[Codito Technologies]

ARC GNU Tools User Guide

< www.codito.com/arc >

arc-support@codito.com

CONTENTS

CHAPTER 1.....5

COMMAND LINE OPTIONS FOR ARC.....5

1.1. GCC Options for ARC.....6
1.2.GNU Assembler Options for ARC.....9
1.2.1. Assembler Options.....9
1.2.2.Assembler Directives Supported. 10
1.3.Linker Commands 11

CHAPTER 2.....12

RUNTIME ENVIRONMENT 12

2.1. Data Types.....13
2.2. Register Usage.....14
2.3. ARCtangent Stack Structure.....15
2.4. Prolog and Epilog.....15
2.4.1.ARCtangent-A4 Prolog/Epilog.....15
2.4.2.ARCtangent-A5 ARC600 and ARC700 Prolog/Epilog.....16
2.5. Parameter Passing.....17
2.6. Return Value.....17
2.7. Position Independent Code.....17

CHAPTER 3.....19

COMPILER BUILTINS AND ATTRIBUTES.....19

3.1 Builtins.....19
3.2 Attributes.....21

CHAPTER 4.....23

USING INLINE ASSEMBLY.....23

CHAPTER 5.....25

SPECIFYING ARCHITECTURE EXTENSIONS 25

CHAPTER 6.....29

USING LINKER SCRIPTS FOR DEFINING VARIABLES.....29

6.1. Defining the symbols....29
6.2. Referencing the linker defined symbols in code....30
6.3. List of symbols defined by the default linker scripts....30
APPENDIX A....31
Using GNU Info....31
APPENDIX B....32
Building toolchain from the released sources....32
B.1. Building the toolchain using the scripts provided....32
B.2. Building the arc-linux-uclibc toolchain using crosstool....34

PREFACE

This manual provides an overview of the GNU C/C++ Compiler Toolkit command line options for the ARCtangent family of processors . It further explains the details of the runtime environment used for ARCtangent processors.

This document is intended to be used in conjunction with the on-line documentation provided with the GNU C/C++ Compiler toolkit for ARCtangent processors. Please refer to Appendix A for more information on using the GNU C/C++ compiler on-line documentation.

Appendix B describes the steps required to build the toolchain afresh from the source release.

Versions of tools provided in the release:

gcc	: 3.4.6
binutils	: 2.15
newlib	: 1.12.0
uClibc	: 0.9.27
crosstool	: 0.35

CHAPTER 1

Command Line Options for ARC

The following topics are discussed in this chapter.

- GCC Options for ARC
- GNU Assembler Options for ARC
- GNU Linker Options for the ARC

1.1. GCC Options for ARC

GCC command-line options specific to ARCtangent processor are discussed below. For information on all the GCC command-line options, refer to the section *"GCC command options"* under "*Using and Porting the GNU Compiler Collection (GCC)"* in GCC toolkit documentation provided as part of the release package (under <install path>/info directory).

These options are defined for ARC implementations:

-EL
-mlittle-endian
Compile code for little endian mode. This is the default.

-EB
-mbig-endian
Compile code for big endian mode.

-mA4
Generate code for ARCtangent-A4 processor. This is the default for arc-elf32-gcc and unsupported for arc-linux-uclibc-gcc.

-mA5
Generate ARCompact 32-bit code for ARCtangent-A5 processor.

-mA6
-mARC600
Generate ARCompact 32-bit code for ARC600 processor.

-mA7
-mARC700
Generate ARCompact 32-bit code for ARC700 processor.This is the default for arc-linux-uclibc-gcc.

-mmixed-code
Generate ARCompact 16-bit instructions intermixed with 32-bit instructions for the ARCtangent-A5, ARC600 and ARC700 processors. With this option, 16-bit instructions are generated wherever possible. Unless specified, the default architecture assumed with mixed-code is ARC700.

'-fPIC'

Generates Position Independent code (For ARC700 processor only). PIC is not supported for the ARCtangent-A4, ARCtangent-A5 and ARC600 processors.

-mtext=*text section*

-mdata=*data section*

-mrodata=*readonly data section*

Put functions, data, and readonly data in *text section*, *data section*, and *readonly data section* respectively. This can be overridden with the section attribute.

-malign-loops

Align loop starts to 32-byte boundaries (cache line size).

-mvolatile-cache

Allow caching of volatile references.

-mno-valatile-cache

Do not cache volatile references.

-mno-cond-exec

Do not generate predicated instructions for conditional execution.

-mnorm

Allow generation of norm instruction through the use of builtins. For ARC700, the -mnorm option is turned on by default.

-mswap

Allow generation of swap instruction through the use of builtins. For ARC700, the -mswap option is turned on by default.

-mbarrel_shifter

Allow generation of multiple shift instruction supported by barrel shifter unit. For post ARCtangent-A4 cores, such as ARCtangent-A5, ARC600, ARC700, the -mbarrel_shifter option is turned on by default.

NOTE: The barrel shifter instruction is not part of the base case ARCtangent-A4 core, and thus the following extension instructions need to be specified as inline assembly in the C file being compiled for an ARCtangent-A4 target.

.extInstruction asl,0x10,0x00,SUFFIX_COND|
SUFFIX_FLAG,SYNTAX_3OP

.extInstruction lsl,0x10,0x00,SUFFIX_COND|
SUFFIX_FLAG,SYNTAX_3OP

.extInstruction lsr,0x11,0x00,SUFFIX_COND|
SUFFIX_FLAG,SYNTAX_3OP

.extInstruction asr,0x12,0x00,SUFFIX_COND|
SUFFIX_FLAG,SYNTAX_3OP

.extInstruction ror,0x13,0x00,SUFFIX_COND|
SUFFIX_FLAG,SYNTAX_3OP

-mmul64

Allow generation of mul64 and mulu64 instructions, through the use of builtins. Not available for ARC700.

-mno-mpy

Disallow generation of ARC700 multiplier instructions from the compiler. The ARC700 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.

-mmin_max

Allow generation of min and max instructions for ARCtangent-A4 also. For ARCtangent-A5, ARC600 and ARC700 cores, the compiler generates the min and max instructions by default.

-msoft-float

Dummy flag to ease building of applications, which use the flag generically across all architectures. The floating point arithmetic is always performed via software emulation routines on ARC cores, and so the flag is ignored by the compiler.

-mlong-calls

Generate all function calls as register indirect. By default, the compiler generates *bl <<s25 offset>>* instruction for function calls. This restricts the range of target addresses for a function call to a signed 25 bit offset. 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.

1.2. GNU Assembler Options for ARC

1.2.1. Assembler Options

The following options are available when GNU assembler “as” is configured for an ARC processor. For more information on GNU assembler, refer to *“Using as”* in GNU C Compiler toolkit documentation.

-mA4 | -mA5 | -mA6 | -mARC600 | -mA7 | -mARC700
Explicitly select a variant of the ARC architecture. By default, `as' assumes `-mA4' and enables the ARCtangent-A4 instruction set (ARC base).

`-mA5', `-mA6'/'-mARC600' and `-mA7'/'-mARC700' enable the ARCompact 16-bit 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.

-mbarrel_shifter
Enable support for the barrel shifter extension for ARCtangent-A4.

-mno-mpy
Disnable support for the ARC700 extension multiplier instructions. Not valid for pre-ARC700 cores.

-mmul64
Enable support for the mul64 and mulu64 instructions and the auxiliary register 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.

-mmin_max

Enable support for min and max as extension instructions for ARCtangent-A4 core. Has no effect with later cores.

1.2.2. Assembler Directives Supported.

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 as 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

Besides these, the following 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.

1.3. Linker Commands

The GNU linker for the ARC processors supports the following emulation modes:

1. arcelf : Used for all binaries on A4, A5, ARC600 and standalone binaries on ARC700. This is the default for A4, A5 and ARC600.
2. arclinux: Used for linux binaries on ARC700. 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 ARC700.

CHAPTER 2

Runtime Environment

This chapter covers the details of the runtime environment followed by GCC for ARCtangent-A4 and ARCtangent-A5 and higher variants.

The following topics are covered:

- Data storage types
- Register Usage
- Stack structure
- Parameter Passing
- Return Value

2.1. Data Types

Table 2-1 lists the data types, their sizes and range in the ARCtangent runtime environment.

Table 2-1: Data Types 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	-32768 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 $^{1023)}$

2.2. Register Usage

GCC for ARC follows the register usage conventions specified by the ABI (as given in the table 2-2 below).

Table 2-2: Register Conventions

Function	Registers
Result	r0
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 (gp)	r26
Frame Pointer (fp)	r27
Stack Pointer (sp)	r28
Interrupt Link Register (ilink1)	r29
Interrupt Link Register (ilink2)	r30
Branch Link Register (blink)	r31

2.3. ARCtangent 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.

2.4. Prolog and Epilog

The called routine is responsible for allocating its own stack frame, making sure to preserve 8-byte alignment on the stack. This action is accomplished by a section of code called the **prolog,** which the compiler places before the body of the routine. After the body of the routine, the compiler generates an **epilog** to restore the registers saved in the prolog code, previous stack frame and to return to the caller.

2.4.1. ARCtangent-A4 Prolog/Epilog

The stack back-trace data structure is a 16-byte structure which is used to save the return register (blink, 4 bytes), the frame pointer register (fp, 4-bytes) and 8-bytes is reserved.

The compiler-generated **prolog** 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 the caller's frame pointer (fp), if required, and sets the new frame pointer to this location
- Decrements the stack pointer to account for the new stack frame
- Saves required non-volatile general-purpose registers into the register save area
- Stores the arguments above the stack back-trace data structure

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

- Restores all non-volatile registers that were saved in the register save area
- Restores the return address register (blink)
- Restores the caller's frame pointer register (fp), if required
- Restores the caller's stack pointer register (sp) to its previous value
- Returns to caller through the address stored in the blink register

2.4.2. ARCtangent-A5 ARC600 and ARC700 Prolog/Epilog

The compiler-generated **prolog** 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 to account for the new stack frame

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

- Restores the stack pointer 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.

2.5. Parameter Passing

Parameters are passed by pushing them onto the stack. First 8 words (32 bytes) of arguments are loaded into registers r0 to r7. For ARCtangent-A4, the remaining arguments are passed by storing them into the stack immediately above the stack back-trace data structure. For ARCtangent-A5 and higher variants, the remaining arguments are passed by storing them into the stack immediately above the stack pointer register (sp).

2.6. Return Value

- In the ARCtangent runtime environment, function values are returned in register **r0**
- A double-word 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**

2.7. 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 PLTn 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 SystemV ABI extension v 2.0

2.8. Reference

1. For more information on GNU C Compiler Toolkit, please refer to the gcc info pages included in the release.

2. For more on the Application Binary Interface for Dynamic Linking please refer to the extension for the SYSv ELF document v2.0. (arcabiv2).

3. For a list of known problems and limitations, please refer to the release notes included in the release. (arc-gnu-tools-relnotes-20060612.codito.txt)

CHAPTER 3

Compiler builtins and attributes

3.1 Builtins

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

1. void __builtin_arc_nop (void)
   Usage: **__builtin_arc_nop ();**
   Instruction generated: **nop**

2. int __builtin_arc_norm (int)
   Usage: **dest = __builtin_arc_norm (src);**
   Instruction generated: **norm dest, src**
   The -mnorm flag needs to be passed to the compiler for generation of norm instruction through the builtin.

3. short int __builtin_arc_normw (short int)
   Usage: **dest = __builtin_arc_normw (src);**
   Instruction generated: **normw dest, src**
   The -mnorm flag needs to be passed to the compiler for generation of norm instruction through the builtin.

4. int __builtin_arc_swap (int)
   Usage: **dest = __builtin_arc_swap (src);**
   Instruction generated: **swap dest, src**
   For pre-ARC700 cores, the -mswap flag needs to be passed to the compiler for generation of swap instruction through the builtin.

5. void __builtin_arc_mul64 (int, int)
   Usage: **__builtin_arc_mul64 (a, b);**
   Instruction generated: **mul64 a, b**
   The -mmul64 flag needs to be passed to the compiler for generation of mul64 instruction through the builtin. This builtin is not available for ARC700.

6. void __builtin_arc_mulu64 (unsigned int, unsigned int)
   Usage: **__builtin_arc_mulu64 (a, b);**

Instruction generated: **mulu64 a, b**
The -mmul64 flag needs to be passed to the compiler for generation of mulu64 instruction through the builtin. This builtin is not available for ARC700.

7. int __builtin_arc_divaw (int, int)
   Usage: **dest = __builtin_arc_divaw (src1, src2);**
   Instruction generated: **divaw dest, src1, src2**
   The -mEA flag needs to be passed to the compiler for generation of divaw instruction through the builtin. This builtin is not available for ARCtangent-A4.

8. void __builtin_arc_brk (void)
   Usage: **__builtin_arc_brk ();**
   Instruction generated: **brk**

9. void __builtin_arc_flag (unsigned int)
   Usage: **__builtin_arc_flag (a);**
   Instruction generated: **flag a**

10. void __builtin_arc_sleep (int)
    Usage: **__builtin_arc_sleep (a);**
    Instruction generated: **sleep** *for ARCtangent-A4 processor*
    sleep a *otherwise*
    The argument passed to the builtin function is ignored for ARCtangent-A4 processor.

11. void __builtin_arc_swi (void)
    Usage: **__builtin_arc_swi ();**
    Instruction generated: **swi**

12. 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.

13. 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.

14. 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.

15.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.

NOTES:

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

2. 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*

3.2 Attributes

The following attributes for function names are supported in the ARC back-end of GCC:

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

2. 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 '-mlong-calls' switch, and thus calls to functions declared with 'short_call' attribute can use *bl <<offset>>* even if the '-mlong-calls' 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 (ref: Appendix A – Using GNU info). 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 16-bit ARCompact instructions
'r'	:	Any of the General Purpose registers
'x'	:	The register r0
'I'	:	Any integer constant in the range (-256, 255) For A4 12-bit signed integer constant for A5, 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*** *(ref. Chapter 3, Compiler builtins).*

CHAPTER 5

Specifying Architecture 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 is 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
```

```
bis_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

0x10-0x1f or 0x03. In case of the ARCompact, for the 32-bit extension instructions valid values range from 0x04-0x07 (inclusive) and for 16-bit instructions valid values range from 0x08-0x0B (inclusive).

***SUBOPCODE**

Subopcode to be used. Valid values are from 0x09-0x3f. However the correct value also depends on SYNTAXCLASS

***SUFFIXCLASS**

Determines the kinds of suffixes to be allowed. Valid values are `SUFFIX_NONE', `SUFFIX_COND', `SUFFIX_FLAG' which indicates the absence or presence of conditional suffixes and flag setting by the extension instruction. 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"
2 Operand Instruction, with no operands. Available only for ARCompact.

``SYNTAX_1OP"
2 Operand Instruction, with 1 source operand. Available only for ARCompact.

``SYNTAX_2OP"
2 Operand Instruction, with 1 source operand and one destination.

``SYNTAX_3OP"
3 Operand Instruction, with 2 source operands and one destination.

In addition there could be modifiers for the syntax class as described below:

Syntax Class Modifiers are:
``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 ORing it with SYNTAX_3OP as given in the example below. This could usually be 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.

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

It really means that the first argument is an implied immediate (that is, the result is discarded). This is the same as though the source code were: inst 0,r1,r2. You use OP1_IMM_IMPLIED by bitwise ORing 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 which has 3 operands, sets the flags, can be used with a condition code, forwhich 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 2 operand instruction with an implicit firstimmediate operand. The result of this operation would be discarded.

CHAPTER 6

Using Linker Scripts for defining 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. Besides this, the 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 under the section “Linker Scripts” in the ld info pages (info ld). This section provides an introduction to defining variables using linker scripts and using them in C/assembly code.

6.1. 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 showing 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' will be defined as zero. The symbol `_etext' will have the same address as the location following the last `.text' input section. The symbol `_bdata' will be defined as sharing its address with the location following the `.text' output section aligned upward to a 4 byte boundary.

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in

the link. The `PROVIDE' keyword may be used to define a symbol only if it is referenced but not defined. e.g. In the above example, the symbol __provided_sym will be defined by the linker only if none of the input objects in the link command define it.

6.2. Referencing the linker defined symbols in code

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 */
}
```

6.3. List of 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	*Adress 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 stand-alone executables
__stack_top	End address of the pre-allocated stack for stand-alone executables

Appendix A

Using GNU Info

GNU Info is a program, which can be used for reading info documents. The GNU Project distributes most of its on-line manuals in the Info format.

To access **GNU Info** or **Info help system**, use the following command.

info -f filename

The info files for GNU C Compiler Toolkit are installed under $(INSTALLDIR)/info directory. For example, GCC documentation can be accessed via info as follows:

info –f /usr/local/arc/info/gcc.info

Appendix B

Building toolchain from the released sources

The toolchain presently has the following options for building:

- Manual build for elf32 or linux target using the provided scripts
- arc-linux-uclibc build (done using crosstool)

This appendix describes the two build procedures in detail.

B.1. Building the toolchain using the scripts provided

The source release includes a subdirectory named 'scripts', that contains the shell scripts needed to build the toolchain from source manually, i.e. with human intervention needed while building.

B.1.1. Preparing the sources

Let us assume that the directory 'SRC_ROOT' is used to unpack the sources. (SRC_ROOT can be any directory other than the directory where the tools will be installed)

```
# Go to the source root directory where the final release sources are
# placed and untar the sources
$ cd $SRC_ROOT
$ tar xvjf arc-gnu-tools-src-20060612.tar.bz2

# Enter the combined source directory and untar individual tarballs
$ cd arc-gnu-tools-src-20060612
$ tar xvfj gcc-3.4.x.tar.bz2
$ tar xvjf binutils-2.15.tar.bz2

# Untar uClibc if building arc-linux-uclibc toolchain
$ tar xvjf uClibc-0.9.27.tar.bz2
```

B.1.2. Preparing the install directory

Before starting to configure and build the compiler, the installation directory must be created with write permissions to the developer building the tools.

B.1.3. Building and installing binutils

The build script provided takes care of configuring the binutils

correctly. However, the developer must provide the source path, install path and target name (arc-linux-uclibc or arc-elf32) to the script for it to build the required configuration.

```
$  cd $SRC_ROOT/arc-gnu-tools-src-20060612/scripts
$  ./build_binutils  <source-path> <install-path> <target>
```

where,

<source-path> - top-level directory under which GNU sources are available (i.e 'SRC_ROOT'/arc-gnu-tools-src-20060612)
<install-path> - directory where the tools will be installed.
<target> - Name of the target (arc-linux-uclibc or arc-elf32)

This builds and installs arc-linux-uclibc/arc-elf32 binutils in the specified <install-path>

B.1.4. Building and installing uClibc *[skip if building arc-elf32 target]*

The uclibc based build of GCC depends on some header files provided by uClibc for completing succesfully. Thus, uClibc must be built and installed before attempting to build arc-linux-uclibc-gcc. Besides the usual configuration options for uClibc (i.e. the processor type and the kernel header files' location), the following must be specified for successfully building the toolchain:

1. Target CPU has a floating point unit (FPU) : Disable
2. Enable full C99 math library support : Enable
3. Generate Position Independent Code : Disable
4. Library Installation options:

i) uClibc runtime library directory : <install-path>
(e.g. /usr/local/arc-linux-uclibc)
ii) uClibc development environment directory : <install-path>/arc-linux-uclibc
(e.g. /usr/local/arc-linux-uclibc/arc-linux-uclibc)

Also, for building uClibc the first time, you need to have an arc-elf32 toolchain present in the PATH.

After menuconfig, the libraries can be built and installed.

```
$ make CROSS=arc-elf32-
$ make install
```

B.1.5. Building and installing GCC

After installing binutils and uClibc (for arc-linux-uclibc target), the gcc

build can be done.

The build_gcc script takes care of most of the configuration parameters required. The developer however, needs to provide it with the source-path, install-path and the target name (arc-linux-uclibc or arc-elf32)

```
$  ./build_gcc      <source-path> <install-path> <target>
```

This would build and install arc-linux-uclibc-gcc/arc-elf32-gcc in the install-path

NOTE: Make sure that the install path is the same for both binutils and gcc

B.2. Building the arc-linux-uclibc toolchain using crosstool

The arc-linux-uclibc toolchain can also be built using the crosstool provided in the source release, which has been adapted for the uclibc based ARC700 target.

B.2.1. *Preparing the sources*

Let us assume that the directory 'SRC_ROOT' is used to unpack the sources. (SRC_ROOT can be any directory other than the directory where the tools will be installed)

```
# Go to the source root directory where the final release sources are
# placed and untar the sources
$ cd $SRC_ROOT
$ tar xvjf arc-gnu-tools-src-20060612.tar.bz2

# Enter the combined source directory and untar Crosstool
$ cd arc-gnu-tools-src-20060612
$ tar xvjf crosstool-0.35.tar.bz2
```

B.2.2. Setting up the sources in locations expected by crosstool

Crosstool expects the source tarballs required for the build to be present in ($HOME)/downloads directory. Thus before invoking crosstool, the developer needs to copy the gcc-3.4.x.tar.bz2 , binutils-2.15.tar.bz2 and uClibc-0.9.27.tar.bz2 into a sub-directory named 'downloads' in his home directory.

Also, the tarball containing kernel sources needed to build uClibc, needs to be present in $(HOME)/downloads as linux-2.4.32.tar.bz2

NOTE: This implies that the ARC700 linux kernel source tarball released by Codito has to be renamed as linux-2.4.32.tar.bz2 and moved into the downloads directory in the developer's $(HOME)

```
# Required contents of $HOME/downloads after this step
$  ls $HOME/downloads
```

binutils-2.15.tar.bz2
gcc-3.4.x.tar.bz2
linux-2.4.32.tar.bz2
uClibc-0.9.27.tar.bz2

B.2.3. Checking the install directory

The toolchain will install at /opt/crosstool/gcc-3.4.x-uClibc-0.9.27/arc-linux-uclibc. Thus, the developer should have sufficient permission to create subdirectories within /opt (or /opt/crosstool).

B.2.4. Invoking crosstool to build the toolchain

Once the tarballs have been placed in $HOME/downloads, and the check for the install directory has been made, the demo-arc-uclibc.sh script present in the crosstool sources can be invoked to build and install the arc-linux-uclibc toolchain.

```
# Enter the crosstool directory and start the build
$ cd crosstool-0.35
$ ./demo-arc-uclibc.sh
```

This builds the toolchain in build/arc-linux-uclibc/gcc-3.4.x-uClibc-0.9.27 and installs in /opt/crosstool/gcc-3.4.x-uClibc-0.9.27/arc-linux-uclibc

- * -