Apex LLVM Readme

This document was generated on 13-August-2018.

Build #530, based on clang 4.0.1

See Build version for individual components.

Introduction

LLVM (Low-Level Virtual Machine) is a framework for developing compilers for various CPU targets. CLANG is the front-end for the C language and also the name of the main executable.

This document is intended both as a quick-start guide for using the compiler and also as a list of APEX specific features/issues.

Online Documentation

Please refer to opensource community documentation referenced below for features that are common across LLVM and clang framework:

- CLANG: http://releases.llvm.org/4.0.0/tools/clang/docs/
- LD-NEW: /doc/ld-new.txt. LD-NEW linker is based on gnu binutils gold linker which aims to be a direct replacement for the original binutils ld. Online gold linker documentation currently directs to the original ld documentation: https://sourceware.org/binutils/docs-2.27/ld/index.html
- LLVM-AR: https://llvm.org/docs/CommandGuide/llvm-ar.html

The help for command line options for all the tools in the release is also available under /doc directory. Below information from this point on are specific to the APEX target.

Bug report

Bugs should be reported to:

https://jira.sw.nxp.com/secure/RapidBoard.jspa?projectKey=CMPAPEX

Compiler

The main part of the compiler is the `clang` executable located in the `bin` directory. This executable can be used to generate both assembly and object files.

Compiler optimizations are controlled with the `-O<n>` options.

- `-O0` will disable optimizations
- `-O3` will enable all optimizations, for speed
- `-O1` and `-O2` are in between
- `-Os` prevents optimizations that can greatly increase the code size
- `-Oz` is used to optimize for code size

A special option for apex, `-no-apex-support` prevents the `apex/apex-support.h` header from being included automatically. Normally, you will not need to use this option.

The backend options are passed to llc. If passed from clang, each option must be prefixed by `-mllvm`.

`-disable-apex-delay-filler` Disable the filling of delay slots with useful instructions. Noops will be used instead. This option **is** implicitly activated on `-O0`.

`-disable-apex-hl` Disable the hardware loops generation for APEX. Normal branch instructions will be used (j/bnez/beqz). This option **is** implicitly activated on `-O0`.

`-disable-apex-packetizer2` Disable the instruction packetizer for APEX. This might aid in debugging. The packetizer is disabled by default at `-O0`.

The APEX compiler accepts the following target specific attributes:

`__attribute__((loop_free))` Marks the functions as not having any hardware loops. The effect is that hardware loops **may** be performed across calls to this function. If a function does contain loops, but is marked with this attribute, the loops are not converted to hardware loops.

`__attribute__((apex_min_loop_count(n)))` Provides hint of minimum loop iteration count to the loop optimizer. If loop count is non-zero, this allows the optimizer to remove the initial loop counter check which check if the loop should be skipped. This attribute can be used before or after `for` and `while` statement.

`__attribute__((apex_hw_reg("reg")))` Provides hint for register allocation to assign specific physical register to object. Parameter is a case-insenstive string of any allocatable physical register: r1-r28, v0-v7 For objects of vec32 type, a register pair can be specified e.g. __attribute__((apex_hw_reg("v56")))

The following table should be helpful in porting Synopsys extensions to the **clang** equivalent.

Synopsys	**clang** equivalent
property(loop_free)	`__attribute__((loop_free))`
property(never_returns)	`__attribute__((noreturn))`
property(functional)	`__declspec(noalias)` Requires -fms-compatibility option
chess_storage(%16)	`__attribute__((aligned(16)))` For vector objects, the alignment parameter must be specified in vec-byte unit. The default natural alignment for vec08,vec16,vec32 are 1,2,4 vec-byte respectively.
chess_loop_range(...)	`__attribute__((apex_min_loop_count(n)))`
chess_unroll_loop(x)	`#pragma unroll(x)`
chess_frequent_then	`__builtin_expect(x, 1)`
chess_frequent_else	`__builtin_expect(x, 0)`

Predefined macros

The compiler provides __**APEX**__, __**NXP**__, __**LLVM_APEX**__ when targetting Apex. __**LLVM_APEX_VERSION**__ contains the tool's build number.
A global unsigned int __**vecsize** is used to identify the current number of CU controlled by each ACP. The default is 32. User can query at runtime using the `apuGetNumberOfCUs()` API which will updates __**vecsize** as it is configured.

ISO C/C++ standard compliance

The default language standard used for compiling C and C++ sources are gnu11 and gnucxx98 respectively. This mean GNU language extension are turned on by default. In particular, the following non-standard C library functions, which are part of POSIX APIs, are treated as library builtins by the frontend:

- `alloca`
- `stpcpy`, `stpncpy`

- `strdup`, `strndup`
- `index`, `rindex`
- `bzero`
- `strcasecmp`, `strncasecmp`
- `_exit`
- `_longjmp`, `siglongjmp`
- `strlcpy`, `strlcat`

User can specify other language standard using `-std` option. If strict ANSI C compliance is desired, `-ansi` or `-pedantic` option should be used.
For other language extensions support by CLANG, please refer to: http://releases.llvm.org/4.0.0/tools/clang/docs/LanguageExtensions.html

Linker

The **gold** linker is provided and can be found in the `bin` directory under the name `ld-new` (`ld-new.exe` for Windows builds). Please see example below for linking with **gold** using the provided lcf. The option --skip-shdrs will discard all section headers except debug sections header from the final executable which is required to load with the Synopsys simulator. Please link with libhosted.a in order to use hosted IO functionality. Currently simulator built with Synopsys checker version 15.2.29 or newer is required in order for the simulator to parse the hosted IO info properly.

If the linker is invoked manually (i.e. not through clang) it will require some options.

- `-L<LIB_DIR>` is the location where the library archives (the `lib-apex` directory) are located.
- `-l<lib>` must be given for each required library.
- `-T <SCRIPT_FILE>` specifies the linker script file, which describes the layout.
- `--skip-shdrs` must be given in order for the generated file to be parsed by the simulator.

The libraries specified with the `-l` options are the files located in `lib-apex`, without the `lib` prefix and the `.a` suffix. For example, `libc.a` is linked by using `-lc`.

Libraries link order Similar to gnu ld, the **gold** linker by default do a single pass through the list of input libraries to resolve symbols for efficient memory usage, this implies that the input library order is important and listing the same library multiple times might be needed to resolve cyclic dependencies. For simplicity, it is recommended to use the --start-group and --end-group options so the linker would do multiple pass over the group for all unresolved symbols before issuing undefined symbol error.

```
$ # link multiple objects
$ ld-new -L"/opt/apex-cpl/lib-apex/" \
>        -T "/opt/apex-cpl/support/APU2.lcf" \
>        -o output a.o b.o c.o --start-group \
>        -lc -lm -lrt -lhosted -lstartc --end-group \
>        --skip-shdrs
```

The `bridge` linker from Synopsys can also be used. See Examples for typical bridge link command. Please refer to Synopsys documentation for detailed linker usage.

Libraries

The compiler comes with the **Embedded Warrior Library 2** compiled for apex.

The default include directory for the headers is `"../include/ewl2"`, relative to the `clang` executable. In order to override this, you can use the EWL2_INCLUDE environment variable.

```
$ # override the default header location
$ export EWL2_INCLUDE="/opt/apex-cpl/include/ewl2"
```

The apex library archives are located in the `lib-apex` directory. They are `libc.a`, `libc99.a`, `libm.a`, `librt.a`, `libhosted.a`, `libhosted_syn.a`, `libstartc.a` and `libstartcpp.a`

- libc.a : Size-optimized C Library
- libc99.a : Full C99 compliant C Library
- libm.a : Math library from Sun implementation
- librt.a : Runtime library includes compiler runtime support
- libhosted.a : Hosted IO library to be used with gold linker
- libhosted_syn.a : Hosted IO library to be used with Synopsys bridge linker
- libstartc.a : Library that contains the startup routines for the C language
- libstartcpp.a : Library that contains the startup routines for the CPP language

These libraries are created using `llvm-ar`.

Library for Instrumentation The apex library archives includes libraries to enable instrumentation like code coverage and profiling. They are `libprofile.a`, `libstart_lprof.a` and `libstart_gcov.a`

- libprofile.a : Library that containe routines that aid instrumentation
- libstart_lprof.a : Library that contains the startup routines for LLVM's source-based code profiling support
- libstart_gcov.a : Library that contains the startup routines for LLVM's GCC-compatible source based coverage support

These libraries are created using `llvm-ar`.

Vector Types and Intrinsics User-level APEX vector intrinsics in available in `"include/ewl2/intrinsics.hpp"`. Basic vector types are also defined in `"include/ewl2/vector-types.h"`. This header is already included by `"include/ewl2/intrinsics.hpp"`. You **do not** need to include these files under normal circumstances. You may need to include one or more of them if you compile with the `-no-apex-support` option.

Support Files The `support/` folder provides the following files to support building an application:

- S32V_APEX.lcf : gnu linker script when building for S32V board
- APU2.lcf : gnu linker script to be used with `gold` linker
- APU2.bcf : linker script to be used with Synopsys `bridge` linker
- startup.s : default startup code which implements the `__main_fsl` entry function and `__exit_fsl` exit function
- startup_cpp.s : the default startup code, adjusted for C++
- startup_lprof.s : default startup code which implements the initializations for LLVM's source-based code profiling support
- startup_gcov.s : default startup code which implements the initializations for LLVM's GCC-compatible source based coverage support

Assembler

A standalone assembler is available in "bin/llvm-mc". It supports GNU assembly syntax. Alternatively, clang can also be used to assemble a file. To assemble a .s file:

```
$ llvm-mc -filetype=obj main.s -o main.o
$ clang main.s -o main.o
```

When using inline asm, the following constraints are supported:

- "r", Rn register, used for scalars and pointers
- "v", Vn register, used for vector types, except vbool
- "b", VCn register, used for vector conditions

APEX specific assembler directive:

- .vliw_start
- .vliw_end

The assembler will try to encode the instructions between the **.vliw_start** and **.vliw_end** directive into an 32-bit or 64-bit instruction bundle. The instruction between these directives are executed in the same cycle. If no valid encoding is found, it will return an error The instructions must be in order and nops must be included if not all functional units are used.

```
.vliw_start
  or r1, r0, r0
  vadd v0, v0, v7
  vsll v1, v6, v7
.vliw_end
```

Disassembler

There are two alternatives for the disassembler. The first one is "bin/llvm-objdump", which is similar to GNU objdump. This can disassemble an elf file:

```
$ cat a.c
int main() {
  return 0;
}
$ clang a.c -O3 -o a.o -c
$ llvm-objdump -disassemble -triple=apex a.o
a.o:    file format ELF32-unknown
Disassembly of section .text:
main:
       0:       17 d0 00 00                                     jr r29
       4:       00 00 00 00                                     nop
       8:       00 10 01 0d                                     or r1, r0, r0
```

The second option is to use "bin/llvm-mc":

```
$ echo "0 0 0 0" | llvm-mc -disassemble
     .text
     nop
```

```
$ echo "0x17 0xd0 0x0 0x0" | llvm-mc -disassemble
      .text
      jr r29
$ echo "[0x00,0x10,0x01,0x0d]" | llvm-mc -disassemble -show-encoding
      .text
      or r1, r0, r0                   ; encoding: [0x00,0x10,0x01,0x0d]
```

Diassembly of a bundled instructions will appear between .vliw_start and .vliw_end directive.

ABI and programming model compatibility

The LLVM compiler implements the ABI and programming model specified in the following documents

- HDD-10294-15-01_APEX2_Application_Binary_Interface
- UG-10310-00-05-APU-2_C_Programmer_Guide

with exceptions as follow

- Explicit cast is required for vector operations between different types and/or signedness.
- When casting vector to a bigger storage, sign extension is based on the type of the source vector.
- Scalar operations from a bigger type to a smaller type results in an implicit truncate operation. To enable warning this for, use the -Wconversion compile option
- Vector control flow language extension: currently vector predication in for and while loop condition is not supported.

Instrumentation

There are two alternatives for generating instrumentation. The first one is LLVM's source-based code profiling support.

In order to enable this compile with the option `"-fprofile-instr-generate -fcoverage-mapping"` Two types of elf needs to be generated for this method. This is because for Synopsys simulator section headers should be stripped while for llvm profiling tools section headers should be generated. This is contolled by `"--skip-shdrs"` option to the linker.

```
$ cat main.c
int main () {
  return 0;
}
$ clang main.c -fprofile-instr-generate -fcoverage-mapping -o main.out
$ # Run the instrumented elf generated without section headers to generate raw profile data
$ simulator main.out
$ # Generate elf with section headers
$ # Use -v option from 1st step and remove --skip-shdrs from the options
$ ld-new.exe -o main.elf -T path_to_APU2.lcf \
>        --start-group main.o -L path_to_(bin/../lib-apex/) \
>        -lc -lm -lrt -lhosted -lprofile -lstart_lprof \
>        --end-group --gc-sections
```

```
$ # Combine multiple raw profiles and index them
$ llvm-profdata.exe merge -o test.profdata default.profraw
$ # Generate a line-oriented coverage report
$ llvm-cov.exe show main.elf -instr-profile=test.profdata main.c
    1|      1|int main () {
```

```
    2|      1|  return 0;
    3|      1|}
```

```
$ # Generate a file-level summary of coverage
$ llvm-cov.exe report main.elf -instr-profile=test.profdata main.c
File 'main.c':
Name                         Regions    Miss   Cover     Lines    Miss   Cover
------------------------------------------------------------------------------
main                               1       0 100.00%         3       0 100.00%
------------------------------------------------------------------------------
TOTAL                              1       0 100.00%         3       0 100.00%
```

Please note that even-though this method is primarily used for Profile Guided Optimisation (PGO), PGO is not tested on APEX. Since APEX is a bare metal target without any OS support some options associated with this method is not available. Specifying the path to wite the raw profile data via `"LLVM_PROFILE_FILE"` environment variable, specifying the name of the raw profile data using pattern strings are examples of the un-supported options.

The second method is LLVM's GCC-compatible source based coverage support. For this method build an instrumented version of the application with `"-fprofile-arcs -ftest-coverage"` option. (Alternatively, `"--coverage"` option can be used, which includes both of those other options.) Application should be compiled with debugging information `"-g"` and without optimization `"-O0"`". Otherwise, the coverage data cannot be accurately mapped back to the source code.

```
$ cat main.c
int main () {
  return 0;
}
$ clang main.c --coverage -g -o main.out
$ # Run the instrumented elf
$ simulator main.out
$ llvm-cov.exe gcov main.elf main.c
File 'main.c'
Lines executed:100.00% of 1
main.c:creating 'main.c.gcov'

File 'main.c'
Lines executed:100.00% of 1
main.c:creating 'main.c.gcov'
$ cat main.c.gcov
        -:    0:Source:main.c
        -:    0:Graph:main.gcno
        -:    0:Data:main.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:int main () {
        1:    2:  return 0;
        -:    3:}
```

License Check

The compiler will check for valid Flex license through the use of Flexera plugins. These shared libraries are located in `"/lib"` for Linux and `"/bin"` for Windows build. On Linux, please add the lib path to LD_LIBRARY_PATH so the loader could locate these libraries sucessfully.

Examples

Clang examples

```
$ # Single C file to elf
$ clang -Os -c file.c -o file.o
$ # Single C file to assembly
$ clang -Os -c file.c -S -o file.s
$ # Multiple C files to elf
$ clang -Os -c file1.c file2.c file3.c # creates file1.o file2.o file3.o
$ # Make an executable file
$ clang -Os -o elf file.c
```

```
$ # link using clang
$ clang -o main.elf main.o
$ # link using gold linker
$ ld-new -o main.elf main.o -L ${EWL2_LIBS} \
>   -T ${EWL2_LIBS}/../support/APU2.lcf \
>   --start-group -lc -lm -lrt -lhosted -lstartc --end-group \
>   --skip-shdrs
```

```
$ # link using synopsys linker
$ bridge -B -g \
>  -I${synopsys_install}/designs/APU2/lib/base \
>  -I${synopsys_install}/designs/APU2/lib/base/isg \
>  -I${synopsys_install}/designs/APU2/inc \
>  -I${synopsys_install}/designs/APU2/iss/inc \
>  -I${synopsys_install}/designs/APU2/lib/ \
>  -c${apex_llvm_install}/support/APU2.bcf \
>  -DAPU_VSIZE=32 -D__tct_patch__=09 \
>  -L${apex_llvm_install}/lib -lc -lm -lrt -lhosted_syn -lstartc -pAPU2 \
>  main.o -o main.elf
```

Tips and tricks

Delay slots

When using assembly files as input (not generated by the compiler), you should be careful of instructions with delay slots. Instructions may have 0, 1 or 2 delay slots. An instruction with 2 delay slots will also execute the following 2 instructions before jumping to the destination.

The symbol at which an instruction jumps may be affected by the size of the instructions in the delay slots. The following restrictions apply.

Conditional branch. There are 2 conditional branch instructions: `BNEZ` and `BEQZ`. Both have 2 delay slots. These instructions will have an addend of 0 when the instruction in the first delay slot is a 32-bit instruction. If the instruction in the first delay slot is a 64-bit instruction, the addend should be -4 in order to compensate for the extra 32-bits. The instruction in the second delay slot does not affect these branch instructions.

```
f:                                      ; @f
     neg r1, r7
$BB0_1:
     addi r1, r1, #1
```

```
     sne r7, r1, r0
     bnez r7, #($BB0_1)-4
     xor r6 : r5, r6 : r5, r0 : r0   ; delay slot 1
     xor r4 : r3, r4 : r3, r6 : r5   ; delay slot 2
; BB#2:
     jr r29
     or r1, r3, r3                   ; delay slot 1
     or r2, r4, r4                   ; delay slot 2
```

Hardware loops. There are 2 hardware loops instructions: `DO` and `DOI`, both with 2 delay slots. The symbol operand of the loop instructions represent the loop end (i.e. the block that will be executed after the loop finishes). The hardware loop consists of all the instructions between the `DO`/`DOI` instruction (excluding the delay slots) and the loop end label. If the instruction in the first delay slot is a 64-bit instruction, the addend should be -4 in order to compensate for the extra 32bits. Similar to the branch instructions, the second delay slot does not affect the addend. If you do not want to perform this optimization on a particular function, you can use the "loop_free" attribute.

Debug information. Using -g option will enable debug info generation. By default dwarf2 debug format is generated. This debug info is compatible with Lauterbach debugger release after 09/01/2016.

Vector casts Casts between vectors of different integer sizes work, but they must be explicit.

```
vec16s v16a = (vec16s)v8; // C-style cast -> works
vec16s v16b = vec16s(v8); // C++-style cast -> works
```

LLVM auto-vectorizer Auto-vectorization of scalar loop requires movement of data to and from CMEM/DMEM which is not yet supported. To avoid this problem, auto vectorization has been disabled by default currently. To enable explicitly, please add the compile options `-fvectorize -fslp-vectorize`.

Flush on exit In order to dump all the IO, a call to `fflush(stdout)` is added after the `main` function is called. This call is also reached from the `exit` function. The `startup.s` file contains this code and can be modified & linked with the application to override the default startup. To exclude the startup library from the default libraries list, you can pass the `-nostartfiles` option to `clang`.

Known issues

Time and Clock support The standard library time and clock functionality and types are not supported under Apex. APEX kernel timing is done from the ARM host immediately before and after APEX device invocation.

C99 variable length array Use of C99 feature variable length array not currently supported

C99 complex type Use of C99 complex types and library functions are not currently supported

Instrumentation Profiling and coverage support does not work in Linux environment due to a bug/limitation of the Synopsys simulator. Till version `"L-2016.09-2"` hosted file i/o support is limited becasue of which library calls like `"fopen"` fails in Linux enviornment. No workaround is know as of now.

Build version

Component	Commit id
LLVM	`d01f2fb216885399adbe219ede961700280e2a15`
CLANG	`30c6f6cb565a55b4e3ef57385465e6534f8b4ace`
GOLD	`87dae28a94a7df20cd011e6fd651bbbb6e694073`

EWL2	c6ccb5062485fe5acc14fda645dfde7f0649bbbf
Compiler-rt	c0618678f072522725fb6ccbdc377d47626623fc
LLVM_tools	e852e1a660763c9f42994da9511dda87348b0203