ARC® GCC
GNU Compiler Collection

Getting Started

ARC® GCC Getting Started

Virage Logic Corporation
47100 Bayside Parkway
Fremont, California 94538, USA
Tel: +1-510-360-8000
Toll-free: 877-360-6690

www.viragelogic.com

5975-004 March 2010

Contents

Chapter 1 — Overview

The ARC® GNU toolchain is a complete compiler toolchain for ARC 600 and ARC 700 processors.

The following sections provide further details on building and using the toolchain:

- The Installation section outlines the build and installation requirements.
- The Tutorial section outlines how to compile and execute a simple HelloWorld application.

Installation

The ARC GCC toolchain is delivered in a source package, so that you can build on your specific host platform (Linux or Cygwin™ for Windows)

To build the ARC GCC toolchain from source, the following are required.

- A native GCC for the Host Operating system version 3.4 or above (for example, GCC x86 for Linux)
- Make version 3.80 or greater
- Texinfo version 3.8
- bison version 1.875 or greater
- byacc
- Building Insight (the Graphical Interface of GDB) requires following additional packages.
- qt (or qt-devel)
- libtermcap/termcap (or libtermcap-devel)
- ncurses (or libtermcap-devel)

A set of shell scripts is provided to automate the build process from source. To build the ARC GCC toolchain (ELF32 version) you can use the script `build_elf32.sh` as follows:

```
# cd $UNZIP_LOC
# ./build_elf32.sh $INSTALL_DIR
```

Where *$INSTALL_DIR* is the directory where toolkit is to be installed.

You can also use the script `build_uclibc.sh` to build the ARC GCC toolchain (UCLIBC version) as follows:

```
# cd $UNZIP_LOC
# ./build_uclibc.sh    $INSTALL_DIR    $LINUX_DIR
```

Where

- *$INSTALL_DIR* is the directory where toolkit will be installed.

- *$LINUX_DIR* is the root directory of ARC Linux sources.

Important Notes

- ARC Linux sources are not the part of compiler distribution. You must obtain the ARC Linux distribution separately (available from ARC).
- ARC Linux sources in this directory must be built with ELF32 version of ARC GCC toolkit. Otherwise, error messages about missing files occur.

When the build is complete you may like to add toolkit path, that is, `$INSTALL_DIR/bin` to environment variable `$PATH`.

Tutorial

After installing the ARC GCC toolkit (or building from source), try compiling and executing `hello.c` with the ARC GCC toolkit (ELF32 version) as explained below.

Example 1 Listing of hello.c

```
#include <stdio.h>
int main()
{
       printf("Hello World \n");
       return 1;
}
```

Invoke the compiler to compile `hello.c` for ARC 700 (-mA7) with debug information (-g).

```
# arc-elf32-gcc  -mA7  -g  hello.c  -o  hello.out
```

The output file named `hello.out` is created.

Now try executing/debugging `hello.out` with simulator as follows

```
# arc-elf32-insight   hello.out
```

Insight begins by showing `hello.c` file.

1. Select Run -> Connect to target.

   A dialog box opens showing target connections.

2. Select Simulator as the target and click OK.
3. Place a breakpoint anywhere in main() and run.

You have now compiled and debugged a simple application for ARC 700 using the simulator.

Chapter 2 — Documentation

The following ARC documents are provided:

- *ARC GCC Release Notes*: Containing a summary of the release, and changes since the last release.
- *ARC GDB-Insight Getting Started*: Containing the ARC GNU Debugger with the Insight GUI.

The following online information is available:

- Installation Instructions
- Online Documentation

Installation Instructions

Particular installation instructions for the toolchain and any third party tools are contained in the readme file provided with the installation.

Online Documentation

The ARC GCC toolchain is supplied with built-in on-line documentation (info pages) that is visible after the toolchain is installed.

In addition, PDF documentation is located in `$UNZIP_DIR/docs`. The starting point is `contents.pdf`.

Chapter 3 — ARC-Specific Features of GCC

This section describes features which are specific to the ARC GCC toolchain.

Command Line options for ARC

This section details the command line options specific to the ARC GCC toolchain.

GCC Command-Line Options for ARC

The following options are defined for ARC implementations:

-EL	Compile code for little-endian mode. This is the default.
-mlittle-endian	
-EB	Compile code for big-endian mode.
-mbig-endian	
-mA6	Generate ARCompact 32-bit code for ARC 600 processor.
-mARC600	
-mA7	Generate ARCompact 32-bit code for ARC 700 processor.
-mARC700	This is the default for arc-linux-uclibc-gcc.
-mmixed-code	Generate ARCompact 16-bit instructions intermixed with 32-bit instructions for the ARC 600 and ARC 700 processors. With this option, 16-bit instructions are generated wherever possible.
-mvolatile-cache	Allow caching of volatile references.
-mno-volatile-cache	Do not cache volatile references.
-mno-cond-exec	Do not generate predicated instructions for conditional execution in `final_prescan_insn`. The target-independent passes that can generate conditional execution are not affected by this option.
-mnorm	Allow generation of **norm** instruction through the use of builtins. For ARC 700, the **-mnorm** option is turned on by default.

-mswap	Allow generation of swap instruction through the use of builtins. For ARC 700, the **–mswap** option is turned on by default.
-mmul64	Allow generation of **mul64** and **mulu64** instructions, through the use of builtins. Not available for ARC 700.
-mno-mpy	Disallow generation of ARC 700 multiplier instructions by the compiler. The ARC 700 multiplier instructions are enabled by default.
-mEA	Generate extended arithmetic instructions. Presently, only the generation of **divaw** with the help of compiler builtins is supported.
-msoft-float	Instruct the compiler to use software emulation for floating point operations. This is the default.
-mlong-calls	Generate all function calls as register indirect. By default, the compiler generates **bl** `<<s25 offset>>` instructions for function calls. This restricts the range of target addresses for a function call to a signed 25-bit offset, unless -mno-cond-exec' is in effect, function calls may be conditionalized, reducing s25 offsets to s21 offsets. To call functions placed at a higher offset, a register indirect jump-and-link instruction needs to be used. This switch instructs the compiler to call functions with a register-indirect and thus enables calling functions within the entire 32-bit address space. This flag can be overridden with the short_call function attribute.
-mno-brcc	Do not generate BRcc instructions for ARC700 target. The default behavior is to generate BRcc instructions with -mA7/-mARC7000 switch.
-mno-sdata	Do not generate code using the small-data sections. For ARCompact targets, the compiler allocates space for externally visible global variables in the small-data sections and generates GP register based loads/stores to access them.
-mno-millicode	Do not generate millicode for saving/restoring registers in functions prologue/epilogue. This flag is required only with -Os switch. For other optimization levels millicode thunk generation is disabled by default.
-mdynamic	Link with dynamic libraries, if these are available (only for the ARC-LINUX-UCLIBC build of the toolchain)
-mspfp	Generate single-precision FPX instructions. The instruction scheduling is
-mspfp_compact	done assuming latencies for the compact build of the FPX SPFP extensions.
-mspfp_fast	Generate single-precision FPX instructions. The instruction scheduling is done assuming latencies for the fast build of the FPX SPFP extensions.
-mdpfp	Generate double-precision FPX instructions. The instruction scheduling is
-mdpfp_compact	done assuming latencies for the compact build of the FPX DPFP extensions.
-mspfp_fast	Generate double-precision FPX instructions. The instruction scheduling is done assuming latencies for the Fast build of the FPX DPFP extensions.
-msimd	Allow generation of vector instructions, through the use of SIMD builtins provided by the compiler.

Assembler Command-Line options for ARC

The following options are available when GNU assembler **as** is configured for an ARC processor. For more information on GNU assembler, refer to the GNU GCC toolkit documentation.

-mA6 | -mARC600 | -mA7 | -mARC700	Explicitly select a variant of the ARC architecture. By default, **as** assumes **-mA7** and enables the ARC 700 instruction set
-EB | -EL	Any ARC configuration of **as** can select big-endian or little-endian output at run time (unlike most other GNU development tools, which must be configured for one or the other). Use **-EB** to select big-endian output and **–EL** for little endian.
-mno-mpy	Disable support for the ARC 700 extension multiplier instructions. Not valid for processor cores prior to ARC 700.
-mmul64	Enable support for the **mul64** and **mulu64** instructions and the auxiliary registers used by them.
-mnorm	Enable support for the normalize extension instruction.
-mswap	Enable support for the **swap** extension instruction.
-mEA	Enables support for the extended-arithmetic extension instructions, registers, and condition codes.
-mspfp | -mspfp_compact | -mspfp_fast	Enable support for single-precision FPX instructions.
-mdpfp | -mdpfp_compact | -mdpfp_fast	Enable support for double-precision FPX instructions.
-msimd	Enable support for SIMD extension instructions.

Assembler Directives Supported for ARC

The following directives are supported by the GNU assembler for ARC processors:

% A register name can optionally be prefixed by a % character. So register `%r0` is equivalent to `r0` in the assembly code.

@ Prefixing an operand with an @ specifies the operand to be a symbol and not a register. This is how the assembler disambiguates the use of an ARC register name as a symbol. So the instruction **mov** `r0`, `@r0` moves the address of symbol `r0` into register `r0`.

The following additional assembler directives have been added for Position Independent Code. These directives are available only with the ARC700 processor. These directives generate relocation entries which are interpreted by he linker in the manner described below.

@gotpc Relative distance of the symbol's GOT entry from the current pc location.

@gotoff Distance of the symbol from the base of the GOT.

@plt32 Distance of the symbol's PLT Entry to the current pc. This is valid only with branch and link instructions currently and pc relative calls.

__GLOBAL_OFFSET_TABLE__

Symbol referring to the base of the GOT .

__DYNAMIC__

An alias for GOT Base. These can be used only with @gotpc modifiers.

Linker command line options for ARC

The GNU linker for the ARC processors supports the following command line switches:

-marcelf Used for all binaries on ARC600 and standalone binaries on ARC700 (arc-elf32-ld).

-marcelf-prof Like –**marcelf** , but preserve special sections used for profiling.

-marclinux Used for Linux binaries on ARC 700. This allows linking with shared libraries. Thus for any program to link shared libraries with the ARC700 options, the **–marclinux** switch will have to be provided to the linker. This is the default for the ARC-LINUX-UCLIBC build of the tools (arc-linux-uclibc-ld).

-marclinux-prof Like **-marclinux**, but preserve special sections used for profiling.

GNU Run-Time Environment

This section covers the details of the runtime environment followed by GCC for the ARC 600 and ARC 700 processors

The following topics are covered:

Data Storage Types

The table below lists the data types, their sizes and range in the ARCtangent runtime environment.

Type	Size (bytes)	Alignment (bytes)	Range
char	1	1	0 to 255
signed char	1	1	-128 to 127
short	2	2	-32,768 to 32,767
unsigned short	2	2	0 to 65,535
int	4	4	-2^{31} to $2^{31}-1$
unsigned int	4	4	0 to $2^{32}-1$
long	4	4	-2^{31} to $2^{31}-1$
unsigned long	4	4	0 to $2^{32}-1$
long long	8	4	-2^{63} to $2^{63}-1$
unsigned long long	8	4	0 to $2^{64}-1$
float	4	4	$\pm (2^{-149}$ to $2^{127})$
double	8	4	$\pm (2^{-1074}$ to $2^{1023})$
long double	8	4	$\pm (2^{-1074}$ to $2^{1023})$

Register usage

GCC for ARC follows the register usage conventions specified by the ABI (as given in the table below).

Function	Registers
Result	R0-R1
Arguments	R0-R7
Caller Saved Registers	R0-R12
Callee Saved Registers	R13-R25
Static chain pointer (if required)	R11
Register for temp calculation	R12
Global Pointer	R26 (GP)
Frame Pointer	R27 (FP)
Stack Pointer	R28 (SP)
Interrupt Link Register 1	R29 (ILINK1)
Interrupt Link Register 2	R30 (ILINK2)
Branch Link Register	R31 (BLINK)

Stack Structure

The ARCtangent runtime environment uses a grow-down stack that contains return register, non-volatile registers, frame pointer, local variables, and a routine's parameter information.

Prologue and Epilogue

The called routine is responsible for allocating its own stack frame, making sure to preserve 4-byte alignment on the stack. This action is accomplished by a section of code called the prologue, which the compiler places before the body of the routine. After the body of the routine, the compiler generates an epilogue to restore the registers saved in the prologue code, previous stack frame and to return to the caller.

The compiler-generated prologue code does the following:

- Allocates space for register arguments in case of variadic function (functions with variable argument lists).
- Saves the return address register (BLINK)
- Saves required non-volatile general-purpose registers into the register save area
- Saves the caller's frame pointer (FP), if required, and sets the new frame pointer to this location

- Decrements the stack pointer (SP) to account for the new stack frame

At the end of the function, the compiler-generated epilogue does the following:

- Restores the stack pointer (SP) to the beginning of the saved register area
- Restores all non-volatile registers that were saved in the register area
- Restores caller's frame pointer register (FP), if required
- Restores the return address register (BLINK)
- Restores caller's stack pointer register (SP)
- Returns to caller through the address stored in the blink register.

NOTE *The return address save/restore is elided for leaf functions*

Parameter Passing

Parameters are passed by pushing them onto the stack. The first 8 words (32 bytes) of arguments are loaded into registers r0 to r7. The remaining arguments are passed by storing them into the stack immediately above the stack pointer register (sp).

Return Value

- In the ARCtangent runtime environment, function values are returned in register r0
- A doubleword result (double or long double) is returned in registers r0 to r1
- Structures that can be contained in a single 4-byte register are returned in r0
- All other structures are returned by storing them at an address passed to the function as a hidden first argument
- Sixty-four-bit integers (long long) are returned in registers r0 to r1

Position Independent Code

A different linker emulation mode for ARC700 with Linux for shared libraries is used (-marclinux)

[__GLOBAL_OFFSET_TABLE__]

contains the address of the dynamic table.

[__GLOBAL_OFFSET_TABLE__ + 4]

contains the module information filled in by the dynamic linker.

[__GLOBAL_OFFSET_TABLE__ + 8]

contains the address of the runtime symbol resolver filled in by the dynamic linker at load time.

The PLT0 Entry has been defined as follows.

● Absolute PLT0

  - ld %r11, [__GLOBAL_OFFSET_TABLE__+4]
  - ld %r10, [__GLOBAL_OFFSET_TABLE__+8]
  - j [%r10]
  - __GLOBAL_OFFSET_TABLE__
- Position Independent PLT0
  - ld %r11, [pcl,__GLOBAL_OFFSET_TABLE__@gotpc + 4]
  - ld %r10, [pcl,__GLOBAL_OFFSET_TABLE__@gotpc + 8]
  - j [%r10]
  - __GLOBAL_OFFSET_TABLE__
- The PLT*n* Entry has been defined as follows.
  - ld %r12, [pcl, func@gotpc]
  - j_s.d [%r12]
  - mov_s %r12, %pc
- DT_PLTGOT points to the first entry of the PLT table.

For ease of the dynamic linker to find the GOT base for every module the last entry in PLT0 has been fixed as to contain the GOT base address.

For more info on the dynamic linking aspects please refer to the System V ABI extension v 2.0.

Compiler Builtins and Attributes

This section details the language extensions supported for gaining low-level access to the ARC processors:

- Generic built-ins
- SIMD Built-ins
- Attributes

Generic built-ins

The GNU compiler for ARC processors supports the following built-in functions. The compiler recognizes these as language extensions. The intrinsics can be used in C code to generate the corresponding assembly instructions, as illustrated below:

void __builtin_arc_nop (void)

Usage: __builtin_arc_nop ();

Instruction generated: nop

int __builtin_arc_norm (int)

Usage: dest = __builtin_arc_norm (src);

Instruction generated: norm dest, src

short int __builtin_arc_normw (short int)

Usage: dest = __builtin_arc_normw (src);

Instruction generated: normw dest, src

int __builtin_arc_swap (int)

Usage: dest = __builtin_arc_swap (src);

Instruction generated: swap dest, src

void __builtin_arc_mul64 (int, int)

Usage: __builtin_arc_mul64 (a, b);

Instruction generated: mul64 a, b

This builtin is not available for ARC700.

void __builtin_arc_mulu64 (unsigned int, unsigned int)

Usage: __builtin_arc_mulu64 (a, b);

Instruction generated: mulu64 a, b

This builtin is not available for ARC700.

int __builtin_arc_divaw (int, int)

Usage: dest = __builtin_arc_divaw (src1, src2);

Instruction generated: divaw dest, src1, src2

void __builtin_arc_brk (void)

Usage: __builtin_arc_brk ();

Instruction generated: brk

void __builtin_arc_flag (unsigned int)

Usage: __builtin_arc_flag (a);

Instruction generated: flag a

void __builtin_arc_sleep (int)

Usage: __builtin_arc_sleep (a);

Instruction generated: sleep a

void __builtin_arc_swi (void)

Usage: __builtin_arc_swi ();

Instruction generated: swi

unsigned int __builtin_arc_core_read (unsigned int)

Usage: dest = __builtin_arc_core_read (a)

Instruction generated: mov dest, r<<a>>

Here, the operand is treated as the register number to be read. The argument 'a' should be a compile time constant.

void __builtin_arc_core_write (unsigned int, unsigned int)

Usage: __builtin_arc_core_write (a, b)

Instruction generated: mov r<<b>>, a

Here, the first operand is treated as the register number to be written. The second argument should be a compile time constant.

unsigned int __builtin_arc_lr (unsigned int)

Usage: dest = __builtin_arc_lr (a)

Instruction generated: lr dest, [a]

Here, the operand is treated as the auxiliary register address to be read. The argument should be a compile time constant.

void __builtin_arc_sr (unsigned int, unsigned int)

Usage: __builtin_arc_sr (a, b)

Instruction generated: sr a, [b]

Here, the first operand is treated as the auxiliary register address to be written. The argument 'b' should be a compile time constant.

NOTE If an intrinsic is not supported for a particular processor version, the call to the builtin is treated as a normal function call.

NOTE Out of the above set of builtins, the instructions generated by the following are not considered as candidates for scheduling, i.e. they are not moved around by the compiler during scheduling, and thus can be trusted to appear where they are put in the C code:

– __builtin_arc_brk

– __builtin_arc_flag

– __builtin_arc_sleep

– __builtin_arc_swi

– __builtin_arc_core_read

– __builtin_arc_core_write

– __builtin_arc_lr

– __builtin_arc_sr

SIMD Built-ins

SIMD builtins provided by the compiler can be used to generate the vector instructions. This section describes the available builtins and how to use them in programs.

With the -msimd switch passed, the compiler provides 128-bit vector types, which can be specified using the vector_size attribute. The header file <arc-simd.h> can be included to use the following predefined types.

```
typedef int   __v4si  __attribute__((vector_size(16)));
typedef short __v8hi  __attribute__((vector_size(16)));
```

These types can be used to define 128-bit variables. The built-in operations listed in the following section can be used on these variables to generate the vector operations.

A list of the SIMD built-ins grouped by their signatures follows:

NOTE For all built-ins **__builtin_arc_<someinsn>**, the header file arc-simd.h also provides equivalent macros called **_<someinsn>** which can be used for programming ease and improved readability. Also provided are the following macros for DMA control

```
#define _setup_dma_in_channel_reg  _vdiwr
#define _setup_dma_out_channel_reg _vdowr
```

a) Return type : __v8hi

First argument : __v8hi

Second argument : __v8hi

```
__v8hi __builtin_arc_vaddaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vaddw (__v8hi, __v8hi)
__v8hi __builtin_arc_vavb (__v8hi, __v8hi)
__v8hi __builtin_arc_vavrb (__v8hi, __v8hi)
__v8hi __builtin_arc_vdifaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vdifw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmaxaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmaxw (__v8hi, __v8hi)
__v8hi __builtin_arc_vminaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vminw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulfaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulfw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulw (__v8hi, __v8hi)
__v8hi __builtin_arc_vsubaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vsubw (__v8hi, __v8hi)
__v8hi __builtin_arc_vsummw (__v8hi, __v8hi)
__v8hi __builtin_arc_vand (__v8hi, __v8hi)
__v8hi __builtin_arc_vandaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vbic (__v8hi, __v8hi)
__v8hi __builtin_arc_vbicaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vor (__v8hi, __v8hi)
__v8hi __builtin_arc_vxor (__v8hi, __v8hi)
__v8hi __builtin_arc_vxoraw (__v8hi, __v8hi)
__v8hi __builtin_arc_veqw (__v8hi, __v8hi)
__v8hi __builtin_arc_vlew (__v8hi, __v8hi)
__v8hi __builtin_arc_vltw (__v8hi, __v8hi)
__v8hi __builtin_arc_vnew (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr1aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr1w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr2aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr2w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr3aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr3w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr4aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr4w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr5aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr5w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr6aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr6w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr7aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr7w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmrb (__v8hi, __v8hi)
__v8hi __builtin_arc_vh264f (__v8hi, __v8hi)
__v8hi __builtin_arc_vh264ft (__v8hi, __v8hi)
__v8hi __builtin_arc_vh264fw (__v8hi, __v8hi)
```

```
__v8hi __builtin_arc_vvc1f (__v8hi, __v8hi)
__v8hi __builtin_arc_vvc1ft (__v8hi, __v8hi)
```

b) Return type : __v8hi

First argument : __v8hi

Second argument : int

```
__v8hi __builtin_arc_vbaddw (__v8hi, int)
__v8hi __builtin_arc_vbmaxw (__v8hi, int)
__v8hi __builtin_arc_vbminw (__v8hi, int)
__v8hi __builtin_arc_vbmulaw (__v8hi, int)
__v8hi __builtin_arc_vbmulfw (__v8hi, int)
__v8hi __builtin_arc_vbmulw (__v8hi, int)
__v8hi __builtin_arc_vbrsubw (__v8hi, int)
__v8hi __builtin_arc_vbsubw (__v8hi, int)
```

c) Return type : __v8hi

First argument : __v8hi

Second argument : const int

The second argument in these builtins has to be an unsigned 3-bit integer constant, as it indicates the registers I0-I7:

```
__v8hi __builtin_arc_vasrw (__v8hi, const int)
__v8hi __builtin_arc_vsr8 (__v8hi, const int)
__v8hi __builtin_arc_vsr8aw (__v8hi, const int)
```

d) Return type : __v8hi

First argument : __v8hi

Second argument : const int

The second argument in these builtins has to be an unsigned 6-bit integer constant:

```
__v8hi __builtin_arc_vasrrwi (__v8hi, const int)
__v8hi __builtin_arc_vasrsrwi (__v8hi, const int)
__v8hi __builtin_arc_vasrwi (__v8hi, const int)
__v8hi __builtin_arc_vasrpwbi (__v8hi, const int)
__v8hi __builtin_arc_vasrrpwbi (__v8hi, const int)
__v8hi __builtin_arc_vsr8awi (__v8hi, const int)
__v8hi __builtin_arc_vsr8i (__v8hi, const int)
```

e) Return type : __v8hi

First argument : __v8hi

Second argument : const int

The second argument in these builtins has to be an unsigned 8-bit integer constant:

```
__v8hi __builtin_arc_vmvaw (__v8hi, const int)
__v8hi __builtin_arc_vmvw (__v8hi, const int)
__v8hi __builtin_arc_vmvzw (__v8hi, const int)
__v8hi __builtin_arc_vd6tapf (__v8hi, const int)
```

f) Return type : __v8hi

First argument : int

Second argument : const int

The second argument in these builtins has to be an unsigned 8-bit integer constant:

```
__v8hi __builtin_arc_vmovaw (int, const int)
__v8hi __builtin_arc_vmovw (int, const int)
__v8hi __builtin_arc_vmovzw (int, const int)
```

g) Return type : __v8hi

First argument : __v8hi

```
__v8hi __builtin_arc_vabsaw (__v8hi)
__v8hi __builtin_arc_vabsw (__v8hi)
__v8hi __builtin_arc_vaddsuw (__v8hi)
__v8hi __builtin_arc_vsignw (__v8hi)
__v8hi __builtin_arc_vexch1 (__v8hi)
__v8hi __builtin_arc_vexch2 (__v8hi)
__v8hi __builtin_arc_vexch4 (__v8hi)
__v8hi __builtin_arc_vupbaw (__v8hi)
__v8hi __builtin_arc_vupbw (__v8hi)
__v8hi __builtin_arc_vupsbaw (__v8hi)
__v8hi __builtin_arc_vupsbw (__v8hi)
```

h) Return type : void

First argument : int

Second argument : int

```
void __builtin_arc_vdirun (int, int)
void __builtin_arc_vdorun (int, int)
```

i) Return type : void

First argument : const int

Second argument : int

The first argument in these built-ins has to be an unsigned 3-bit integer constant, as it indicates DR0-DR7 DMA channel setup registers. The file arc-simd.h also provides defines which can be used in place of the DMA register numbers to facilitate better code readability:

```
void __builtin_arc_vdiwr (const int, int)
void __builtin_arc_vdowr (const int, int)
```

j) Return type : void

First argument : int

```
void __builtin_arc_vrec (int)
void __builtin_arc_vrun (int)
void __builtin_arc_vrecrun (int)
void __builtin_arc_vendrec (int)
```

k) Return type : __v8hi

First argument : __v8hi

Second argument : const int

Third argument : const int

The second argument in these built-ins has to be an unsigned 3-bit integer constant, as it indicates I0-I7 registers. The third argument has to be an unsigned 8-bit quantity. The file arc-simd.h also provides defines which can be used in place of the I0-I7 register numbers to facilitate better code readability:

```
__v8hi __builtin_arc_vld32wh (__v8hi, const int, const int)
__v8hi __builtin_arc_vld32wl (__v8hi, const int, const int)
__v8hi __builtin_arc_vld64 (__v8hi, const int, const int)
__v8hi __builtin_arc_vld32 (__v8hi, const int, const int)
```

NOTE Although the equivalent hardware instructions do not take a SIMD register as an operand, these built-ins overwrite the relevant bits of the __v8hi quantity provided as the first argument with the value loaded from [Ib, u8] location in the SDM.

l) Return type : __v8hi

First argument : const int

Second argument : const int

The first argument in these built-ins has to be an unsigned 3-bit integer constant, as it indicates I0-I7 registers.

The second argument has to be an unsigned 8-bit quantity. The file arc-simd.h also provides defines which can be used in place of the I0-I7 register numbers to facilitate better code readability:

```
__v8hi __builtin_arc_vld64w (const int, const int)
__v8hi __builtin_arc_vld128 (const int, const int)
```

m) Return type : void

First argument : __v8hi

Second argument : const int

Third argument : const int

The second argument in these built-ins has to be an unsigned 3-bit integer constant, as it indicates I0-I7 registers. The third argument has to be an unsigned 8-bit quantity The file arc-simd.h also provides defines which can be used in place of the I0-I7 register numbers to facilitate better code readability:

```
void __builtin_arc_vst128 (__v8hi, const int, const int)
void __builtin_arc_vst64 (__v8hi, const int, const int)
```

n) Return type : void

First argument : __v8hi

Second argument : const int

Third argument : const int

The second argument has to be an unsigned 3-bit quantity to identify the 16-bit sub-register to be stored.

The third argument in these built-ins has to be an unsigned 3-bit integer constant, as it indicates I0-I7 registers.

The fourth argument has to be an unsigned 8-bit quantity. The file arc-simd.h also provides defines which can be used in place of the I0-I7 register numbers to facilitate better code readability:

```
void __builtin_arc_vst16_n (__v8hi, const int, const int, const int)
void __builtin_arc_vst32_n (__v8hi, const int, const int, const int)
```

o) Return type : void

First argument : const int

The argument has to be an unsigned 6-bit quantity.

```
void __builtin_arc_vinti (const int)
```

Example

```
/* Add corresponding 16-bit elements of two 128-bit vectors
   Broadcast multiply the result vector by the integer argument.
*/

#include <arc-simd.h>

__v8hi vector_operand1;
__v8hi vector_operand2;
__v8hi vector_result;

int vec_func (int intarg)
{
  vector_result = _vaddw (vector_operand1,vector_operand2);
  vector_result = _vbmulw (vector_result, intarg);
}
```

Attributes

The following attributes for function names are supported in the ARC backend of GCC:

long_call — The function marked with this attribute are always called using register-indirect jump-and-link instructions, thereby enabling the called function to be placed anywhere within the 32-bit address space.

short_call — This attribute indicates that the called function is close enough to be called with a bl <<s25 offset>> instruction, i.e. within a signed 25-bit offset from the call site.

These attributes override the **-mlongcalls** switch, and thus calls to functions declared with 'short_call' attribute can use bl <<offset>> even if the **-mlongcalls** switch is passed (please refer to the example below).

NOTE Multiple function declarations need to have consistent attribute specification. A failure to do so is reported as an error.

Example of attribute usage:

```
extern void nearfunc() __attribute__((short_call));
extern void farfunc() __attribute__((long_call));
extern void nearfunc(); /* Error: Earlier declared as a short_call function */

void foo()
{
  nearfunc (); /* This can be called as 'bl nearfunc' */
  farfunc (); /* This will be called as 'mov reg, @farfunc jl [reg]' */
}
```

Chapter 4 — Using Inline Assembly

The GNU C compiler allows the user to write assembly code fragments within C code. The user can thus insert handwritten assembly code at the required points.

The generic usage information for inline assembly can be found in sections 5.35-5.38 of the gcc info pages present in the info subdirectory inside the installation location. This section describes the ARC specific constraints that can be used in **asm** instructions.

The following constraint characters can be used, when writing extended inline assembly for ARC:

'q' : For registers usable in 16bit ARCompact instructions

'r' : Any of the General Purpose registers

'x' : The register r0

'I' : 12-bit signed integer constant for ARC600 and ARC700

'J' : Any integer constant

'K' : 3-bit unsigned integer constant

'L' : 6-bit unsigned integer constant

'M' : 5-bit unsigned integer constant

'N' : Constant integer 1 (one)

'0' : 7-bit unsigned integer constant

'P' : 8-bit unsigned integer constant

Example: To return the result of norm instruction for a 6-bit configurable immediate constant, the following code can be used:

```
#define NORM_IMM 3
int normalize (void)
{
  asm (“norm %0, %1”:”=x” (retval) : /* output */ ”L” (NORM_IMM) /* input */);
  return retval;
}
```

The code generated for this function body under optimization is:

```
norm r0, 3
```

The constraint “=x” for retval makes sure that return value register r0 is given to retval, and the constraint “L” for the constant ensures that the immediate passed to the norm instruction is a valid 6-bit unsigned constant integer.

NOTE	The norm instruction can also be generated by using the compiler provided builtin __builtin_arc_norm.

Chapter 5 — Specifying ARC Extensions

An extension to the base core can be specified to the assembler by using the extension directives. The directives available on ARC are explained below:

.extAuxRegister NAME,ADDRESS,MODE

The ARC cores have extensible auxiliary register space. The auxiliary registers can be defined in the assembler source code by using this directive. The first parameter is the NAME of the new auxiliary register. The second parameter is the ADDRESS of the register in the auxiliary register memory map for the variant of the ARC. The third parameter specifies the MODE in which the register can be operated on and it can be one of:

- *r* (readonly)
- *w* (write only)
- *r*|*w* (read or write)

For example:

```
.extAuxRegister mulhi,0x12,w
```

This specifies an extension auxiliary register called _mulhi_ which is at address 0x12 in the memory space and which is only writable.

.extCondCode SUFFIX,VALUE

The condition codes on the ARC cores are extensible and can be specified by means of this assembler directive. They are specified by the suffix and the value for the condition code. They can be used to specify extra condition codes with any values. For example:

```
.extCondCode is_busy,0x14
add.is_busy r1,r2,r3
b.is_busy _main
```

.extCoreRegister NAME,REGNUM,MODE,SHORTCUT

Specifies an extension core register NAME for the application. This allows a register NAME with a valid *REGNUM* between 0 and 60, with the following as valid values for MODE

- *r* (readonly)
- *w* (write only)
- *r*|*w* (read or write)

The other parameter gives a description of the register having a *SHORTCUT* in the pipeline. The valid values are:

- *can_shortcut*
- *cannot_shortcut*

For example:

```
.extCoreRegister mlo,57,r,can_shortcut
```

This defines an extension core register mlo with the value 57 which can shortcut the pipeline.

.extInstruction NAME,OPCODE,SUBOPCODE,SUFFIXCLASS,SYNTAXCLASS

The ARC cores allows the user to specify extension instructions. The extension instructions are not macros. The assembler creates encodings for use of these instructions according to the specification by the user. The parameters are:

- *NAME* Name of the extension instruction. In case of the ARCompact, if the instruction name Is suffixed with **_s** it indicates a 16-bit extension instruction.
- *OPCODE* Opcode to be used. (Bits 27:31 in the encoding). Valid values are *0x10* to *0x1f* or *0x03*. In case of the ARCompact; for the 32-bit extension instructions valid values range from *0x04* to *0x07* (inclusive), and for 16-bit instructions valid values range from *0x08* to *0x0B* (inclusive).
- *SUBOPCODE* Sub-opcode to be used. Valid values are from *0x090x3f*. However the correct value also depends on *SYNTAXCLASS*.
- *SUFFIXCLASS* Determines the kinds of suffixes to be allowed. This parameter indicates the absence or presence of conditional suffixes and flag setting by the extension instruction. Valid values are *SUFFIX_NONE*, *SUFFIX_COND*, and *SUFFIX_FLAG*. It is also possible to specify that an instruction sets the flags *and* is conditional by using `SUFFIX_CODE | SUFFIX_FLAG`.
- *SYNTAXCLASS* Determines the syntax class for the instruction. It can have the following values:

  SYNTAX_NOP Two-operand instruction with no operands

  SYNTAX_1OP Two-operand instruction with one source operand

  SYNTAX_2OP Two-operand instruction with one source operand and one destination

  SYNTAX_3OP Three-operand Instruction with two source operands and one destination

  The syntax class also permits modifiers as described below:

  OP1_MUST_BE_IMM Modifies syntax class *SYNTAX_3OP*, specifying that the first operand of a three-operand instruction must be an immediate (i.e., the result is discarded). *OP1_MUST_BE_IMM* is used by bitwise **OR**ing it with *SYNTAX_3OP* as given in the example below. This is usually used to set the flags using specific instructions and not retain results.

  OP1_IMM_IMPLIED Modifies syntax class *SYNTAX_20P*, it specifies that there is an implied immediate destination operand which does not appear in the syntax.

  OP1_DEST_IGNORED Used with *OP1_MUST_BE_IMM* and *OP1_IMM_IMPLIED* when the instruction ignores the destination operand. It allows the assembler to choose a more efficient encoding of the instruction. GAS currently ignores this syntax class modifier.

For example:

```
.extInstruction mul64,0x14,0x0,SUFFIX_COND,SYNTAX_3OP | OP1_MUST_BE_IMM
```

In this case, the first argument is an implied immediate (that is, the result is discarded). This is as if the source code were `inst 0,r1,r2`. You use *OP1_IMM_IMPLIED* by bitwise **OR**ing it with *SYNTAX_20P*. For example, defining 64-bit multiplier with immediate operands:

```
.extInstruction mul64,0x14,0x0,SUFFIX_COND | SUFFIX_FLAG ,
SYNTAX_3OP|OP1_MUST_BE_IMM
```

The above specifies an extension instruction called **mul64** that has 3 operands, sets the flags, and can be used with a condition code, for which the first operand is an immediate (equivalent to discarding the result of the operation).

```
.extInstruction mul64,0x14,0x00,SUFFIX_COND, SYNTAX_2OP | OP1_IMM_IMPLIED
```

This describes a Two-operand instruction with an implicit first-immediate operand. The result of this operation is discarded.

Chapter 6 — Linker Scripts and Variables

The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. A linker script can also be used to instruct the linker to perform many other operations. The commands available for use in these scripts are documented in the **ld** info pages (**info ld**). This section provides an introduction to defining variables using linker scripts and using them in C and assembly code.

Defining the Symbols

Symbol assignments can be written as commands in their own right, or as statements within a **SECTIONS** command, or as part of an output section description in a **SECTIONS** command. The following example shows the three different places that symbol assignments may be used:

```
floating_point = 0;
SECTIONS
{
  .text :
  {
    PROVIDE(__provided_sym=.);
    *(.text)
    _etext = .;
  }
  _bdata = (. + 3) & ~ 3;
  .data : { *(.data) }
}
```

In this example, the symbol `floating_point` is defined as zero. The symbol `_etext` is given the same address as the location following the last `.text` input section. The symbol `_bdata` is defined as sharing its address with the location following the `.text` output section aligned upward to a four-byte boundary. In some cases, it is desirable for a linker script to define a symbol only if the sysbol is referenced and is not defined by any object included in the link. The **PROVIDE** keyword defines a symbol only if it is referenced but not defined. In the above example, the symbol `__provided_sym` is defined by the linker only if none of the input objects in the link command define it.

Referencing Linker-Defined Symbols

The linker-defined symbols are actually aliases to address locations, and thus the value assigned to them at link time can be obtained by taking their addresses, as shown in the example below:

```
int get_etext()
{
  return &_etext; /* &_etext = The value defined in the linker script */
}
```

Symbols Defined by the Default Linker Scripts

The following table lists some important symbols defined by the linker scripts, and the locations they alias to.

Symbol name	Address assigned to it
`__executable_start`	Start address of the executable
`etext, _etext, __etext`	End of text section
`edata, _edata`	End of data section
`__bss_start`	Start of bss section
`_end`	End of the executable
`__start_heap`	Start of the heap
`__end_heap`	End of the heap
`__stack`	Start address of the pre-allocated stack for standalone executables
`__stack_top`	End address of the pre-allocated stack for standalone executables