump human-readable views of various internal data structures. The syntax is: *-qq <phase_name ><object_name >*. Listing 24 presents how phase names and various object names are defined. For example, in order to dump *flang1*'s *AST*, one can pass *-Hq,0,1 -Hqq,parser,ast* to the compiler. In order to dump *flang1*'s *symtab*, one can pass *-Hq,0,1 -Hqq,parser,symtab* to the compiler.

```
static char *who[] = {"init",     "parser",   "bblock", "
    vectorize", "optimize",
                  "schedule", "assemble", "xref",   "
    unroll"};

  /* Create a datastructure of various dump actions and
     their names */
  action_map_t *dump_map; /* Deallocated after arguments
     are parsed */
  create_action_map(&dump_map);
  add_action(dump_map, "ast", dump_ast);
  add_action(dump_map, "dtype", dumpdts);
  add_action(dump_map, "std", dump_stds);
  add_action(dump_map, "sstd", dump_sstds);
  add_action(dump_map, "stdp", dump_stdps);
  add_action(dump_map, "sym", dump_symbols);
  add_action(dump_map, "syms", dump_symbols);
  add_action(dump_map, "symtab", dump_symbols);
  add_action(dump_map, "allsym", dump_all_symbols);
  add_action(dump_map, "stats", dump_stg_stats);
  add_action(dump_map, "area", report_area);
  add_action(dump_map, "olddtype", dmp_dtype);
  add_action(dump_map, "odtype", dmp_dtype);
  add_action(dump_map, "oldsym", dump_old_symbols);
  add_action(dump_map, "osym", dump_current_symbols);
  add_action(dump_map, "hsym", dump_current_symbols);
  add_action(dump_map, "hsyms", dump_current_symbols);
  add_action(dump_map, "common", dcommons);
  add_action(dump_map, "commons", dcommons);
  add_action(dump_map, "nast", dumpasts);
  add_action(dump_map, "stdtree", dumpstdtrees);
  add_action(dump_map, "stdtrees", dumpstdtrees);
  add_action(dump_map, "shape", dumpshapes);
  add_action(dump_map, "aux", dumplists);
  /* Initialize the map that will be used by dump handler
     later */
  create_action_map(&phase_dump_map);
```

Listing 24: phase names and object names that can be passed with *-qq* parameter.

6 CONCLUSIONS

This article concludes as follows:

- The Flang compiler project has successfully introduced a Fortran compiler frontend that utilizes LLVM compilers infrastructure.
- Now that it is publicly available, the next logical step would be to work towards having it accepted as a part of the LLVM project on equal footing with the Clang compiler.
- Being a part of the LLVM ecosystem and using its backend, Flang can target multiple hardware architectures. Nowadays the support is present for AArch64, 64-bit Power and x86_64 architectures, but it should be fairly easy to add any other 64-bit hardware architecture to this list.
- With its source code based on the commercial PGI® Fortran compiler, Flang inherited most of the interior concepts: two–pass parser, semantic analyzer, two levels of internal representation (ILM, ILI) and internal optimizer.
- The source code is written mostly in ANSI C98 with some utilities written in C++ and some of runtime library routines written in Fortran.
- Documentation can be generated from provided .n files (*nroff* format).
- Some may criticize that Flang source code still contains too many references to its commercial predecessor. We must keep in mind that this can change in time and is completely outweighed by consequences of having the code base developed for more than two decades – such heritage offers maturity and good Fortran standards conformance [1].
- The generated LLVM IR is outputted into a plain text file. All text operations are performed literally, no LLVM API is used for this process.
- Performance comparison of workloads compiled by Flang and GFortran show that although Flang is doing its job well, there is still work to do in this area. Flang performs most of its optimizations within LLVM passes and this arrangement (Fortran frontend combined with LLVM middleend and backend) is still very young and there is a room for further improvement.

REFERENCES

[1] Jason Blevins. 2017. Fortran 2003 status. (3 Mar 2017). http://fortranwiki.org/fortran/show/Fortran+2003+status
[2] Walt Brainerd. 2003. The Importance Of Fortran In The 21st Century. *Journal of Modern Applied Statistical Methods* 2, Article 3 (2003). Issue 1.
[3] LLVM Contributors. 2017. C Language Family Front-end (fork). (2017). https://github.com/flang-compiler/clang
[4] Hal Finkel. 2013. A Fortran language frontend for LLVM. (2013). https://github.com/hfinkel/lfort
[5] Kitware Inc. Ongoing project. CMake. (Ongoing project). https://cmake.org
[6] Don Johnston. 2015. NNSA, national labs team with Nvidia to develop open–source Fortran compiler technology. (13 Nov 2015). https://www.llnl.gov/news/nnsa-national-labs-team-nvidia-develop-open-source-fortran-compiler-technology
[7] Michael Larabel. 2017. FLANG: NVIDIA Brings Fortran To LLVM. (18 May 2017). https://www.phoronix.com/scan.php?page=news_item&px=LLVM-NVIDIA-Fortran-Flang
[8] Eugene Loh. 2010. The Ideal HPC Programming Language. Maybe it's Fortran. Or maybe it just doesn't matter. *ACM Queue* 8 (2010). Issue 6.
[9] Aleksei Lorenz. 2013. End of GSoC report. (23 Sep 2013). http://flang-gsoc.blogspot.co.uk
[10] Jussi Pakkanen. 2013. Why you should consider using separate build directories. (16 Apr 2013). http://voices.canonical.com/jussi.pakkanen/2013/04/16/why-you-should-consider-using-separate-build-directories
[11] PGI. 2017. (2017). https://github.com/flang-compiler/flang
[12] PGI. 2017. flang–compiler. (2017). https://github.com/flang-compiler
[13] PGI. 2017. flang-compiler on Slack. (2017). https://flang-compiler.slack.com/messages
[14] PGI. 2017. Join flang-compiler on Slack. (2017). https://join.slack.com/t/flang-compiler/shared_invite/MjExOTEyMzQ3MjIxLTE0OTk4NzQyNzUtODQzZWEyMjkwYw

[15] Louis-Noël Pouchet, Mohanish Narayan, and Uday Bondugula. 2012. PolyBench/-Fortran, the Polyhedral Benchmark suite. (2012). http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/polybench-fortran.html
[16] James Price. 2017. [llvm–dev] LLVM Fortran front–end. (18 May 2017). http://lists.llvm.org/pipermail/llvm-dev/2017-May/113131.html
[17] Fortran UK. 2015. The Polyhedron Fortran Benchmarks Suite. (2015). https://www.fortran.uk/fortran-compiler-comparisons/the-polyhedron-solutions-benchmark-suite
[18] Bill Wendling. 2012. Fortran Front–End. (2012). https://github.com/isanbard/flang